Lightweight Analysis of Acyclic Unshared Lists

Roman Manevich[2], Shuvendu K. Lahiri[1], and Mooly Sagiv[2]

[1] Microsoft Research, shuvendu@microsoft.com
[2] Tel Aviv University, {rumster,msagiv}@tau.ac.il

Abstract. We describe a simple analysis for tracking properties such as may-aliasing, memory leaks, and disjointness for programs manipulating singly-linked lists. We restrict the set of programs we consider to ones that manipulate acyclic and unshared lists. We benefit from these restrictions in terms of simplicity and efficiency of the algorithm.
We demonstrate that most common list-manipulating programs satisfy the above restrictions or can be locally transformed to meet the requirements. Our algorithm successfully answers may-aliasing, memory leak and disjointness queries for these programs. The analysis also allows us to prove interesting *summary content* properties that relate the contents of a set of input lists to a procedure with the content of lists returned from the procedure.

1 Introduction

This work is motivated by the need to precisely check properties such as may-aliasing, disjointness, and absence of memory leaks for programs manipulating dynamically-allocated memory. For programs containing recursive and linked data structures, such as trees and lists, checking these properties is known to be undecidable even when all program conditions are treated non-deterministically [14, 23, 5]. Therefore, several different approximation techniques have been developed [7, 19, 1, 13].

We restrict our attention to singly-linked lists, which are very common in many applications. It is easy to adopt the proof of Chakaravathy [5](see Appendix A for details) to show that the problem of deciding may-aliasing for such programs (with uninterpreted program conditions) remains undecidable, even in the absence of any *heap sharing*.[3]

In this paper, we provide a lightweight sound analysis for checking may-aliasing, disjointness, and memory-leak properties for acyclic and unshared lists. Our work is motivated by the observation that most programs manipulating acyclic linked lists do not create complicated sharing patterns. Most examples either operate on completely unshared lists, or create very restricted sharing temporarily. For most of the latter cases, one can eliminate the sharing by simply rearranging the statements locally inside a basic block.

[3] Heap sharing occurs when there exist two distinct cells u and v in the heap, such that $u.\texttt{next} = v.\texttt{next}$.

Our approach is based on predicate abstraction [9], where we create a conservative approximation of the underlying program based on a set of predicates. The approach differs from other predicate abstraction based approaches in two ways: (1) the shape of the predicates is fixed and only parameterized by the set of program variables, and (2) we manually derive the abstract transformers for the underlying abstract system induced by the predicates. The advantage of the latter is that we do not require a theorem prover for computing the abstract transformer. Assuming the update formulae are polynomially bounded in the number of variables in the input program (which is true in our case), the complexity of our analysis is *output sensitive* — it is polynomial in the number of abstract reachable states. This is very useful, as the set of abstract reachable states is usually small for many programs that arise in practice.

We also extend our analysis to prove simple *summary content* properties for list-manipulating programs. A summary content property relates the contents of various pointer variables (the set of nodes reachable from the pointer) at the start of a procedure to the contents of pointers at the termination of the procedure. For example, this allows us to prove that the content of a list at the termination of a procedure is the same as the union of the contents of a set of input lists. We augment the predicates to conservatively record additional *may-flow* information between variables — if *may-flow*(`x`, `y`) is true, then the contents of `x` at the start of the procedure may have flowed into the content of `y` during the procedure. We successfully prove interesting content properties for a number of common list manipulating programs.

We have implemented a prototype of the algorithm using the TVLA [17][4] system and have shown that it is able to check properties for several example programs that display common usage of singly-linked lists. In particular, we are able to handle all the examples with acyclic lists present in [19]. We benefit from the restrictions on the shapes of the list in terms of both the number of predicates required and the complexity of the abstract transformer, compared to previous approaches [19, 1].

The rest of the paper is organized as follows: In Sec. 1.1, we describe the running example for the paper. In Sec. 2, we describe the programming language of lists and basics of predicate abstraction. Sec. 3.4 describes the predicates and the abstract transformers used by the analysis. In Sec. 4, we describe the summary content properties. Sec. 5 describes the implementation and empirical evaluation of our analysis on a number of common list programs.

1.1 Running Example

Fig. 1 shows a method that accepts as input two sorted lists and returns a sorted list containing the elements of these lists. This program serves as the running example in the paper.

The goal of the analysis is to prove the following properties for this method: (i) the method never dereferences a null pointer, (ii) the output list is unshared

[4] We used TVLA mainly as a convenient way to specify the updates independently from the 3-valued first order logic mechanisms of the system.

and acyclic, (iii) the method does not create a memory leak, and (iv) the set of cells in the output list is the union of the sets of cells of the two input lists.

```
      List Merge(List p, List q) {
L0:     if (p == null) return q;
L1:     if (q == null) return p;
L2:     if (p.data < q.data) { h = p; p = p.n; }
L3:     else { h = q; q = q.n; }
L4:     t = h;
L5:     while (p != null && q != null) {
L6:         if (p.data < q.data) {
L7:             t.n = p;
L8:             temp = p.n;
L9:             p = temp;
            }
L10:        else {
L11:            t.n = q;
L12:            temp = q.n;
L13:            q = temp;
           }
L14:       t = t.n;
        }
L15:    if (p != null) { t.n = p; }
L16:    else if (q != null) { t.n = q; }
L17:    return h;
    }
```

Fig. 1. A C# method that merges two singly-linked list into one list

2 Background

In this section, we define a simple programming language for singly-linked lists, and the concrete semantics. We also describe the basics of predicate abstraction.

2.1 A Simple Programming Language of Singly-Linked Lists

Our language contains a single data type *List* (representing a singly-linked list) declared as

`class List {List n; int data;}`.

The programming language conservatively ignores the data field of the list type.

Let *PVar* be a set of variables of type *List*. A *program* is a control flow graph (CFG), where each node in the CFG represents a control location and there is an entry node *entry*, and an exit node *exit*. The edges in the CFG represent either *assignment statements*, *assume* statements, or *assert* statements.

There are five types of assignment statements: (1) `x` := `new List()`, (2) `x` := `null`, (3) `x` := `y`, (4) `x` := `y.n`, and (5) `x.n` := `y`. Control flow is achieved by using the *assume* statements. An assume statement is of the form `assume` (`x` == `y`)

or `assume` $(\texttt{x}! = \texttt{y})$, and an assert statement is of the form `assert` $(\texttt{x} == \texttt{y})$ or `assert` $(\texttt{x}! = \texttt{y})$.

We also assume that programs in this language are normalized in the following way. Every statement of the form `x=y` or `x=y.n` is preceded by a statement `x=null`; every statement of the form `x.n=y` is preceded by a statement of the form `x.n=null`; in every statement of the form `y=x.n` or `x.n=y` the variables `x` and `y` are different. Finally, every statement that dereferences a variable `x` is preceded by an assert statement `assert` $(\texttt{x}! = \texttt{null})$. These assumptions are not restrictive, as it is possible to preprocess programs in linear time into the normalized form by introducing new temporaries, however they do simplify the specification of the abstract semantics shown later.

Semantics. A concrete state of the program σ is an evaluation of: (i) each variable $\texttt{x} \in PVar$ as an object of type *List*, and (ii) the field `n` as a function n from *List* to *List*. We often refer to an object of type *List* as a (list) cell. We denote x as the value of a program variable $\texttt{x} \in PVar$ in a given state. For any state, we require that $n(null) = null$.

We associate a transition function $[\![st]\!]$ with every statement st in the program. Each statement st takes a concrete state σ, and transforms it to a state $\sigma' = [\![st]\!](\sigma)$. The concrete semantics of program statements is shown in Table 1. The semantics is insensitive to the possible existence of garbage cells (i.e., cells that are not reachable from any variable). This is achieved by restricting the the variables that appear in all first-order formulae to range exclusively over non-garbage cells. This restriction is implicitly used throughout this paper.

Table 1. Concrete semantics of program statements. Unprimed symbols denote values before a statement is executed and primed symbols denote post-execution values

Statement	**Semantics**
`x = new List()`	$x' = v_{new}$, $n' = \lambda\, v\,.\,(v = v_{new}\ ?\ null\ :\ n(v))$ where v_{new} is a fresh cell
`x = null`	$x' = null$
`x = y`	$x' = y$
`x = y.n`	$x' = n(y)$
`x.n = null`	$n' = \lambda\, v\,.\,(v = x\ ?\ null\ :\ n(v))$
`x.n = y`	$n' = \lambda\, v\,.\,(v = x\ ?\ y\ :\ n(v))$
`assert(x != y)`	if $x \neq y$ then nop else abort with error message
`assert(x == y)`	if $x = y$ then nop else abort with error message
`assume(x != y)`	if $x \neq y$ then nop else ignore execution
`assume(x == y)`	if $x = y$ then nop else ignore execution

2.2 Predicate Abstraction

Predicate abstraction [9] is a scheme for performing *abstract interpretation* [6] of programs. A *predicate* P_i is a Boolean expression over the signature $\Sigma \cup PVar$, where Σ consists of a set of interpreted function or predicate symbols.

Given a set of predicates $P = \{P_1, \ldots, P_n\}$, predicate abstraction approximates the concrete set of states by Boolean combinations of predicates in P. An *abstract* state s_a is an evaluation of the predicates in P to Boolean values $\{0, 1\}$. Alternately, we can view an abstract state as a conjunction $\bigwedge_{P_i \in P} \tilde{P_i}$, where $\tilde{P_i} \subseteq \{P_i, \neg P_i\}$. Such a conjunction is called a *cube* over P. Given a concrete state σ, the abstraction operation α maps σ to an abstract state by evaluating the predicates on σ. The operation γ maps a abstract state to the set of concrete states it represents. We define α and γ for a set of states by pointwise extension of the operations for individual elements.

Given a program and a set of predicates P, we can define an abstract transition relation $[\![st]\!]^{\#}$ for each statement *st* in the program as follows:

$$[\![st]\!]^{\#}(s_a) = \{\alpha(t_a) \mid t_a \in [\![st]\!](\gamma(s_a))\} \ .$$

We also refer to the abstract transition relation as the *best transformer*.

Predicate abstraction provides a *conservative* approximation of the underlying concrete system — if we prove a property ϕ on the abstract program, then the property also holds for the underlying concrete program. However, a counterexample to ϕ in the abstract system need not be realizable in the concrete system.

3 A Static Analysis for Acyclic Unshared Lists

In this section, we define a static analysis based on predicate abstraction for programs in the language described in Sec. 2.1. We start by giving basic definitions needed to formally define acyclic unshared lists. We then describe a new predicate abstraction for acyclic unshared lists and explain how to express different properties over the predicates. Finally, we provide the abstract semantics for every program statement.

3.1 Definitions

In the sequel, we will use the notation n^j to denote the jth application of the function n. The relation n^* stands for the reflexive transitive closure of n, i.e., $n^*(u, v)$ is true if and only if there exists a $j \geq 0$ such that $n^j(u) = v$, and n^+ stands for the non-reflexive transitive closure of n, i.e., $n^+(u, v)$ is true if and only if there exists a $j > 0$ such that $n^j(u) = v$.

Definition 1. *We say that a heap is* acyclic*, when for every cell u of type List (apart from null), $n^+(u, u)$ is false.*

Definition 2. *We say that a heap is* unshared*, when for every pair of cells u and v (apart from null) of type List, $n(u) = n(v) \Leftrightarrow (u = v \vee n(u) = null)$.*

3.2 A Predicate Abstraction for Unshared Acyclic Lists

We use the predicates shown in Table 2, which are parameterized by the set of program variables *PVar*, to abstract concrete program states. The predicates of the form *Reach*[x, y] and *Aliased*[x, y] determine which non-null variables point to the same lists and capture the ordering between variables that point to the same lists. The predicates of the form *NotNull*[x] are needed for detecting null dereference situations and, along with *Aliased*[x, y], for interpreting conditions. The predicates of the form *Next*[x, y] are useful since lists are most often mutated by pairs of variables that reference consecutive list cells. The predicates of the form *NextNull*[x] are useful for detecting memory leaks.

Table 2. State predicates for acyclic unshared lists. We use `x` and `y` to stand for any two distinct variables in *PVar*

Predicate	Defining Formula Intended Meaning
NotNull[x]	$x \neq null$ `x` does not point to null
Aliased[x, y]	$x \neq null \wedge y \neq null \wedge x = y$ `x` and `y` both point to the same cell
Next[x, y]	$x \neq null \wedge y \neq null \wedge n(x) = y$ `x.n` and `y` point to the same cell
NextNull[x]	$x \neq null \wedge n(x) = null$ `x.n` is *null*
Reach[x, y]	$x \neq null \wedge y \neq null \wedge n^+(x, y)$ `y` is reachable from `x` by following one or more `n` links

Fig. 2(a) shows a concrete program state that arises before the execution of the statement `t.n=q` at program label `L11` (of Fig. 1), and Fig. 2(b) shows its abstraction using our predicates. The abstraction exactly captures the existence of the three lists, the reachability relations between the variables, and the fact that `t` points to the tail of a list. However, the abstraction loses all information on the length of the lists.

3.3 Using Predicates to Express Properties

Using combinations of the predicates in Table 2, we can express several useful properties:

Intersection We can check whether two variables `x` and `y` reach a common cell or belong to disjoint lists by using the following formula

$$Intersect[x, y] \Longleftrightarrow Aliased[x, y] \vee Reach[x, y] \vee Reach[y, x] \ .$$

This property is useful for program parallelization and for inferring *modifies* clauses, which are useful, e.g., for program verifiers based on assume-guarantee reasoning.

Detecting creation of sharing A shared node is created by a statement `x.n=y` when `y` is not a *root*, i.e., the cell pointed-to by `y` has an incoming `n` link

from a non-garbage cell u. We define the macro

$$\mathit{IsRoot}[\mathtt{y}] \equiv \bigwedge_{z \in PVar \setminus \{y\}} \neg \mathit{Reach}[\mathtt{z},\mathtt{y}]$$

and check whether $\neg \mathit{IsRoot}[\mathtt{y}]$ holds to detect creation of sharing.

Detecting creation of cycles A cycle is created by a statement `x.n=y` when there exists a (possibly empty) path of `n` links from the cell pointed-to by `y` to the cell pointed-to by `x`. We define the macro

$$\mathit{ReachOrAliased}[\mathtt{y},\mathtt{x}] \equiv \mathit{Aliased}[\mathtt{y},\mathtt{x}] \vee \mathit{Reach}[\mathtt{y},\mathtt{x}]$$

and check whether $\mathit{ReachOrAliased}[\mathtt{y},\mathtt{x}]$ holds to detect these situations.

Detecting memory leaks A memory leak can occur for two types of statements — `x=null` and `x.n=null`.

In the first case, a leak is created when `x` is not already null and it is the last reference to a cell. We define the macro

$$\mathit{LastRef}[x] \equiv \mathit{NotNull}[\mathtt{x}] \wedge \mathit{IsRoot}[\mathtt{x}] \wedge \bigwedge_{w \in PVar \setminus \{x\}} \neg \mathit{Aliased}[\mathtt{x},\mathtt{w}]$$

and whether $\mathit{LastRef}[x]$ holds to detect a possible memory leak.

In the second case, a leak is created when `x.n` is not already null and it is the last reference to a cell. Since we assume there is no sharing, the statement does not cause a memory leak if and only if `x.n` is referenced by another program variable. We define the macro

$$\mathit{NextPtByVar}[\mathtt{x}] \equiv \bigvee_{w \in PVar \setminus \{x\}} \mathit{Next}[\mathtt{x},\mathtt{w}]$$

and check whether $\neg \mathit{NextNull}[\mathtt{x}] \wedge \neg \mathit{NextPtByVar}[\mathtt{x}]$ holds to detect a possible memory leak.

Detecting null dereferences To detect null dereferences and interpret program conditions, we define the macro

$$\mathit{IsNull}[\mathtt{x}] \equiv \neg \mathit{NotNull}[\mathtt{x}] \ .$$

3.4 An Abstract Semantics for Program Statements

Table 3 contains the update formulae for all program statements except `x=y.n`. For every predicate that is affected by the statement, the value in the next state (appearing on left side of :=) is computed by a Boolean combination of the predicates in the current state (the right side of :=). To save space, we do not mention the predicates that are not affected by the statement. Notice that the size of the update formulae in Table 3 is constant (at most 5 literals are used) and, therefore, the time to compute an output abstract state (a cube) is linear in the number of predicates. Also, the abstract semantics for these statements produces a single output state from every input state.

Fig. 3 shows the predicate update formulae for the program statement `x=y.n`, which are more complex than the updates for the other statements. The formulae

Table 3. Update formulae for all program statements, except `x=y.n`. We use the convention that w and z stand for any two distinct variables that are different from the variables that appear in the statement (`x` and `y`)

Statement	**Update formulae**
`x=null`	$\mathit{NotNull}[\mathtt{x}]' := 0$
	$\mathit{Aliased}[\mathtt{x},\mathtt{w}]' := 0$
	$\mathit{Aliased}[\mathtt{w},\mathtt{x}]' := 0$
	$\mathit{Next}[\mathtt{x},\mathtt{w}]' := 0$
	$\mathit{Next}[\mathtt{w},\mathtt{x}]' := 0$
	$\mathit{NextNull}[\mathtt{x}]' := 0$
	$\mathit{Reach}[\mathtt{x},\mathtt{w}]' := 0$
	$\mathit{Reach}[\mathtt{w},\mathtt{x}]' := 0$
`x=new List()`	$\mathit{NotNull}[\mathtt{x}]' := 1$
	$\mathit{NextNull}[\mathtt{x}]' := 1$
`x=y`	$\mathit{NotNull}[\mathtt{x}]' := \mathit{NotNull}[\mathtt{y}]$
	$\mathit{Aliased}[\mathtt{x},\mathtt{z}]' := \mathit{Aliased}[\mathtt{y},\mathtt{z}]$
	$\mathit{Aliased}[\mathtt{z},\mathtt{x}]' := \mathit{Aliased}[\mathtt{z},\mathtt{y}]$
	$\mathit{Aliased}[\mathtt{x},\mathtt{y}]' := \mathit{NotNull}[\mathtt{y}]$
	$\mathit{Aliased}[\mathtt{y},\mathtt{x}]' := \mathit{NotNull}[\mathtt{y}]$
	$\mathit{Next}[\mathtt{x},\mathtt{z}]' := \mathit{Next}[\mathtt{y},\mathtt{z}]$
	$\mathit{Next}[\mathtt{z},\mathtt{x}]' := \mathit{Next}[\mathtt{z},\mathtt{y}]$
	$\mathit{NextNull}[\mathtt{x}]' := \mathit{NextNull}[\mathtt{y}]$
	$\mathit{Reach}[\mathtt{x},\mathtt{z}]' := \mathit{Reach}[\mathtt{y},\mathtt{z}]$
	$\mathit{Reach}[\mathtt{z},\mathtt{x}]' := \mathit{Reach}[\mathtt{z},\mathtt{y}]$
`x.n=null`	$\mathit{Next}[\mathtt{w},\mathtt{z}]' := \mathit{Next}[\mathtt{w},\mathtt{z}] \wedge \neg \mathit{Aliased}[\mathtt{x},\mathtt{z}]$
	$\mathit{Next}[\mathtt{x},\mathtt{z}]' := 0$
	$\mathit{NextNull}[\mathtt{z}]' := \mathit{NextNull}[\mathtt{z}] \vee \mathit{Aliased}[\mathtt{z},\mathtt{x}]$
	$\mathit{NextNull}[\mathtt{x}]' := 1$
	$\mathit{Reach}[\mathtt{w},\mathtt{z}]' := \mathit{Reach}[\mathtt{w},\mathtt{z}] \wedge \neg(\mathit{ReachOrAliased}[\mathtt{w},\mathtt{x}] \wedge \mathit{Reach}[\mathtt{x},\mathtt{z}])$
	$\mathit{Reach}[\mathtt{x},\mathtt{z}]' := 0$
`x.n=y`	$\mathit{Next}[\mathtt{w},\mathtt{z}]' := \mathit{Next}[\mathtt{w},\mathtt{z}] \vee \mathit{Aliased}[\mathtt{w},\mathtt{x}] \wedge \mathit{Aliased}[\mathtt{z},\mathtt{y}]$
	$\mathit{Next}[\mathtt{x},\mathtt{z}]' := \mathit{Aliased}[\mathtt{z},\mathtt{y}]$
	$\mathit{Next}[\mathtt{x},\mathtt{y}]' := \mathit{NotNull}[\mathtt{y}]$
	$\mathit{NextNull}[\mathtt{z}]' := \mathit{Aliased}[\mathtt{z},\mathtt{x}] \wedge \neg \mathit{NotNull}[\mathtt{y}] \vee \neg \mathit{Aliased}[\mathtt{z},\mathtt{x}] \wedge \mathit{NextNull}[\mathtt{z}]$
	$\mathit{NextNull}[\mathtt{x}]' := \mathit{IsNull}[\mathtt{y}]$
	$\mathit{Reach}[\mathtt{w},\mathtt{z}]' := \mathit{Reach}[\mathtt{w},\mathtt{z}] \vee \mathit{ReachOrAliased}[\mathtt{w},\mathtt{x}] \wedge \mathit{ReachOrAliased}[\mathtt{y},\mathtt{z}]$
	$\mathit{Reach}[\mathtt{x},\mathtt{z}]' := \mathit{ReachOrAliased}[\mathtt{y},\mathtt{z}]$
	$\mathit{Reach}[\mathtt{w},\mathtt{y}]' := \mathit{NotNull}[\mathtt{y}] \wedge \mathit{ReachOrAliased}[\mathtt{w},\mathtt{x}]$
	$\mathit{Reach}[\mathtt{x},\mathtt{y}]' := \mathit{NotNull}[\mathtt{y}]$
`assert(x==y)`	if $\mathit{Aliased}[\mathtt{x},\mathtt{y}] \vee (\mathit{IsNull}[\mathtt{x}] \wedge \mathit{IsNull}[\mathtt{y}]$ holds nop
	else abort with error message
`assert(x!=y)`	if $\neg(\mathit{Aliased}[\mathtt{x},\mathtt{y}] \vee (\mathit{IsNull}[\mathtt{x}] \wedge \mathit{IsNull}[\mathtt{y}]))$ holds nop
	else abort with error message
`assume(x==y)`	if $\mathit{Aliased}[\mathtt{x},\mathtt{y}] \vee (\mathit{IsNull}[\mathtt{x}] \wedge \mathit{IsNull}[\mathtt{y}]$ holds nop else stop
`assume(x!=y)`	if $\neg(\mathit{Aliased}[\mathtt{x},\mathtt{y}] \vee (\mathit{IsNull}[\mathtt{x}] \wedge \mathit{IsNull}[\mathtt{y}]))$ holds nop else stop

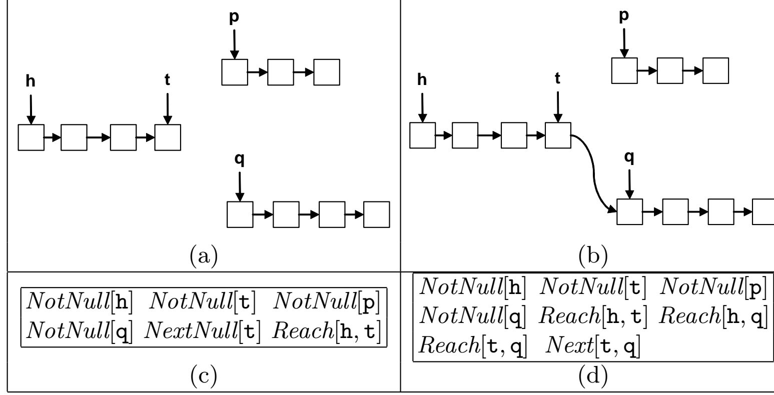

Fig. 2. Concrete and abstract states that appear before the statement `t.n=q` at program label `L11` and after the statement: (a) A concrete state before the statement; (b) The effect of executing `t.n=q` on (a); (c) The predicate abstraction of (a); and (d) The effect of applying the abstract semantics of `t.n=q` to (c). For clarity, the null object is not shown in (a) and (b). Predicates that do not hold are not shown in (c) and (d)

make use of the following macros:

$$\begin{aligned} \mathit{ReachFirst}[\mathtt{x},\mathtt{y}] &\equiv \mathit{Reach}[\mathtt{x},\mathtt{y}] \wedge \textstyle\bigwedge_{z\in PVar\setminus\{x,y\}} \mathit{Reach}[\mathtt{x},\mathtt{z}] \implies \mathit{ReachOrAliased}[\mathtt{y},\mathtt{z}] \\ \mathit{ReachNull}[\mathtt{x}] &\equiv \textstyle\bigwedge_{z\in PVar\setminus\{x\}} \neg\mathit{Reach}[\mathtt{x},\mathtt{z}] \\ \mathit{NextPtByVar}[\mathtt{x}] &\equiv \textstyle\bigvee_{z\in PVar\setminus\{x\}} \mathit{Next}[\mathtt{x},\mathtt{z}] \end{aligned}$$

To simplify the updates, we split the update formulae for the predicates *NextNull*[x] and *Next*[x, z] into 4 cases. The rest of the predicates are updated using the same formulae in all of the cases. The case splitting is achieved by defining condition formulae, which determine the case to enable, and thus which update formulae should be used. Recall that the statement `x=y.n` is preceded by the statement `assert y!=null`, and therefore *NotNull*[y] is assured to hold. Since *NextPtByVar*[y] implies ¬*NextNull*[y], for every input state, exactly one of the condition formulae is enabled.

For the first two cases, a single abstract state is produced. Case 3 and case 4 are more interesting. In these cases two output states are produced due to non-determinism imposed by the list length abstraction. We use *focus* formulae to make the non-determinism evident. These are first-order formulae that cannot be expressed in terms of the predicates we use for the abstraction. Therefore, their value is determined to be 0 or 1 non-deterministically, leading to two output cubes.

In case 3, the update $\mathit{NextNull}[\mathrm{x}]' := \mathit{NextNextNull}[\mathrm{y}]$ produces a cube with the value $\mathit{NextNull}[\mathrm{x}]' = 0$ and a cube with the value $\mathit{NextNull}[\mathrm{x}]' = 1$. The non-determinism arises since a pointer advances on a list whose length is only known to be greater than 1. The value 1 represents the case where the length is exactly 2, and therefore the `n` field of the advanced pointer is *null*, and the

value 0 represents the cases where the length is greater than 2 and therefore the n field of the advanced pointer is not *null*.

In case 4, the update $\mathit{Next}[\mathtt{x},\mathtt{z}]' := \mathit{NextNextPtByVar}[\mathtt{y}] \wedge \mathit{ReachFirst}[\mathtt{y},\mathtt{z}]$ produces an abstract state with the value $\mathit{Next}[\mathtt{x},\mathtt{z}]' := \mathit{ReachFirst}[\mathtt{y},\mathtt{z}]$ and an abstract state with the value $\mathit{Next}[\mathtt{x},\mathtt{z}]' := 0$. The non-determinism arises since a pointer advances into a list that reaches another pointer variable, say z, which is the closest among the variables following y. However, the length of the path between y and z is only known to be greater than 1. The value 0 represents the case where the length is greater than 2, and therefore the n field of the advanced pointer is not pointed-to by any variable, and the value $\mathit{ReachFirst}[\mathtt{y},\mathtt{z}]$ represents the case where the length is exactly 2 and therefore the n field of the advanced pointer is pointed-to by z.

Case 2 and case 4 contain predicate update formulae that are linear in the number of variables (for the predicates of the form $\mathit{Next}[\mathtt{x},\mathtt{z}]$ and $\mathit{NextNull}[\mathtt{x}]$). Therefore, the time to produce an output state from an input state is states is $O(n\sqrt{n})$ where n is the number of predicates (and $n = O(k^2)$ where k is the number of program variables).

The size of the update formulae for all program statements are at most quadratic in the number of variables. This means that the time to produce a new abstract state from an existing abstract state is polynomial in the number of predicates. Therefore, the analysis has polynomial output sensitivity, i.e., the analysis time is polynomial in the number of reachable abstract states.

Fig. 2(d) shows the effect of applying the update formulae for `t.n=q` to the abstract state in Fig. 2(c). The output state precisely captures the reachability between all program variables and the fact that t and q point to adjacent cells.

We claim that the update formulae above provide the best transformers for the respective program statements. In Appendix B, we provide a methodology for deriving optimal abstract transformers for program statements. In particular, we prove that the abstract transformers for all statements except `x=y.n` are *complete*, and that the abstract transformer for `x=y.n` is the best transformer.

4 Summary Content Properties

In this section, we extend our analysis for linked lists to prove a set of more interesting properties, we call *summary content* properties. Intuitively, our goal is to relate the content[5] of the lists, at any control location l_1 in the program, with the content of the lists at the control location *entry* in the program. This is an extension of the memory leak property, where we also check that the content of certain lists only end up in certain other lists. This is particularly useful for proving the relationships of the contents between the lists that are passed as input to a procedure and the lists that are returned from the procedure. For example, one can show that after reversing a list, the content of the output list is exactly the content of the input list.

[5] We use *content* to refer to all the cells reachable from a pointer variable, and not the value inside the cells.

Fig. 3. Update formulae for `x=y.n`. We assume that this statement cannot dereference null, i.e., $\mathit{NotNull}[\mathtt{y}]$ holds. We use the convention that z stands for any variable that is different from `x` and `y`

$\mathit{NotNull}[\mathtt{x}]' := \neg \mathit{NextNull}[\mathtt{y}]$
$\mathit{Aliased}[\mathtt{x},\mathtt{z}]' := \mathit{Next}[\mathtt{y},\mathtt{z}]$
$\mathit{Aliased}[\mathtt{z},\mathtt{x}]' := \mathit{Next}[\mathtt{y},\mathtt{z}]$
$\mathit{Next}[\mathtt{z},\mathtt{x}]' := \neg \mathit{NextNull}[\mathtt{y}] \wedge \mathit{Aliased}[\mathtt{y},\mathtt{z}]$
$\mathit{Next}[\mathtt{y},\mathtt{x}]' := \neg \mathit{NextNull}[\mathtt{y}]$
$\mathit{Reach}[\mathtt{x},\mathtt{z}]' := \neg \mathit{NextNull}[\mathtt{y}] \wedge \mathit{Reach}[\mathtt{y},\mathtt{z}] \wedge \neg \mathit{Next}[\mathtt{y},\mathtt{z}]$
$\mathit{Reach}[\mathtt{z},\mathtt{x}]' := \neg \mathit{NextNull}[\mathtt{y}] \wedge \mathit{ReachOrAliased}[\mathtt{z},\mathtt{y}]$
$\mathit{Reach}[\mathtt{y},\mathtt{x}]' := \neg \mathit{NextNull}[\mathtt{y}]$
if $\mathit{NextNull}[\mathtt{y}]$
then // Case 1
  $\mathit{Next}[\mathtt{x},\mathtt{z}]' := \mathit{Next}[\mathtt{x},\mathtt{z}]$
  $\mathit{NextNull}[\mathtt{x}]' := \mathit{NextNull}[\mathtt{x}]$
else if $\neg \mathit{NextNull}[\mathtt{y}] \wedge \mathit{NextPtByVar}[\mathtt{y}]$
then // Case 2
  $\mathit{Next}[\mathtt{x},\mathtt{z}]' := \bigvee_{w \in PVar \setminus \{x,y,z\}} \mathit{Next}[\mathtt{y},\mathtt{w}] \wedge \mathit{Next}[\mathtt{w},\mathtt{z}]$
  $\mathit{NextNull}[\mathtt{x}]' := \bigvee_{w \in PVar \setminus \{x,y\}} \mathit{Next}[\mathtt{y},\mathtt{w}] \wedge \mathit{NextNull}[\mathtt{w}]$
else if $\neg \mathit{NextNull}[\mathtt{y}] \wedge \mathit{ReachNull}[\mathtt{y}]$
then // Case 3
  $\mathbf{let} \mathit{NextNextNull}[\mathtt{y}] \equiv n(n(y)) = \mathit{null}$
  $\mathit{Focus}(\mathit{NextNextNull}[\mathtt{y}])$
  $\mathit{Next}[\mathtt{x},\mathtt{z}]' := 0$
  $\mathit{NextNull}[\mathtt{x}]' := \mathit{NextNextNull}[\mathtt{y}]$
else if $\neg \mathit{NextNull}[\mathtt{y}] \wedge \neg \mathit{ReachNull}[\mathtt{y}] \wedge \neg \mathit{NextPtByVar}[\mathtt{y}]$
then // Case 4
  $\mathbf{let} \mathit{NextNextPtByVar}[\mathtt{y}] \equiv \bigvee_{t \in PVar \setminus \{y\}} t \neq \mathit{null} \wedge n(n(y)) = t$
  $\mathit{Focus}(\mathit{NextNextPtByVar}[\mathtt{y}])$
  $\mathit{Next}[\mathtt{x},\mathtt{z}]' := \mathit{NextNextPtByVar}[\mathtt{y}] \wedge \mathit{ReachFirst}[\mathtt{y},\mathtt{z}]$
  $\mathit{NextNull}[\mathtt{x}]' := 0$

Before proceeding further, let us modify the concrete semantics to store the set of unallocated cells. We first add a new variable `freeList` of type *List* to model dynamic memory allocation inside a program. At any point in the program, `freeList` points to an acyclic list of all the unallocated cells of type *List* in the heap. Each statement `x = newList()` is translated to (i) `x = freeList`, (ii) `freeList = freeList.n`, and (iii) `x.n = null`.

The *content* of a program variable $\mathtt{x} \in PVar \cup \{\mathtt{freeList}\}$, $\mathit{Reach}(\mathtt{x})$ is defined as the set of cells reachable from `x` following the `n` fields. We define $\mathit{Reach}(\mathtt{x})@\sigma$ to denote the content of `x` in a state σ.

Let Q_0 be the set of initial concrete states at the location *entry* of a program and let us use $\sigma_1 \rightsquigarrow_l \sigma_2$ to denote that the state σ_2 at control location l is reachable from the state σ_1 in some execution. We define a relation $\mathit{flowto}(\mathtt{x})$ for any control location as follows:

Definition 3. *For any control location l, and for any variable* $\mathtt{x} \in PVar \cup \{\mathtt{freeList}\}$*, we define* $flowto(\mathtt{x})$ *at l as* $\{\mathtt{y} \in PVar \mid \exists \sigma_0 \in Q_0, \exists \sigma, \sigma_0 \rightsquigarrow_l \sigma, Reach(\mathtt{x})@\sigma_0 \cap Reach(\mathtt{y})@\sigma \neq \{\}\}$

The definition basically says that $flowto(\mathtt{x})$ at any control location is the set of variables whose reachable set intersects with the content of x in some initial state. In other words, these are the variables that the content of x in Q_0 have flown into. We extend both *Reach*, and $flowto(\cdot)$ to operate on a set of program variables, by simply taking pointwise union over the set.

To capture the $flowto()$ information conservatively, we maintain a predicate $mayflow(\mathtt{x})$ for each variable x in the abstract state. For any program location l, $flowto(\mathtt{x}) \subseteq mayflow(\mathtt{x})$. In other words, $mayflow()$ maintains $flowto()$ information conservatively. In the next section, we provide the updates to this predicate and show that the updates are conservative.

4.1 Updating *mayflow*

In any state $\sigma_0 \in Q_0$, we initialize $mayflow()$ as follows:

$$mayflow(\mathtt{x}) = \{\mathtt{y} \in PVar \mid Intersect[\mathtt{y}, \mathtt{x}]\}$$

The content of x in σ_0 intersects with the content of all the variables t that satisfies $Intersect[\mathtt{t}, \mathtt{x}]$. Notice that the *mayflow* sets can be empty for variables that point to *null*.

Let us look at the different statements:

1. `x = null`: For every $\mathtt{z} \in PVar$,
$$mayflow(\mathtt{z})' = mayflow(\mathtt{z}) \setminus \{\mathtt{x}\}$$
After x is assigned *null*, the content of x (the set $\{\}$) can't intersect with the content of any variable. Hence we remove x from the $mayflow()$ of every variable z.
2. `x = new List()`: Assuming this statement is preceded by `x = null`, we only update $mayflow(\mathtt{freeList})$:
$$mayflow(\mathtt{freeList})' = mayflow(\mathtt{freeList}) \cup \{\mathtt{x}\}$$
After this statement, x holds a cell initially reachable from freeList in σ_0.
3. `x = y`: We know that x is *null* before this statement. For any variable z such that $\mathtt{y} \in mayflow(\mathtt{z})$:
$$mayflow(\mathtt{z})' = mayflow(\mathtt{z}) \cup \{\mathtt{x}\}$$
4. `x = y.n`: We know that x is *null* before this statement. For any variable z such that $\mathtt{y} \in mayflow(\mathtt{z})$:
$$mayflow(\mathtt{z})' = mayflow(\mathtt{z}) \cup \{\mathtt{x}\}$$
5. `x.n = y`: We know that x.n is *null* before this statement. For any variable z such that $\mathtt{y} \in mayflow(\mathtt{z})$:
$$mayflow(\mathtt{z})' = mayflow(\mathtt{z}) \cup \{\mathtt{w} \mid ReachOrAliased[\mathtt{w}, \mathtt{x}]\}$$
After this statement, every variable that reaches x could intersect with the initial content of all z that y (potentially) intersects before the statement.
6. `x.n = null`: In this case, the $mayflow()$ relation does not change.

The updates ensure that *mayflow*() overapproximates the *flowto*() set at each program location.

4.2 Proving Summary Content Properties

We now define a *summary content* property to relate the contents of two sets of variables V_I and V_O in the program as follows:

Definition 4. *For two sets of variables $V_I \subseteq PVar$ and $V_O \subseteq PVar$, we define a* summary content *property $SC(V_I, V_O)$ to denote that for any state $\sigma_0 \in Q_0$, and a state σ at location l such $\sigma_0 \rightsquigarrow_l \sigma$, $Reach(V_O)@\sigma = Reach(V_I)@\sigma_0$.*

The following proposition allows us to prove such summary content properties using the existing predicates augmented with *mayflow*() predicates:

Proposition 1. *For any program manipulating acyclic unshared lists, and two sets of variables $V_I \subseteq PVar$ and $V_O \subseteq PVar$, and any abstract state s_a such that:*

1. *The program has no memory leaks for every possible reachable state,*
2. *For every $\mathtt{y} \in mayflow(V_I)$ in s_a, either $\mathtt{y} \in V_O$ or $\mathtt{y} = null$ in s_a or there exists a $\mathtt{z} \in V_O$ such that $ReachOrAliased[\mathtt{z}, \mathtt{y}]$ in s_a, and*
3. *For every $\mathtt{y} \in V_O$ in s_a, such that $\mathtt{y} \in mayflow(\mathtt{z})$ in s_a but $\mathtt{z} \notin V_I$, either $\mathtt{z} = null$ at $\alpha(Q_0)$ or there exists a variable $\mathtt{x} \in V_I$ and $ReachOrAliased[\mathtt{x}, \mathtt{z}]$ at $\alpha(Q_0)$.*

Then, $SC(V_I, V_O)$ in all the concrete states represented by s_a.

We note that Proposition 1 can be expressed as a Boolean formula over the predicates in a given abstract state. Hence, the property can be checked automatically. The above proposition enables us to relate the contents of the variables in V_O at the control location *exit* in the program with the contents of the variables in V_I at the *entry* to the program, after the abstract analysis terminates.

We also note that the requirement for absence of memory leaks can be lifted and the analysis can be extended to handle memory deallocation statements. However, we do not discuss the details of these extensions here.

4.3 Example

Let us consider the *merge* example from Figure 1. Let us consider the case when both `p` and `q` are not null. The summary content property of interest in this case is $SC(\{\mathtt{p}, \mathtt{q}\}, \{\mathtt{h}\})$ at the location `L17`, i.e., we want to check that the content of the output list (pointed to-by `h`) is precisely the union of content of the disjoint lists pointed-to by `p` and `q`.

Since all the variables apart from `p` and `q` are local, they are initialized to *null*. Hence the initial content of all the local variables is $\{\}$, and hence $mayflow(\mathtt{x})$ for any local variable `x` remains $\{\}$ throughout.

The assignments in L2 and L3 results in $\mathit{mayflow}(\mathtt{p}) = \{\mathtt{p}, \mathtt{h}\}$ and $\mathit{mayflow}(\mathtt{q}) = \{\mathtt{q}, \mathtt{h}\}$. After L4, $\mathit{mayflow}(\mathtt{p}) = \{\mathtt{p}, \mathtt{h}, \mathtt{t}\}$ and $\mathit{mayflow}(\mathtt{q}) = \{\mathtt{q}, \mathtt{h}, \mathtt{t}\}$. After executing L7, the mayflow information remains unchanged. Also note that the program obtained after preprocessing the example in Figure 1 assigns *null* to `temp` before L8; hence `temp` does not belong to $\mathit{mayflow}(\mathtt{p})$ and $\mathit{mayflow}(\mathtt{q})$. Hence, after executing L8 and L9, $\mathit{mayflow}(\mathtt{p}) = \{\mathtt{p}, \mathtt{h}, \mathtt{t}, \mathtt{temp}\}$, but does not contain `q`.

Finally, one can verify that at L17, $\mathit{mayflow}(\mathtt{p}) = \{\mathtt{p}, \mathtt{h}, \mathtt{t}, \mathtt{temp}\}$, and $\mathit{mayflow}(\mathtt{q}) = \{\mathtt{q}, \mathtt{h}, \mathtt{t}, \mathtt{temp}\}$. But, we know that at this point $\mathit{ReachOrAliased}[\mathtt{h}, \mathtt{t}]$, and either (i) both `p` and `q` are *null* or (ii) `p` is *null* and $\mathit{ReachOrAliased}[\mathtt{h}, \mathtt{q}]$ or (iii) `q` is *null* and $\mathit{ReachOrAliased}[\mathtt{h}, \mathtt{p}]$. Also, either `temp` is *null* or $\mathit{ReachOrAliased}[\mathtt{h}, \mathtt{temp}]$ holds. Moreover, our analysis also proves that there is no memory leak in the program. Therefore, by Proposition 1, we can conclude that $\mathit{SC}(\{\mathtt{p}, \mathtt{q}\}, \{\mathtt{h}\})$ holds at exit.

5 Experimental Results

We used TVLA to implement our analysis based on the predicates and abstract transformers described in Sec. 3. We applied the analysis to verify various specifications of programs operating on lists, described in Table 4. The table contains, in each column the measures produced by the analysis in this paper (left side) to the measures produced by the analysis in [19] (right side). (We have not run the analysis of [19] on the `append` and `append_nullderef` examples.) For all examples, in addition to checking that lists do not become cyclic or shared, we checked the absence of *null* dereferences and absence of memory leaks.

The `merge` and `bubbleSort` examples create temporary sharing patterns. Our analysis detects these cases and provides the information needed to understand which statements are responsible for creating the sharing. We modified these examples by changing the order in which the `n` links are mutated and thus avoided created temporary sharing. In the `bubbleSort` example, three variables point to three adjacent cells and are used to swap the order of the second and third cells. Avoiding sharing consists of first setting the `n` links of the three cells, and then setting the links to reflect the new order between the cells.

The experiments were conducted using TVLA version 2, running with SUN's JRE 1.5, on a desktop computer with a 3.4 GHZ Intel Pentium Processor with 1 GB RAM. The results of the analysis are shown in Table 4. In all of the examples, the analysis produced no false alarms. For the `search_nullderef` and `append_nullderef` the analysis detected the null dereference violations.

Although the examples we have used are rather small (at most 40 CFG locations and 67 CFG edges), we are encouraged by the low running times of the analysis. It is also interesting that our analysis was able to prove the same properties as the analysis in [19] using less time and space resources and fewer reachable states (indicated by the #Cubes column). Although it is hard to predict whether the analysis will be faster for larger examples from the times taken on the current examples, we believe that the reduction of the number of reachable states (ranging between 1.8 and 4.9) is an indication that the analysis in

Table 4. Time, space and number of reachable states for the analysis of this paper (numbers left to the /) and the analysis of [19] (numbers right to the /)

Benchmark	**Description**	**Time (sec)**	**Space (MB)**	**#Cubes**
`create`	Dynamically allocates a new linked list	0.01 / 0.05	0 / 0	29 / 40
`delete`	Removes a cell from a list	0.11 / 0.41	0.02 / 1.2	159 / 281
`getLast`	Retrieves the last cell in a list	0.01 / 0.22	0 / 0	69 / 120
`append`	Concatenates two lists	0.05 / 0.44	0 / 0.2	78 / 264
`append_nullderef`	Erroneous implementation of append that dereferences a *null* pointer in some cases	0.06 / 0.5	0 / 0.1	78 / 264
`insert`	Inserts a cell into a sorted list	0.08 / 0.34	0 /1.3	95 / 153
`merge`	Merges two sorted lists into a single list	0.16 / 1.61	0.15 / 7.5	263 / 1288
`reverse`	Reverses an acyclic list in-place	0.05 / 0.25	0 / 0	96 / 172
`search`	Searches for an cell with a specified value	0.03 / 0.06	0 / 0	42 / 76
`search_nullderef`	Erroneous implementation of search that dereferences a *null* pointer in some cases	0.05 / 0.16	0 / 0	55 / 102
`bubbleSort`	Bubble sorting a list	0.22 / 2.06	0.8 / 7.0	529 / 1636

this paper has a better hope of scaling to larger programs than the analysis of [19]. For example, the analysis in [19] produced 1636 reachable states for the `bubbleSort` example, while our analysis produced 529 reachable states.

We have proved the summary content properties for all of the example programs by manually examining all the abstract states that reach the exit location. For all these examples, every pointer variable either points to *null* or is reachable from the head of the output lists. Hence, by Proposition 1, we can show that the contents of the inputs is exactly the content of the output lists. For example, for `reverse`, we show that the content of the reversed list is the same as content of the input list. For `delete` example, we have a pointer to hold the deleted element at the end of the procedure; we can show that the content of the input list is the union of the contents of the output list and the deleted cell (which are disjoint). We are currently automating the check of Proposition 1.

6 Related Work

Shape Analysis Our static analysis algorithm is a special case of shape analysis [12] of dynamically allocated data structures. In contrast to most existing shape analysis algorithms [28, 18, 27, 15] which address the problem of estimating the shapes in programs manipulating arbitrary data structures, in this paper, we design a specialized analysis for programs manipulating acyclic unshared lists. The main idea is to trade generality for simplicity and efficiency and to shift some of the interpretation overhead from static analysis time to design time. From the same reason, we limited the class of properties proved by the our analysis to simple content properties. Interestingly, this covers most of the example programs used in shape analysis and our analysis verified many of the interesting properties proved by the above mentioned shape analysis algorithms.

In the future, we plan to generalize our analysis in order to handle more data structures, e.g., doubly-linked lists, and to prove other properties, e.g., sortedness of linked lists [16].

Decision Procedures In a seminal paper in 1969, Rabin showed the monadic second order logic with one function symbol is decidable [22]. This allows to modularly verify programs manipulating singly linked lists. Indeed, the Mona system automatically proves partial correctness of programs manipulating unshared acyclic lists and trees assuming programmer specified pre- and post-conditions, and loop invariants expressed in Monadic second order logic [21].

Decision procedures can be also employed to automatically generate the best transformers in abstract interpretation and thus avoid the need for programmer specified loop invariants [24]. In this paper we focused on a specific parametric abstraction and eliminated the need for employing decision procedures at analysis time. For example, the analysis of bubble-sort takes 0.22 seconds by our analysis in contrast to 204 seconds (5, 930 calls to a theorem prover) in [2] which uses a specialized decision procedure. Of course, decision procedures can still be employed at analysis generation time to automate the design of specialized shape analyses.

Shape Analysis via Predicate Abstraction Predicate abstraction was used for shape analysis in [7, 1, 19]. The present paper provides an effective solution to the limited problem of proving content properties of unshared acyclic linked lists.

Regular Model Checking Recently regular model checking was also employed to prove properties of programs manipulating singly linked lists [4, 3]. The cost of their analysis is comparable with the cost of decision procedures and is up to three orders of magnitude slower than our analysis.

Separation Logic Separation logic [11] allows to separate assertions on the heap into claims about disjoints parts the heap. This goes beyond our method which only separates the treatments of disjoint lists. For example, when a part of a list is mutated, our analysis can change the predicates of the whole list. In the future we plan to use the notion of cutpoints [25, 26] in order to handle procedures by localizing the treatment of procedure calls.

References

1. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, volume 3148 of *Lecture Notes in Computer Science*. Springer, January. Available at http://www.cs.tau.ac.il/∼rumster/vmcai05.pdf.
2. J. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. Technical report, Intel Corp., 2005. http://www.cs.ubc.ca/cgi-bin/tr/2005/TR-2005-19.pdf.
3. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *In Proc. 12th Intern. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, April 2005.

4. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *In Proc. 16th Intern. Conf. on Computer Aided Verification (CAV'04)*, LNCS, July 2004.
5. V. T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 115–125. ACM Press, 2003.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
7. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 310–324. Springer-Verlag, 2003.
8. G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. Comput.*, 120(1):101–106, 1995.
9. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *LNCS*, 1254:72–83, 1997.
10. N. Immerman. *Descriptive Complexity*. Springer-Verlag New York, Inc., 1994.
11. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. *ACM SIGPLAN Notices*, 36(3):14–26, March 2001.
12. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
13. S. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. Technical Report MSR-TR-2005-97, Microsoft Research, July 2005.
14. W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
15. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In Sagiv, editor, *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005*, 2005.
16. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Proc. of the Int. Symp. on Software Testing and Analysis*, pages 26–38, 2000.
17. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *Proc. Static Analysis Symp.*, volume 1824 of *LNCS*, pages 280–301. Springer-Verlag, 2000.
18. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symp.*, pages 280–301, 2000.
19. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, volume 3148 of *Lecture Notes in Computer Science*. Springer, January. Available at http://www.cs.tau.ac.il/∼rumster/vmcai05.pdf.
20. Y. Matijasevič. *Hilbert's 10th Problem*. MIT Press, 1993.
21. A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '01*, June 2001. Also in SIGPLAN Notices 36(5) (May 2001).
22. M. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc*, 141(1):1–35, 1969.

23. G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.
24. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Proc. Verification, Model Checking, and Abstract Interpretation*, pages 252–266. Springer-Verlag, 2004.
25. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, 2005.
26. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th International Static Analysis Symposium (SAS)*, 2005.
27. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.
28. E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language*. PhD thesis, Univ. of Calif., Berkeley, CA, 1994.

A Undecidability Results for Singly-linked Lists

In this section we strengthen Chakaravathy's [5] undecidability result for the flow-sensitive may-aliasing problem on programs with singly-linked lists. We start by explaining the setting for the original proof and repeating the original proof. Then, we show how the proof can be modified to show undecidability under more restricted semantics, thus strengthening this undecidability result.

A.1 Undecidability of Flow-Sensitive Analysis with Singly-linked Lists

The problem of single-procedural flow-sensitive may-alias analysis of singly-linked lists is the following. Given the programming language described in Sec. 2, we define the *all paths semantics* to be the semantics that conservatively ignores conditions. That is, the original conditions are replaced by if(..)then...else..., and while(..)... thus making every program path executable. Now, given two variables p and q, the goal is to check if there is some path from the entry node to the exit node in the control flow graph, such that, at the end of executing the statements along the path, p and q point to the same list cell.

Theorem 1 (Theorem 2 of [5]). *Single-procedural flow-sensitive may-alias analysis of singly-linked lists is undecidable.*

Proof. The problem of checking whether a multivariate polynomial has integer roots is known to be undecidable. In this problem, we are given a polynomial $P(x_1, x_2, \ldots, x_n)$ over the variables $x_1, x_2, \ldots, x_n$. A sequence of (positive or negative) integer constants $a_1, a_2, \ldots, a_n$, not all zero, is called an integer root of P if $P(a_1, a_2, \ldots, a_n) = 0$. Given the polynomial, the problem is to check if it has any integer roots. The problem is also known as the Hilberts tenth problem. Building on the work of Davis, Putnam and Robinson, Matijasevič proved the undecidability of the Hilberts tenth problem [20].

We prove the undecidability result via a reduction from the above problem. The polynomial $P(x_1, x_2, x_3) = x_1 + x_1x_2 - x_2x_3$ is used as a running example to

illustrate the reduction. The output program for this example is given in the end. Here we explain the ideas used. The output program starts with the following piece of code:

```
Success = new List();
Failure = new List();
dummy = new List();
D = new List();
D.n = Success;
Zero = temp = new List();
WHILE(..) { temp.n = new List(); temp = temp.n; }
```

The fifth statement makes $D.n$ point to *Success*. We will make sure that the polynomial has integer roots if and only if there is an execution path in which $D.n$ remains pointing to *Success* at the exit statement of the program. This would prove the required undecidability. The next few statements in the above code create a singly linked list with *Zero* as the head. This would simulate the positive integer number line, with the kth node representing integer k.

The next segment of the output program simulates choosing constant values for each variable x_i. The value has a magnitude and a (positive or negative) sign. To choose the magnitude of x_i we use a pointer X_i and traverse the linked list. For our example polynomial, the next segment of code is:

```
X1 = Zero; WHILE(..) {X1 = X1.n; }
X2 = Zero; WHILE(..) {X2 = X2.n; }
X3 = Zero; WHILE(..) {X3 = X3.n; }
```

If a path iterates the first loop a_1 times, X_1 would point to the a_1^{th} node in our linked list. This corresponds to assigning $x_1 = a_1$. In general, let $X_1, X_2, \ldots, X_n$ point to nodes $a_1, a_2, \ldots, a_n$ of the linked list. Then, this simulates choosing these values for the variables.

Recall that we want to check if there is a non-zero integer root. Thus, we need to ensure that at least one variable is assigned a non-zero value. The next segment of output our program, ensures that by using a multiway branch with n branches. The code segment for our example would be:

```
SWITCH(..) {
        CASE: X1 = X1.n;
        CASE: X2 = X2.n;
        CASE: X3 = X3.n;
}
```

As one of the branches must be executed, not all variables can point to node zero of the linked list.

Next we simulate choosing signs for the variables. As each of the n variables can be positive or negative, the signs can be chosen in 2^n possible ways. We use a multiway branch[6] (a "switch" statement) with 2^n branches to do the

[6] The multiway branch can easily be translated into a sequence of if(..)then...else... statements.

simulation. Each branch represents choosing a particular combination of signs for the variables. Consider any one branch with one such fixed combination of signs. Then the sign of any term in the polynomial also gets fixed. The sign of a term is determined by its sign in the input polynomial and the combination fixed by the branch. In our example, consider the branch that fixes x_1, x_3 to be positive and x_2 to be negative. Then, sign of the term $-x_2x_3$ would be positive. We separate the terms of the polynomial into groups of positive and negative terms and get two polynomials P_1 and P_2. Then, $a_1, a_2, \ldots, a_n$ is an integer root of P iff if $P_1(a_1, a_2, \ldots, a_n) = P_2(a_1, a_2, \ldots, a_n)$. In our example, consider a branch that represents choosing x_1, x_3 to be positive and x_2 to be negative. Now, irrespective of the magnitudes, the terms x_1 and $-x_2x_3$ would be positive, whereas the term x_1x_2 would be negative. So $P_1 = x_1 + x_2x_3$ and $P_2 = x_1x_2$.

Before proceeding further, we define a macro[7] used in the remainder of our program. The macro takes two parameters A and B and checks if they point to the same location:

```
ALIAS-CHECK(A,B) :
    temp1 = A.n;
    temp2 = B.n;
    A.n = D;
    B.n = dummy;
    A.n.n = Failure;
    A.n = temp1;
    B.n = temp2
```

Ignore the first two and the last two statements for the moment. Then, if A and B point to different locations then the variable D would point to *Failure*. On the the other hand, if they point to the same location then D would remain pointing to *Success* and only a dummy variable will be made to point to *Failure*. The first two and the last two statements ensure that no other variable is affected by this macro.

Recall that we want to check if the polynomials P_1 and P_2 evaluate to the same value. For this purpose, we use two pointers p_1 and p_2. We first set p_1 and p_2 to point to node *Zero* of the linked list. Then we consider the terms of the polynomial one by one. A term would contribute to P_1 if it is positive and to P_2 if it is negative. The sign of the term is determined by two factors: the sign given to the term in the polynomial and which branch of the SWITCH statement we are dealing with. Suppose the term contributes to P_1. In that case we would move p_1 forward on the linked list. If it contributes to P_2, we would move the pointer p_2. In either case, the number of nodes by which the pointer moves is determined by the chosen values $a_1, a_2, \ldots, a_n$. For example, let us take the term x_1x_2. Suppose we are writing code for the branch that represents choosing x_1 and x_3 to be positive and x_2 to be negative. The term x_1x_2 would be negative. So, we move p_2. We need to move it by $a_1 \times a_2$ number of nodes. We use nested loops to achieve this:

[7] Using macros is not an issue, as they can always be expanded.

```
r1 = Zero;
WHILE(..) {
        r1 = r1.n; r2 = Zero;
        WHILE(..) { r2 = r2.n; p2 = p2.n; }
}
```

The problem with the above code is that the loops may be executed arbitrary number times. But we want the inner loop to run for exactly a_2 iterations and the outer loop for exactly a_1 times. To ensure this, we use the fact that X_1 and X_2 are pointing to a_1 and a_2 and do alias checks. The new program fragment is:

```
r1 = Zero;
WHILE(..) {
        r1 = r1.n;
        r2 = Zero;
        WHILE(..) { r2 = r2.n; p2 = p2.n; }
        ALIAS-CHECK(X2, r2);
}
ALIAS-CHECK(X1, r1);
```

Now either p_2 points to node numbered $a_1 \times a_2$ or D points to *Failure*. Our program will make sure that it never goes back to *Success*. After evaluating all the terms of the polynomial we finally check whether p_1 and p_2 point to the same location, using the macro ALIAS-CHECK. Thus D can point to *Success* at the exit of the program iff the polynomial has integer roots.

For each term of the polynomial, we define a macro. The macro takes a parameter p. Suppose the the value of the term at the chosen constants is v, Then the macro either moves forward p by v nodes on the linked list or makes D point to *Failure*.

```
TERM1(p):
        r1 = Zero;
        WHILE(..) {r1 = r1.n; p = p.n; }
        ALIAS-CHECK(X1, r1);

TERM2(p):
        r1 = Zero;
        WHILE(..) {
                r1 = r1.n;
                r2 = Zero;
                WHILE(..) {r2 = r2.n; p = p.n; }
                ALIAS-CHECK(X2, r2);
        }
        ALIAS-CHECK(X1, r1);

TERM3(p):
        r1 = Zero;
```

```
WHILE(..) {
        r1 = r1.n;
        r3 = Zero;
        WHILE(..) {r3 = r3.n; p = p.n; }
        ALIAS-CHECK(X3, r3);
}
ALIAS-CHECK(X1, r1);
```

Now we are ready to present the code:

```
List D = new List();
List Success = new List();
List Failure = new List();
List dummy = new List();
List X1, X2, X3;
List p1, p2;
List r1, r2, r3, temp;

/* Initialize D */
D = Success;
/* Setup number line */
Zero = temp = new List();
WHILE(..) { temp.n = new List(); temp = temp.n; }

/* Choose values */
X1 = Zero; WHILE(..) {X1 = X1; }
X2 = Zero; WHILE(..) {X2 = X2; }
X3 = Zero; WHILE(..) {X3 = X3; }

/* Make sure not all values are zero */
SWITCH(..) {
        CASE: X1 = X1.n;
        CASE: X2 = X2.n;
        CASE: X3 = X3.n;
}

/* Initialize p1 and p2 */
p1 = Zero;
p2 = Zero;

/*
Choose signs and evaluate the two polynomials.
Each branch chooses a particular combination of
signs. In any branch, we consider all the three
terms. And move p1 if the term is positive and
move p2 if the term is negative Whether a term is
positive or negative is determined by the sign of
```

term in the input polynomial and the combination
of signs represented by the branch.
*/
SWITCH(..) {
    CASE: TERM$_1$(p_1); TERM$_2$(p_1); TERM$_3$(p_2);
    /*$+X_1$,$+X_2$,$+X_3$*/
    CASE: TERM$_1$(p_1); TERM$_2$(p_1); TERM$_3$(p_1);
    /*$+X_1$,$+X_2$,$-X_3$*/
    CASE: TERM$_1$(p_1); TERM$_2$(p_2); TERM$_3$(p_1);
    /*$+X_1$,$-X_2$,$+X_3$*/
    CASE: TERM$_1$(p_1); TERM$_2$(p_2); TERM$_3$(p_2);
    /*$+X_1$,$-X_2$,$-X_3$*/
    CASE: TERM$_1$(p_2); TERM$_2$(p_2); TERM$_3$(p_2);
    /*$-X_1$,$+X_2$,$+X_3$*/
    CASE: TERM$_1$(p_2); TERM$_2$(p_2); TERM$_3$(p_1);
    /*$-X_1$,$+X_2$,$-X_3$*/
    CASE: TERM$_1$(p_2); TERM$_2$(p_1); TERM$_3$(p_1);
    /*$-X_1$,$-X_2$,$+X_3$*/
    CASE: TERM$_1$(p_2); TERM$_2$(p_1); TERM$_3$(p_2);
    /*$-X_1$,$-X_2$,$-X_3$*/
}

/* Finally check if p1 and p2 point to same node
in the number line */
ALIAS-CHECK(p_1, p_2)

The polynomial has non-zero integer roots if and only if there is a execution path in the program such that at the last statement D points to *Success*.

□

A.2 More Undecidability Results for Flow-Sensitive Analysis of Singly-linked Lists

We now show that even if we restrict the program semantics such that the shapes of the lists are very simple, the problem of flow-sensitive may-aliasing remains undecidable.

Theorem 2. *Single-procedural flow-sensitive may-alias analysis of singly-linked lists is undecidable even when all lists are acyclic and unshared.*

Proof. Notice that the proof in Appendix A.1 creates a single unshared acyclic list for the integer line, and 4 more list elements referenced by D, *Success*, *Failure*, and *dummy*. The shape of the heap, after setting up the integer line is shown in Fig. 4. At this stage, the heap is free of cycles and sharing.

In the rest of the code pointers are advanced on the integer line list, which does not create cycles or sharing. The only destructive update to the `n` fields

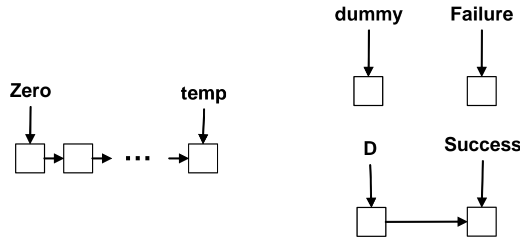

Fig. 4. The shape of the heap after setting up the integer line

happens in ALIAS-CHECK which could actually result in making D.n and *dummy*.n point to *Failure* and thus create sharing.

To see this, consider the situation where ALIAS-CHECK is first applied to aliased pointers. This makes D point to *Failure*. Then, if ALIAS-CHECK is applied to a pair of non-aliased pointers then *dummy* is made to point to *Failure*.

In order to avoid creating this sharing, we slightly alter the ALIAS-CHECK macro:

ALIAS-CHECK(A,B) :
$temp_1 = A.\mathrm{n}$;
$temp_2 = B.\mathrm{n}$;
$A.\mathrm{n} = D$;
$B.\mathrm{n} = dummy$;
$A.\mathrm{n.n} = \mathrm{null}$;
$A.\mathrm{n} = temp_1$;
$B.\mathrm{n} = temp_2$

The change from ALIAS-CHECK is that instead of using the cell referenced by *Failure*, we are setting A.n.n to null. Since it is okay for two list elements to refer to null without creating sharing this solves the problem.

To see that no sharing is created during the application of the revised ALIAS-CHECK and that it operates correctly, we consider the two following cases.

When A and B are not aliased, the configurations shown in Fig. 5 are produced. Note that no matter whether D.n references *Success* before the macro is applied (indicated by the dashed link), after the statement A.n.n = null, the D points to null. The last two statements restore the configuration to the one shown in Fig. 5(a).

When A and B are aliased, the configurations shown in Fig. 6 are produced. Note that the D.n is unchanged. The last two statements restore the configuration to the one shown in Fig. 5(a).

□

We proceed by further restricting the semantics of the programming language. Specifically, we restrict the way pointer variables are allowed to be manipulated.

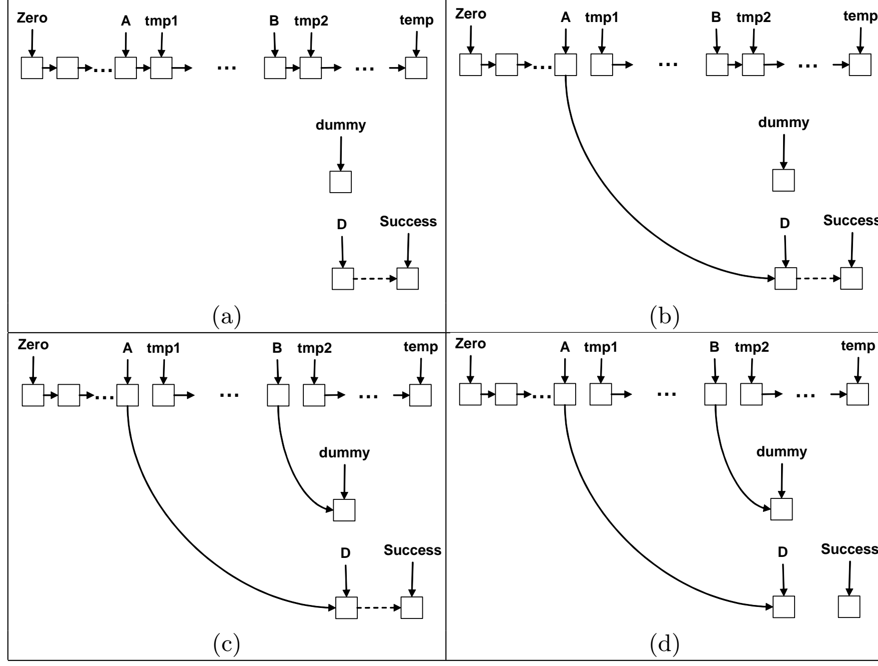

Fig. 5. Configurations produced when ALIAS-CHECK is applied to non-aliased pointers: (a) The configuration after the first two statements, (b) The configuration after $A.\mathrm{n} = D$, (c) The configuration after $B.\mathrm{n} = \mathit{dummy}$, and (d) The configuration after $A.\mathrm{n}.\mathrm{n} = \mathrm{null}$

Definition 5 (List root). *A variable* $\boldsymbol{p}$ *is a* list root, *or* root, *for short, of an acyclic unshared list if it points to the first element of the list. We use the macro* $\mathit{IsRoot}[\boldsymbol{p}]$*, defined below, to denote that* $\boldsymbol{p}$ *is a root.*

$$\mathit{IsRoot}[\boldsymbol{p}] \equiv p \neq \mathit{null} \wedge \forall u : n(u) \neq v \ \ .$$

We now give an inductive definition of an iterator, parameterized by a positive integer k.

Definition 6 (List iterator). *We say that a variable* $\boldsymbol{p}$ *is the first iterator of an acyclic unshared list if it points to an element that is reachable from a root* $\boldsymbol{x}$ *and no element between* $\boldsymbol{x}$ *and* $\boldsymbol{p}$ *is pointed-to by a variable:*

$$\begin{array}{c} \mathit{IsIterator}_1[p] \equiv \exists x \in \mathit{PVar} : \mathit{IsRoot}[\boldsymbol{x}] \wedge n^*(x,p) \wedge x \neq p \wedge \\ \bigwedge_{y \in \mathit{PVar}} n^*(x,p) \implies n^*(x,y) \ \ . \end{array}$$

We say that a variable $\boldsymbol{p}$ *is an iterator of an acyclic unshared list if it is the first iterator or there exists an iterator* $\boldsymbol{y}$ *and the distance between* $\boldsymbol{p}$ *and* $\boldsymbol{y}$ *is not bounded by* k*:*

$$\begin{array}{c} \mathit{IsIterator}_n[p] \equiv \mathit{IsIterator}_1[p] \vee \bigvee_{x,y \in \mathit{PVar}} \mathit{IsRoot}[\boldsymbol{x}] \wedge \mathit{IsIterator}_{n-1}[y] \wedge \\ n^*(x,y) \wedge n^*(y,p) \wedge \bigwedge_{i=1\ldots k} p \neq n^i(y) \ \ . \end{array}$$

Theorem 3. *Single-procedural flow-sensitive may-alias analysis of singly-linked lists is undecidable for acyclic unshared lists with at most two iterators per list.*

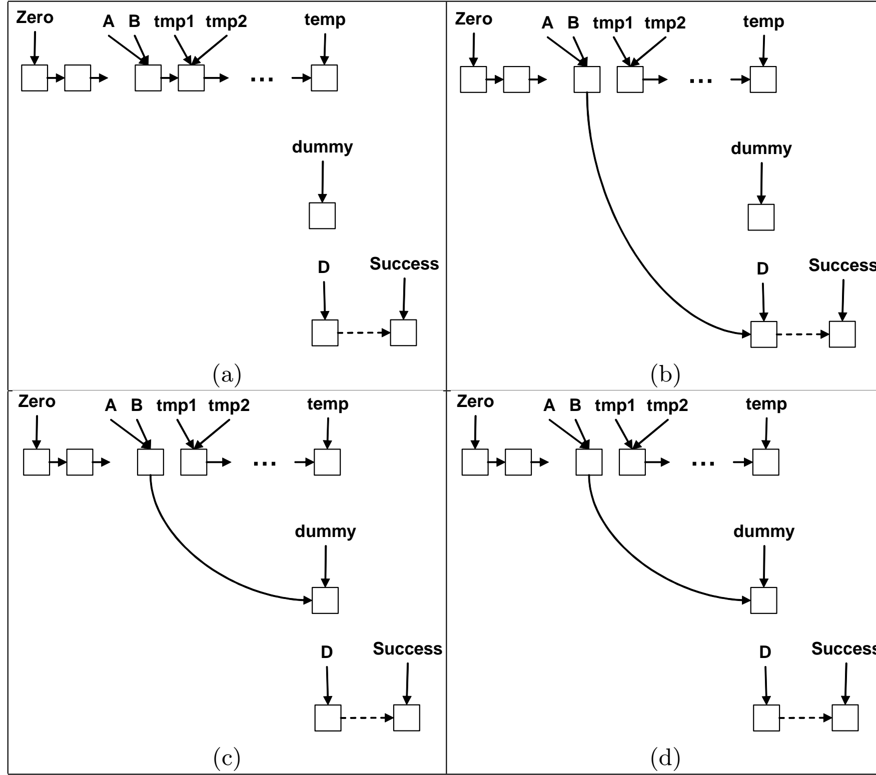

Fig. 6. Configurations produced when ALIAS-CHECK is applied to aliased pointers: (a) The configuration after the first two statements, (b) The configuration after A.n = D, (c) The configuration after B.n = *dummy*, and (d) The configuration after A.n.n = null

Proof. Notice that the undecidability proof in Sec. A.1 creates a single list to simulate the integer line on which several pointer variables are advanced to compute the polynomial terms. This creates configurations where numerous iterators exist in the same list. To show the undecidability proof for the case of two iterators we create numerous integer line lists and advance at most two pointers on every list.

Specifically, let the polynomial terms be indexed by $1, \ldots, m$ and the exponents of a polynomial variable x_i in the term j be indexed by $x_i^{(j,1)}, \ldots, x_i^{(j,n)}$ where n is the exponent of x_i in the term j. We create n integer lines for every polynomial variable x_i in a term where x_i has exponent n. In addition, the output program creates an integer line which is used to equate p_1 and p_2. The list root is $Zero_{p_1,p_2}$ and p_1 and p_2 are the only list iterators on this list. The code fragment below takes care to set the variable *temp* to null when the list is created to avoid increasing the number of iterators to 3.

$Success$ = new List();
$dummy$ = new List();
D = new List();
D.n = $Success$;
$Zero_{p_1,p_2}$ = $temp$ = new List();
WHILE(..) {
    $temp$.n = new List(); $temp$ = $temp$.n;
    /* Create an integer line for every indexed polynomial variable (i, j) */
    ...
    $temp_{(i,j)}$.n = new List(); $temp_{(i,j)}$ = $temp_{(i,j)}$.n;
    ...
}
$temp$ = null
...
$temp_{(i,j)}$ = null
...

We use the statement $temp$.n = new List(), which is a shorthand for $temp_2$ = new List(); $temp$.n = $temp_2$; $temp_2$ = null. (Since the distance from $temp$ to $temp_2$ is exactly 1 then $temp_2$ is not an iterator.)

To choose values for polynomial variables we create a separate integer line for every X_i, as shown in the code below.

$X_1 = Zero_1 =$ new List();
$X_1^{(1,1)} = Zero_1^{(1,1)}$
WHILE(..)
{
X_1.n = new List(); $X_1 = X_1$.n;
$X_1^{(1,1)} = X_1^{(1,1)}$.n
$X_1^{(2,1)} = X_1^{(2,1)}$.n
}
$X_2 = Zero_2 =$ new List();
$X_2^{(2,2)} = Zero_2^{(2,2)} =$ new List();
$X_2^{(3,1)} = Zero_2^{(3,1)} =$ new List();
WHILE(..) {
X_2.n = new List(); $X_2 = X_2$.n;
$X_2^{(2,2)} = X_2^{(2,2)}$.n
$X_2^{(3,1)} = X_2^{(3,1)}$.n
}
$X_3 = Zero_3 =$ new List();
$X_3^{(3,1)} = Zero_3^{(3,1)} = newList()$;
$X_3^{(3,2)} = Zero_3^{(3,2)} = newList()$;
WHILE(..) {
X_3.n = new List(); $X_3 = X_3$.n;
$X_3^{(3,1)} = X_3^{(3,1)}$.n
$X_3^{(3,2)} = X_3^{(3,2)}$.n
}

For a list rooted by $Zero_i$ the variable X_i is an iterator and the program allows only one more pointer to advance on the list. We note that the lengths of the lists for X_i and for the indexed multiplication operations do not have to be of the same lengths. This is because failing to guess long enough lists can only result in null dereferences, which cause the program execution to stop.

To make sure that not all values are zero we use the following code

SWITCH(..) {
CASE: X_1.n = new List(); $X_1 = X_1$.n;
$X_1^{(2,1)}$.n = new List(); $X_1^{(2,1)} = X_1^{(2,1)}$.n;
$X_1^{(1,1)}$.n = new List(); $X_1^{(1,1)} = X_1^{(1,1)}$.n;
CASE: X_2.n = new List(); $X_2 = X_2$.n;
$X_2^{(2,2)}$.n = new List(); $X_2^{(2,2)} = X_2^{(2,2)}$.n;
$X_2^{(3,1)}$.n = new List(); $X_2^{(3,1)} = X_2^{(3,1)}$.n;
CASE: X_3.n = new List(); $X_3 = X_3$.n;
$X_3^{(3,2)}$.n = new List(); $X_3^{(3,2)} = X_3^{(3,2)}$.n;
}

Now, we have a sufficient number of copies of the value of each X_i to simulate all of the multiplications needed to compute the polynomial terms. We define a new version of the TERM macro that was used by the original undecidability proof; one that uses the indexed multiplication lines.

To compute the term $x_1 \times x_2$ we use the following code

```
r_1^(2,1) = Zero_1^(2,1);
WHILE(..) {
    r_1^(2,1) = r_1^(2,1).n;
    r_2^(2,2) = Zero_2^(2,2);
    WHILE(..) { r_2^(2,2) = r_2^(2,2).n; p_2 = p_2.n; }
    ALIAS-CHECK2(X_2^(2,2), r_2^(2,2));
}
ALIAS-CHECK2(X_1^(2,1), r_1^(2,1));
```

Since $X_1^{(2,1)}$ is distant from $Zero_1^{(2,1)}$ as X_1 is from $Zero_1$, then the outer loop advances $r_1^{(2,1)}$ according to the number that the program guessed for the polynomial variable x_1. Similarly for the inner loop and x_2. However, since $r_1^{(2,1)}$ is the only pointer advancing on the list rooted by $Zero_1^{(2,1)}$, there exist only two iterators on the list ($r_1^{(2,1)}$ and $X_1^{(2,1)}$). Similarly, on the list rooted by $Zero_2^{(2,2)}$, there also exist only two iterators on the list ($r_2^{(2,2)}$ and $X_2^{(2,2)}$).

In general, in order to compute a term, every multiplication is carried out in a loop where a variable $r_l^{(i,j)}$ advances on a list rooted by $Zero_l^{(i,j)}$ with the statement ALIAS-CHECK2($X_l^{(i,j)}$, $r_l^{(i,j)}$) to make sure we only count until the value we guessed for the respective variable.

Notice that when ALIAS-CHECK2 is applied the pointers D, *Success*, and *Failure* are at distance at most 2 from the operands of the macro (A and B). To avoid counting them as list iterators, we use $k = 2$ as the constant length in the definition of list iterators.

Choosing the signs for the polynomial variables is done by a multi-way branch exactly as in the original proof.

□

Not surprisingly, the undecidability result holds for properties related to aliasing.

Definition 7. *We say that pointer variables p and q (of type List)* intersect *if they can reach a common element. That is, there exist non-negative integers m and l such that the following formula holds:*

$$u = n^m(p) \wedge v = n^l(q) \wedge u \neq null \wedge v \neq null \wedge u = v \ .$$

We say that p and q are disjoint *if they do not intersect.*

Theorem 4. *Given a single-procedural program with acyclic unshared lists with at most two iterators per list, it is undecidable to check whether two pointers are disjoint.*

Proof. This follows immediately from the proof of Theorem 3, since D.n and *Success* intersect if and only if they are aliased. □

B Deriving Optimal Abstract Transformers for Program Statements

In this section we derive abstract transformers for the different types of program statements. We describe the criteria for which these transformers are *optimal* and supply proofs.

Outline. The rest of this section is organized as follows. In Subsection B.1 we supply definitions and notations used in the rest of this section. In Subsection B.2 we describe a technique that can be used for deriving complete abstract transformers for certain program statements. In Subsection B.3 we describe a technique that can be used for deriving the best abstract transformers for certain program statements. In Subsection B.4 we define a variant of the best transformer and give sufficient conditions under which the the analysis using the variant of the best transformer and the analysis using the best transformer yield the same results. In Subsection B.5 we explain how to use a known result to update reachability predicates. In Subsection B.6 we show how to derive a complete abstract transformer for statements of the form `x=new List()`. In Subsection B.7 we show how to derive a complete abstract transformer for statements of the form `x.n=null`. In Subsection B.8 we show how to derive a complete abstract transformer for statements of the form `x.n=y`. In Subsection B.9 we show how to derive the best abstract transformer for statements of the form `x=y.n`.

The derivation of complete abstract transformers for the statements of the form `x=null` and `x=y` is straightforward, and thus we omit the details.

B.1 Preliminaries

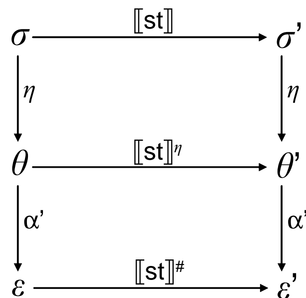

Fig. 7. A pair of commutative diagrams showing the relations between the concrete semantics, the instrumented semantics, and the abstract semantics

We start by providing definitions and notations needed to state and prove the claims about the precision of the abstract transformers. We use Fig. 7 to illustrate the relations between the following terms:

Concrete states The set of symbols $\tau = PVar \cup \{n\}$ is used to define functions over list elements. For convenience of notation we shall also refer to an indexed version of the functions in τ by $f_1, \ldots, f_m$.

A concrete state is a first-order structure $\sigma = (U, I)$ where U is a set of individuals (list elements) and I is a function that interprets symbols in τ. The symbols in $PVar$ are constants, i.e., I maps every symbol to an individual in U (representing either an allocated list element or the null element). The symbol n is interpreted by I as a function $U \to U$.

Concrete semantics We separate the set of statements into two classes: (a) *mutating* statements, which include `x=null`, `x=new List()`, `x=y`, `x.n=null`, `x.n=y`, and `x=y.n`; and *filtering* statements, which include `assume` statements and `assert` statements.

The concrete operational semantics specifies how a mutating statement transforms a concrete state $\sigma = (U, I)$ into a new concrete state $\sigma' = (U', I') = [\![st]\!](\sigma)$. This is done by using a first-order formula ψ_i to relate the interpretation of the symbol $f_i \in \tau$ in I' to the interpretation of the symbol $f_i \in \tau$ in I, for every $i = 1, \ldots, m$. In this paper, the update formulae for the concrete operational semantics are specified in Table 1. We specify the update to the set of elements U directly — U' is either U or $U' = U \cup \{v_{new}\}$ where v_{new} is a fresh element.

The semantics of `assume` statements and `assert` statements is given by state formulae that act as conditions. When the condition holds, the output state is identical to the input state. When the condition fails, the output is the special state $\bot$, which intuitively means that execution has stopped (in the set extension of the semantics we drop bottom states).

The operational semantics is extended to operate on sets of concrete states by using set union to join the effect of the semantics on individual states:

$$[\![st]\!](XS) = \{[\![st]\!](\sigma) \mid \sigma \in XS\} \setminus \{\bot\} \ .$$

Instrumented States We extend the vocabulary of every concrete state to contain a set of *derived* state predicates $P = \{P_1, \ldots, P_n\}$, $\upsilon = \tau \cup P$. The value of every predicate $P_i \in P$ in an instrumented state is defined by a state formula ϕ_i specified in first-order logic with transitive closure.

In this paper, the set of predicates P and the corresponding defining formulae are specified in Table 2.

We denote by η the function that maps a concrete state σ to an instrumented state θ where the interpretation of each predicate $P_i \in P$ is obtained by evaluating its defining formula ϕ_i in σ.

Instrumented semantics The instrumented semantics of a program statement st maps an instrumented state θ to $\theta' = [\![st]\!]^\eta(\theta)$ by using the concrete semantics to update the interpretation of the concrete symbols (in τ), and using first-order formulae over the symbols in υ to update the predicates in P. We denote by μ_i the formula used by the instrumented semantics to update the predicate $P_i \in P$. The updates to the predicates in P yield a commutative diagram (shown in the upper part of Fig. 7). That is, the following equation holds.

$$\eta \circ [\![st]\!] = [\![st]\!]^\eta \circ \eta \ . \qquad (1)$$

Abstract states The abstraction of an instrumented state θ is obtained by projection on the predicates in P, which are therefore referred to as *abstraction predicates*. Formally, $\varepsilon = \alpha'(\theta) = \langle b_1, \ldots, b_n \rangle$ where b_i is a (proposition) Boolean variable and $b_i = [\![P_i]\!]^\theta$ for $i = 1, \ldots, n$. In logical form, the abstract state is the cube $\bigwedge_{P_i \in P} \tilde{P}_i$, where $\tilde{P}_i = P_i$ when $b_i = 1$ and $\tilde{P}_i = \neg P_i$ when $b_i = 0$. The abstraction of a concrete state σ is $\varepsilon = \alpha(\eta(\sigma))$, i.e., $\alpha = \alpha' \circ \eta$. The abstraction of a set of states (concrete states or instrumented states) is the set of abstract states obtained from the abstraction of the individual states.

Note that we do not consider trivial cubes, i.e., cubes that do not represent any concrete state, to be valid abstract states. This is justified by Theorem 8.

We use the subset relation to define a partial order relation $\sqsubseteq$ on sets. This is a standard definition in predicate abstraction.

Abstract semantics In contrast to the concrete semantics and instrumented semantics, where a single input state is mapped to a single output state, the abstract semantics of a program statement *st* is a function from abstract states to sets of abstract states. Given an abstract state ε, the value of a predicate P_i in each output state $\varepsilon' \in [\![st]\!]^\#(\varepsilon)$ is specified by an update formula that is a Boolean function, δ_i, over the variables $\{b_i\}_{i=1}^n$ in ε. (In order to reduce notational overhead we do not introduce the variables $\{b_i\}_{i=1}^n$ but rather use the names of the corresponding predicates.)

The update formulae for the mutating statements `x=null`, `x=new List()`, `x=y`, `x.n=null`, and `x.n=y` are given in Table 3. For these statements, the abstract transformer maps a single abstract state to a single abstract state (a singleton set), resulting with the commutative diagram shown in the lower part of Fig. 7,

$$\alpha' \circ [\![st]\!]^\eta = [\![st]\!]^\# \circ \alpha' \ . \tag{2}$$

The abstract semantics of filtering statements is specified by a Boolean function over the predicates in P, as shown in Table 3. In the sequel, we prove that the abstract semantics is *complete* for these statements.

The abstract semantics for statements of the form `x=y.n` is incomplete, resulting by either 1 or 2 abstract states, as specified in Fig. 3. However, the abstract transformer is the most precise, i.e., the best transformer for this statement.

We extend the abstract semantics of a program statement to operate on sets of abstract states by taking the point-wise extension of the semantics on single abstract states, i.e., the set union of the semantics on each abstract state.

B.2 A Methodology for Deriving Complete Abstract Transformers

We now define what it means for an abstract transformer of a statement to be complete for a given abstraction.

Definition 8 (Completeness of an abstract transformer). *Let st be a program statement. An abstract transformer $[\![st]\!]^\#$ is said to be* complete *when the*

following holds

$$\alpha \circ [\![st]\!] = [\![st]\!]^{\#} \circ \alpha \ .$$

We say that an abstract transformer $[\![st]\!]^{\#}$ is *sound* when $\alpha([\![st]\!](XS)) \sqsubseteq [\![st]\!]^{\#}(\alpha(XS))$ holds for every set of concrete states XS.

Whether a complete abstract transformer exists for a program statement depends on the given abstraction. The following proposition supplies a negative test for completeness, i.e., it gives a way of showing that no abstract transformer is complete for a given program statement and a given abstraction.

Proposition 2. *Let $[\![st]\!]^{\#}$ be a sound abstract transformer for a statement st. The transformer $[\![st]\!]^{\#}$ is complete if and only if every abstract state ε is mapped to a single abstract state.*

Proof. By contradiction. Assume that $[\![st]\!]^{\#}$ is a complete abstract transformer and let ε be an abstract state such that $[\![st]\!]^{\#}(\varepsilon)$ contains two distinct abstract states.

Now, let σ be a concrete state such that $\alpha(\sigma) = \varepsilon$. Then, $\sigma' = [\![st]\!](\sigma)$ is another concrete state (possibly $\bot$). Now, $\alpha(\{\sigma'\}) = \{\varepsilon'\} \setminus \{\bot\}$ where $\varepsilon' = \alpha(\sigma')$ is a single abstract state. We arrived to a contradiction since $|\alpha(\{\sigma'\})| \leq 1$, and thus $[\![st]\!]^{\#}(\alpha(\sigma)) \neq \alpha([\![st]\!](\sigma))$. □

In order to show that no abstract transformer is complete for a given program statement and a given abstraction, we need to show that there exist two concrete states σ_1 and σ_2 such that $\alpha(\sigma_1) = \alpha(\sigma_2)$ and $\alpha([\![st]\!](\sigma_1)) \neq \alpha([\![st]\!](\sigma_2))$.

The following two theorems give sufficient conditions for showing that a complete abstract transformer exists for a given statement and predicate abstraction.

Theorem 5. *Let st be a mutating program statement.*

If for every predicate in P, the update formulae $\{\mu_i\}_{i=1}^{n}$ used by the instrumented semantics are Boolean functions over the current state predicates in P, then the abstract transformer $[\![st]\!]^{\#}$ where $\lambda_i = \mu_i$ for $i = 1, \ldots, n$ is complete.

Proof. Recall that the instrumented semantics, by definition, induces a commutative diagram, as expressed by Equation 1.

Since $\lambda_i = \mu_i$ for $i = 1, \ldots, n$, and the update formulae depend only on the abstraction predicates, Equation 2 ($\alpha' \circ [\![st]\!]^{\eta} = [\![st]\!]^{\#} \circ \alpha'$) holds.

We can now compose the two commutative diagrams, as shown in Fig. 7, and obtain the desired claim:

$$\begin{array}{l} [\![st]\!]^{\#} \circ \alpha = \\ [\![st]\!]^{\#} \circ \alpha' \circ \eta = \\ (\text{Since } \alpha = \alpha' \circ \eta) \\ \alpha' \circ [\![st]\!]^{\eta} \circ \eta = \\ (\text{Using } \mathit{Equation}\ 2) \\ \alpha' \circ \eta \circ [\![st]\!] = \\ (\text{Using } \mathit{Equation}\ 1) \\ \alpha \circ [\![st]\!] \\ (\text{Since } \alpha = \alpha' \circ \eta) \ . \end{array}$$

□

Theorem 6. *Let st be a filtering statement. If the condition can be expressed by a Boolean function over the predicates in P then the corresponding abstract transformer is complete.*

Proof. This immediately follows since the condition formula is expressed by equivalent formulae over the predicates. □

In this paper, the conditions in `assume` and `assert` statements can be expressed by the abstraction predicates (see Table 3). Therefore, by Theorem 6, the abstract transformers are complete for these statements.

For mutating statements we show the completeness of the abstract transformers by using Theorem 5 in the following way. The state $\alpha' \circ [\![st]\!]$ is

$$\langle [\![\phi_1]\!]^{\sigma'}, \ldots, [\![\phi_k]\!]^{\sigma'} \rangle \ .$$

The state $[\![st]\!]^{\eta} \circ \alpha'$ is given by

$$\langle \mu_1(P_1, \ldots, P_n), \ldots, \mu_k(P_1, \ldots, P_n) \rangle \ .$$

To show that the two terms are equal, we show that

$$[\![\phi_i]\!]^{\sigma'} \Longleftrightarrow \mu_i(P_1, \ldots, P_n)$$

holds for every $i = 1, \ldots, k$.

To show this, we use folding/unfolwing transformations:

1. **Unfolding the defining formula of the post-state predicate** We start by writing the formula ϕ_i using the primed versions of the symbols in τ (representing $[\![\phi_i]\!]^{\sigma'}$).
2. **Unfolding the concrete semantics** We use the fact that the set of individuals does not change (this is true for all mutating statements, except `x=new List()`) and substitute the concrete concrete update formulae (from Table 1) with the corresponding symbols from the concrete vocabulary to obtain the formula $\phi_i[\psi_i/P_1, \ldots, \psi_n/P_1]$. This formula contains only current-state (unprimed) symbols.
3. **Equivalence-preserving Transformations** We apply transformations that preserve the equivalence of formulae for our set of models.
4. **Folding** The last transformation results in a Boolean function over the current-state abstraction predicates, which is identical to $\mu_i(b_1, \ldots, b_n)$.

For statements of the form `x=new List()`, the second step cannot be applied directly, since the set of individuals changes. However, most of the state remains unchanged by the allocation statement, which allowing us to apply reasoning specific to this statement to show that the commutativity condition holds.

B.3 A Methodology for Deriving Best Abstract Transformers

We now describe a technique for showing that an incomplete abstract transformer is the most precise. Intuitively, the technique is based on finding formulae that allow to conduct a case analysis. In each case, we use the technique described in the previous subsection to derive a complete transformer. In addition, there is a proof obligation for showing that every abstract state represents concrete states for all cases. We proceed by formally defining the notion of case splitting.

Definition 9 (Focus formulae). *Let P_{focus} be a predicate with the associated defining formula ϕ_{focus} such that for every concrete state σ_1, there exists a concrete state σ_2 such that: (a) $\alpha(\sigma_1) = \alpha(\sigma_2)$; and (b) either $\sigma_1 \models \phi_{focus}$ and $\sigma_2 \models \neg\phi_{focus}$ or $\sigma_2 \models \phi_{focus}$ and $\sigma_1 \models \neg\phi_{focus}$.*

We call P_{focus} a focus predicate *and ϕ_{focus} a* focus formula.

The next theorem supplies sufficient conditions for which a single focus formula can be used to define the best transformer, i.e., there are only two cases. The generalization to more than two cases, where several focus formulae are needed to perform the case splitting, is simple.

Theorem 7. *Let st be a mutating statement.*

Let P_{focus} and ϕ_{focus} be a focus predicate and the associated focus formula, respectively.

Assume that the update formulae δ_i can be expressed by a Boolean function over the predicates in P and P_{focus} for every $i = 1, \ldots, n$ (notice that we do no update P_{focus} itself as it is only needed for the case splitting).

Let μ_i^0 be $\delta_i[0/P_{focus}]$ and let μ_i^1 be $\delta_i[1/P_{focus}]$ for $i = 1, \ldots, n$. Let $[\![st]\!]_0^\#$ be the transformer defined via the update formulae $\{\mu_i^0\}_{i=1}^n$, and let $[\![st]\!]_1^\#$ be the transformer defined via the update formulae $\{\mu_i^1\}_{i=1}^n$.

We define the following abstract transformer. Given an abstract state ε,

$$[\![st]\!]^\#(\varepsilon) = \{[\![st]\!]_0^\#(\varepsilon), [\![st]\!]_1^\#(\varepsilon)\} \ .$$

That is, the output consists of two cubes where the update formulae are the ones used by the instrumented semantics and the value of P_{focus} is set nondeterministically to 0 and 1.

The abstract transformer $[\![st]\!]^\#$ defined above is the best transformer for st.

Proof. Let $[\![st]\!]^b$ be the best transformer for the mutating statement st[8] and let $[\![st]\!]^\#$ be the abstract transformer defined above. We have to show two things: (i) the abstract transformer $[\![st]\!]^\#$ is sound, i.e., $\alpha([\![st]\!](\sigma)) \sqsubseteq [\![st]\!]^\#(\alpha(\sigma))$ holds for every concrete state σ; and (ii) the transformer is as precise as the best, i.e., $[\![st]\!]^\#(\alpha(\sigma)) \sqsubseteq [\![st]\!]^b(\alpha(\sigma))$ holds for every concrete state σ. (These two properties directly extend to sets of concrete states.)

Let σ be a concrete state such $\sigma \models \phi_{focus}$. The instrumented state $\theta = \eta(\sigma)$ also satisfies ϕ_{focus}. Therefore, $[\![\mu_i^1]\!]^\theta = [\![\delta_i]\!]^\theta$ for $i = 1, \ldots, n$, and thus the following equation holds

$$\alpha'([\![st]\!]^\eta(\theta)) = [\![st]\!]_1^\#(\alpha'(\theta)) \ .$$

Now, notice that, for every concrete state that satisfies ϕ_{focus}, we have two commutative diagrams, as shown in Fig. 7, which means that the transformer $[\![st]\!]_1^\#$ is complete for this set of states. (This is the same reasoning used in the proof of Theorem 5). Similarly, the transformer $[\![st]\!]_0^\#$ is complete for the set of states that satisfy $\neg\phi_{focus}$.

Since every concrete state satisfies either ϕ_{focus} or $\neg\phi_{focus}$, we have that $\alpha([\![st]\!](\sigma)) \in [\![st]\!]^\#(\alpha(\sigma))$ holds for every concrete state. Thus, $\alpha([\![st]\!](\sigma)) \sqsubseteq [\![st]\!]^\#(\alpha(\sigma))$, which shows that the abstract transformer $[\![st]\!]^\#$ is sound.

[8] The best abstract transformer is known to exist for every predicate abstraction.

Assume that $\sigma \models \phi_{focus}$ (the proof for the case where $\sigma \models \neg\phi_{focus}$ is symmetric). Recall that $[\![st]\!]^{\#}(\alpha(\sigma)) = \{[\![st]\!]_0^{\#}(\alpha(\sigma)), [\![st]\!]_1^{\#}(\alpha(\sigma))\}$. To show that $[\![st]\!]^{\#}(\alpha(\sigma)) \sqsubseteq [\![st]\!]^{b}(\alpha(\sigma))$ holds we have to show that both $[\![st]\!]_0^{\#}(\alpha(\sigma)) \in [\![st]\!]^{b}(\alpha(\sigma))$ and $[\![st]\!]_1^{\#}(\alpha(\sigma)) \in [\![st]\!]^{b}(\alpha(\sigma))$ hold.

Now, since $[\![st]\!]_1^{\#}(\alpha(\sigma)) = \alpha([\![st]\!](\sigma))$ holds and the best transformer is sound, we have that $[\![st]\!]_1^{\#}(\alpha(\sigma)) \in [\![st]\!]^{b}(\alpha(\sigma))$ holds.

Since ϕ_{focus} is a focus formula, there exists a state σ' such that $\sigma' \models \neg\phi_{focus}$ and $\alpha(\sigma) = \alpha(\sigma')$. From the fact $\alpha(\sigma) = \alpha(\sigma')$, we have that the following equation holds

$$[\![st]\!]^{b}(\alpha(\sigma)) = [\![st]\!]^{b}(\alpha(\sigma')) \ . \tag{3}$$

From the fact $\sigma' \models \neg\phi_{focus}$, we have that the following equation holds

$$\alpha([\![st]\!](\sigma')) = [\![st]\!]_0^{\#}(\alpha(\sigma')) \ . \tag{4}$$

Since $[\![st]\!]^{b}$ is a sound abstract transformer we have that the following holds

$$\alpha([\![st]\!](\sigma')) \in [\![st]\!]^{b}(\alpha(\sigma')) \ . \tag{5}$$

Combining Equation 5 and Equation 3 we obtain

$$\alpha([\![st]\!](\sigma')) \in [\![st]\!]^{b}(\alpha(\sigma)) \ . \tag{6}$$

Combining Equation 6 and Equation 4 we obtain

$$[\![st]\!]_0^{\#}(\alpha(\sigma')) \in [\![st]\!]^{b}(\alpha(\sigma)) \ . \tag{7}$$

Now, since $[\![st]\!]_0^{\#}(\alpha(\sigma')) = [\![st]\!]_0^{\#}(\alpha(\sigma))$, we can rewrite Equation 7 and obtain

$$[\![st]\!]_0^{\#}(\alpha(\sigma)) \in [\![st]\!]^{b}(\alpha(\sigma)) \ . \tag{8}$$

□

B.4 A Variant of the Best Transformer

We now define a variant of the best transformer and show that, under reasonable assumptions, the set of abstractly reachable states is the same as the set of states that are abstractly reachable with the best transformer.

Let $\widetilde{[\![st]\!]}^{\#}$ be an abstract transformer, such that for every cube a, if $\gamma(a) \neq \emptyset$ then $\widetilde{[\![st]\!]}^{\#}(a) = [\![st]\!]^{\#}(a)$. That is, $\widetilde{[\![st]\!]}^{\#}$ identifies with $[\![st]\!]^{\#}$ on input cubes that represent at least one concrete state.

Theorem 8. *Let S_a be a set of cubes such that $\forall a \in S_a : \gamma(a) \neq \emptyset$. Then, the set of abstract states that are reachable from S_a by the transition relation induced by $[\![st]\!]^{\#}$ is the set of states reachable by the transition relation induced by $\widetilde{[\![st]\!]}^{\#}$.*

In other words, for every sequence of statements $st_1, \ldots, st_k$, the following holds:

$$[\![st_k]\!]^{\#} \circ \ldots \circ [\![st_1]\!]^{\#}(s_a) = \widetilde{[\![st_k]\!]}^{\#} \circ \ldots \circ \widetilde{[\![st_1]\!]}^{\#}(s_a) \ .$$

Proof. We know that the claim holds for a single statement. It suffices to prove that if a is a cube a such that $a \neq \emptyset$ then $\gamma(b) \neq \emptyset$ for every cube $b \in [\![st]\!]^{\#}(a)$. The rest of the proof is by induction on the length the sequence of statements.

Let a be a cube such that $\gamma(a) \neq \emptyset$. Let $[\![st]\!](\gamma(a))$ be the set of concrete states that result from application of the concrete transformer to the concrete states

represented by a. If $[\![st]\!](\gamma(a))$ is the empty set, then $[\![st]\!]^{\#}(a) = \{\alpha(t_a) \mid t_a \in [\![st]\!](\gamma(a))\} = \emptyset$, and the claim holds vacuously (there are no output cubes). If $[\![st]\!](\gamma(a))$ is not empty then $[\![st]\!]^{\#}(a) = \{\alpha(t_a) \mid t_a \in [\![st]\!](\gamma(a))\}$ is the set of cubes mapped by α from some concrete state in $[\![st]\!](\gamma(a))$. Therefore, every cube represents at least the concrete states mapped to it and therefore applying γ to the cube returns a non-empty set, and thus the claim holds for all output cubes. □

Thus, under the assumption that the set of input cubes does not contain trivial cubes (cubes that do not represent any concrete states), our transformers identify with the best transformers. Since the logic we use is decidable, this assumption can also be enforced, i.e., we can check whether a cube is non-trivial. For the benchmarks we used, the input cubes were produced from combinations of lists pointed-to by single variables:

$$NotNull[\mathtt{x}] \wedge NextNull[\mathtt{x}], NotNull[\mathtt{x}] \wedge \neg NextNull[\mathtt{x}] \enspace ,$$

and thus the cubes are non-trivial.

B.5 Using the Dong and Su Reachability Update Formulae

In the construction of the update formulae for `x.n=null` and `x.n=y`, we use the fact that *REACH(acyclic)* $\in$ *DynFO* [10] (Theorem 14.20, pages 230–231), originally shown by Dong and Su [8]. This result lets us update the path predicate P, which is the reflexive-transitive closure of an acyclic relation E, using the following first-order formulae:

Edge insertion When an edge (a, b) is added to the graph, the update to the path predicate is given by

$$P'(x, y) := P(x, y) \vee (P(x, a) \wedge P(b, y)) \enspace . \tag{9}$$

Edge deletion When an edge (a, b) is deleted from the graph, the update to the path predicate is given by

$$\begin{aligned} P'(x, y) := P(x, y) \wedge \big[&\neg(P(x, a) \wedge P(b, y)) \vee \\ &(\exists u\, v)\big(P(x.u) \wedge E(u, v) \wedge P(v, y) \\ &\wedge P(u, a) \wedge \neg P(v, a) \wedge (a \neq u \vee b \neq v)\big)\big] \enspace . \end{aligned} \tag{10}$$

We use these formulae to obtain the following two lemmas.

Lemma 1. *For every four list elements x, y, w, and z, such that $x \neq null$, $z \neq null$, $w \neq null$, $x \neq y$, and $n(x) = null$ hold , the following holds.*

$$(\lambda\, v\, .\, (v = x ? y : n(v)))^+(w, z) \Longleftrightarrow n^+(w, z) \vee n^*(w, x) \wedge n^*(y, z) \tag{11}$$

Proof. First, if y is *null* we have

$$(\lambda\, v\, .\, (v = x ? y : n(v)))^+(w, z) \Longleftrightarrow (\lambda\, v\, .\, (v = x ? null : n(v)))^+(w, z)$$

But, since $n(x) = null$ we have

$$(\lambda\, v\, .\, (v = x ? null : n(v)))^+(w, z) \Longleftrightarrow n^+(w, z) \enspace .$$

On the other hand,

$$n^+(w, z) \vee n^*(w, x) \wedge n^*(y, z) \Longleftrightarrow n^+(w, z) \vee n^*(w, x) \wedge n^*(null, z) \enspace .$$

Since $z \neq \mathit{null}$, we have $n^*(\mathit{null}, z) = 0$ and thus

$$n^+(w,z) \vee n^*(w,x) \wedge n^*(\mathit{null}, z) \Longleftrightarrow n^+(w,z) \enspace .$$

Therefore, the equivalence holds in the case of $y = \mathit{null}$.

We now proceed to prove the equivalence in the case of $y \neq \mathit{null}$.

Since the heaps we consider contain only acyclic lists, we can write

$$(\lambda\, v\, .\, (v = x?y : n(v)))^+(w,z) \Longleftrightarrow (\lambda\, v\, .\, (v = x?y : n(v)))^*(w,z) \wedge w \neq z$$

We can now use Equation 9 and write

$$\begin{gathered}(\lambda\, v\, .\, (v = x?y : n(v))^*(w,z) \wedge w \neq z \Longleftrightarrow \\ (n^*(w,z) \vee (n^*(w,x) \wedge n^*(y,z))) \wedge w \neq z \enspace .\end{gathered}$$

Opening parenthesis, we get

$$\begin{gathered}(n^*(w,z) \vee (n^*(w,x) \wedge n^*(y,z))) \wedge w \neq z \Longleftrightarrow \\ (n^*(w,z) \wedge w \neq z) \vee (n^*(w,x) \wedge n^*(y,z) \wedge w \neq z) \enspace .\end{gathered}$$

Now, $n^*(w,z) \wedge w \neq z \Longleftrightarrow n^+(w,z)$, and since heaps are acyclic and $x \neq y$, we also have that $n^*(w,x) \wedge n^*(y,z) \implies w \neq z$, and therefore we can write

$$\begin{gathered}(n^*(w,z) \wedge w \neq z) \vee (n^*(w,x) \wedge n^*(y,z) \wedge w \neq z) \Longleftrightarrow \\ n^+(w,z) \vee n^*(w,x) \wedge n^*(y,z) \enspace .\end{gathered}$$

□

Lemma 2. *For list elements x, w, and z, such that $x \neq \mathit{null}$, $z \neq \mathit{null}$, and $w \neq \mathit{null}$ holds, the following holds.*

$$(\lambda\, v\, .\, (v = x?\mathit{null} : n(v)))^+(w,z) \Longleftrightarrow n^+(w,z) \wedge \neg(n^*(w,x) \wedge n^+(x,z)) \quad (12)$$

Proof. Since the heaps we consider are deterministic, we can simplify Equation 10 to

$$P'(x,y) := P(x,y) \wedge \neg(P(x,a) \wedge P(b,y)) \enspace .$$

Now, since we assumed all lists are acyclic,

$$(\lambda\, v\, .\, (v = x?\mathit{null} : n(v)))^+(w,z) \Longleftrightarrow (\lambda\, v\, .\, (v = x?\mathit{null} : n(v)))^*(w,z) \wedge w \neq z \enspace .$$

Using the simplified version of Equation 10, where the deleted edge is $(x, n(x))$, we write

$$\begin{gathered}(\lambda\, v\, .\, (v = x?\mathit{null} : n(v)))^*(w,z) \wedge w \neq z \Longleftrightarrow \\ n^*(w,z) \wedge \neg(n^*(w,x) \wedge n^*(n(x),z)) \wedge w \neq z \enspace .\end{gathered}$$

We use the assumption that all lists are acyclic and thus $n^*(w,z) \wedge w \neq z \Longleftrightarrow n^+(w,z)$, and that $n^*(n(x),z) \Longleftrightarrow n^+(x,z)$, and write

$$\begin{gathered}n^*(w,z) \wedge \neg(n^*(w,x) \wedge n^*(n(x),z)) \wedge w \neq z \Longleftrightarrow \\ n^+(w,z) \wedge \neg(n^*(w,x) \wedge n^+(x,z)) \enspace .\end{gathered}$$

□

B.6 Deriving Update Formulae for `x=new List()`

We assume that this statement is preceded by the statements `x=null`.

The interpretation of the symbols in $\mathit{PVar} \setminus \{x\}$ is unaffected by the statement. Therefore, the predicates of the form *Aliased*[`w`, `z`] and *NotNull*[`z`] retain their current values for every $z \in \mathit{PVar} \setminus \{x\}$.

Since v_{new} is a fresh cell, the links between other cells remain unaffected by the statement. Therefore, Therefore, the predicates of the form $Next[\texttt{w},\texttt{z}]$ and $Reach[\texttt{w},\texttt{z}]$ retain their current values for every $z \in PVar \setminus \{x\}$.

We shall use the fact that, following the application of the statement, for every element $u \neq v_{new}$, $\neg n'^{+}(u, v_{new})$ and $\neg n'^{+}(v_{new}, u)$ hold.

Assume that $z \in PVar \setminus \{x\}$.

$$\begin{aligned}
NotNull[\texttt{x}]' &\iff x' \neq null \\
&\qquad \text{(Unfolding the defining formula of } NotNull[\texttt{x}]'\text{)} \\
&\iff v_{new} \neq null \\
&\qquad \text{(Unfolding the concrete semantics of } \texttt{x=y.n}\text{)} \\
&\iff 1 \\
&\qquad \text{(Since we assumed that } v_{new} \text{ is fresh)}
\end{aligned}$$

$$\begin{aligned}
Next[\texttt{x},\texttt{z}]' &\iff x' \neq null \wedge z' \neq null \wedge n'(x') = z' \\
&\qquad \text{(Unfolding the defining formula of } Next[\texttt{x},\texttt{z}]'\text{)} \\
&\iff v_{new} \neq null \wedge z \neq null \wedge (\lambda\, v\, .\, (v = v_{new}\, ?\, null\, :\, n(v)))(v_{new}) = z \\
&\qquad \text{(Unfolding the concrete semantics of } \texttt{x=new List()}\text{)} \\
&\iff 1 \wedge z \neq null \wedge null = z) \\
&\qquad (\lambda \text{ application}) \\
&\iff 0 \\
&\qquad (z \neq null \text{ and } null = z \text{ Contradict each other})
\end{aligned}$$

$$\begin{aligned}
Next[\texttt{z},\texttt{x}]' &\iff z' \neq null \wedge x' \neq null \wedge n'(z') = x' \\
&\qquad \text{(Unfolding the defining formula of } Next[\texttt{z},\texttt{x}]'\text{)} \\
&\iff z' \neq null \wedge v_{new} \neq null \wedge (\lambda\, v\, .\, (v = v_{new}\, ?\, null\, :\, n(v)))(z) = v_{new} \\
&\qquad \text{(Unfolding the concrete semantics of } \texttt{x=new List()}\text{)} \\
&\iff z \neq null \wedge 1 \wedge n(z) = v_{new}) \\
&\qquad (\lambda \text{ application}) \\
&\iff 0 \\
&\qquad \text{(Since } n(z) \text{ is an element in the pre-state and } v_{new} \text{ is not)}
\end{aligned}$$

$$\begin{aligned}
NextNull[\texttt{x}]' &\iff x' \neq null \wedge n'(x') = null \\
&\qquad \text{(Unfolding the defining formula of } NextNull[\texttt{x}]'\text{)} \\
&\iff v_{new} \neq null \wedge \lambda\, v\, .\, (v = v_{new}\, ?\, null\, :\, n(v))(v_{new}) = null \\
&\qquad \text{(Unfolding the concrete semantics of } \texttt{x=new List()}\text{)} \\
&\iff 1 \wedge null = null) \\
&\qquad (\lambda \text{ application}) \\
&\iff 1
\end{aligned}$$

$$
\begin{aligned}
\mathit{Reach}[\mathtt{x},\mathtt{z}]' &\iff x' \neq \mathit{null} \wedge z' \neq \mathit{null} \wedge {n'}^{+}(x', z') \\
&\quad\text{(Unfolding the defining formula of } \mathit{Reach}[\mathtt{x},\mathtt{z}]'\text{)} \\
&\iff v_{new} \neq \mathit{null} \wedge z \neq \mathit{null} \wedge {n'}^{+}(v_{new}, z) \\
&\quad\text{(Unfolding the concrete semantics of } \mathtt{x=new\ List()}\text{)} \\
&\iff 0 \\
&\quad\text{(Since } n'(v_{new}) = \mathit{null} \text{ and } z \neq \mathit{null}\text{)}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Reach}[\mathtt{z},\mathtt{x}]' &\iff z' \neq \mathit{null} \wedge x' \neq \mathit{null} \wedge {n'}^{+}(z', x') \\
&\quad\text{(Unfolding the defining formula of } \mathit{Reach}[\mathtt{z},\mathtt{x}]'\text{)} \\
&\iff z \neq \mathit{null} \wedge v_{new} \neq \mathit{null} \wedge {n'}^{+}(z, v_{new}) \\
&\quad\text{(Unfolding the concrete semantics of } \mathtt{x=new\ List()}\text{)} \\
&\iff 0 \\
&\quad\text{(Since } {n'}^{+}(z, v_{new}) \text{ does not hold for any element)}
\end{aligned}
$$

B.7 Deriving Update Formulae for `x.n=null`

We assume the this statement is preceded by `assert x != null`, and therefore $x \neq \mathit{null}$ holds.

Since the concrete semantics only changes the function n, the predicates of the form $\mathit{Aliased}[\mathtt{w},\mathtt{z}]$ and $\mathit{NotNull}[\mathtt{z}]$ retain their current values.

Assume that $w, z \in \mathit{PVar} \setminus \{x\}$ and that w and z are different ($\mathit{Next}[\mathtt{w},\mathtt{z}]$ is defined for different variables).

$$
\begin{aligned}
\mathit{Next}[\mathtt{w},\mathtt{z}]' &\iff w' \neq \mathit{null} \wedge z' \neq \mathit{null} \wedge n'(w') = z' \\
&\quad\text{(Unfolding the defining formula of } \mathit{Next}[\mathtt{w},\mathtt{z}]'\text{)} \\
&\iff w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (\lambda\, v.v = x ? \mathit{null} : n(v))(w) = z \\
&\quad\text{(Unfolding the concrete semantics of } \mathtt{x.n=null}\text{)} \\
&\iff w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (w = x \wedge z = \mathit{null} \vee w \neq x \wedge n(w) = z) \\
&\quad\text{(}\lambda\text{ application)} \\
&\iff w \neq \mathit{null} \wedge (z \neq \mathit{null} \wedge w \neq x \wedge n(w) = z) \\
&\quad\text{(Distributing } \wedge \text{ over } \vee\text{)} \\
&\iff w \neq x \wedge w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n(w) = z \\
&\quad\text{(Opening parenthesis and reordering terms)} \\
&\iff \neg \mathit{Aliased}[\mathtt{w},\mathtt{x}] \wedge \mathit{Next}[\mathtt{w},\mathtt{z}] \\
&\quad\text{(Folding)}
\end{aligned}
$$

When w is x, the step before folding yields 0, and therefore $\mathit{Next}[\mathtt{x},\mathtt{z}]' = 0$.
When z is x, we continue from the last step before folding

$$
\begin{aligned}
\mathit{Next}[\mathtt{w},\mathtt{x}]' &\iff w \neq \mathit{null} \wedge w \neq x \wedge x \neq \mathit{null} \wedge n(w) = x \\
&\iff w \neq \mathit{null} \wedge x \neq \mathit{null} \wedge n(w) = x \\
&\quad\text{(from Def. 1 we have that } n(w) = x \implies w \neq x\text{)} \\
&\iff \mathit{Next}[\mathtt{w},\mathtt{x}] \\
&\quad\text{(Folding)}
\end{aligned}
$$

Assume that $z \in PVar \setminus \{x\}$.

$$
\begin{aligned}
\mathit{NextNull}[\mathtt{z}]' \iff & z' \neq \mathit{null} \wedge n'(z') = \mathit{null} \\
& \text{(Unfolding the defining formula of } \mathit{NextNull}[\mathtt{z}]'\text{)} \\
\iff & z \neq \mathit{null} \wedge (\lambda\, v.v = x ? \mathit{null} : n(v))(z) = \mathit{null} \\
& \text{(Unfolding the concrete semantics of } \mathtt{x.n{=}null}\text{)} \\
\iff & z \neq \mathit{null} \wedge (z = x \wedge \mathit{null} = \mathit{null} \vee z \neq x \wedge n(z) = \mathit{null}) \\
& (\lambda \text{ application}) \\
\iff & z \neq \mathit{null} \wedge z = x \vee z \neq \mathit{null} \wedge z \neq x \wedge n(z) = \mathit{null} \\
& \text{(Opening parenthesis)} \\
\iff & z = x \vee z \neq x \wedge z \neq \mathit{null} \wedge n(z) = \mathit{null} \\
& (z = x \text{ and we assumed that } x \neq \mathit{null}) \\
\iff & z = x \vee z \neq \mathit{null} \wedge n(z) = \mathit{null} \\
& (A \vee \neg A \wedge B \;\leftrightarrow\; A \vee B) \\
\iff & \mathit{Aliased}[\mathtt{z},\mathtt{x}] \vee \mathit{NextNull}[\mathtt{z}] \\
& \text{(Folding)}
\end{aligned}
$$

When z is x the step before folding yields 1, and therefore $\mathit{NextNull}[\mathtt{x}]' = 1$.

Assume that $w, z \in PVar$ and that w and z are different ($\mathit{Reach}[\mathtt{w},\mathtt{z}]$ is defined for different variables).

$$
\begin{aligned}
\mathit{Reach}[\mathtt{w},\mathtt{z}]' \iff & w' \neq \mathit{null} \wedge z' \neq \mathit{null} \wedge n'^{+}(w', z') \\
& \text{(Unfolding the defining formula of } \mathit{Reach}[\mathtt{w},\mathtt{z}]'\text{)} \\
\iff & w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (\lambda\, v.v = x ? \mathit{null} : n(v))^{+}(w, z) \\
& \text{(Unfolding the concrete semantics of } \mathtt{x.n{=}null}\text{)} \\
\iff & w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n^{+}(w, z) \wedge \neg(n^{*}(w, x) \wedge n^{+}(x, z)) \\
& \text{(Lemma 2)} \\
\iff & \mathit{Reach}[\mathtt{w},\mathtt{z}] \wedge \neg(\mathit{ReachOrAliased}[\mathtt{w},\mathtt{x}] \wedge \mathit{Reach}[\mathtt{x},\mathtt{z}]) \\
& (x \neq \mathit{null} \text{ and folding})
\end{aligned}
$$

When w is x, we get

$$
\begin{aligned}
\mathit{Reach}[\mathtt{x},\mathtt{z}]' \iff & x \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n^{+}(x, z) \wedge \neg(n^{*}(x, x) \wedge n^{+}(x, z)) \\
& \text{(Lemma 2)} \\
\iff & x \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n^{+}(x, z) \wedge \neg n^{+}(x, z) \\
& (n^{*}(x, x) \text{ always holds}) \\
\iff & 0
\end{aligned}
$$

When z is x, we get

$$
\begin{aligned}
\mathit{Reach}[\mathtt{w},\mathtt{x}]' \iff & w \neq \mathit{null} \wedge x \neq \mathit{null} \wedge n^{+}(w, x) \wedge \neg(n^{*}(w, x) \wedge n^{+}(x, x)) \\
& \text{(Lemma 2)} \\
\iff & w \neq \mathit{null} \wedge x \neq \mathit{null} \wedge n^{+}(w, x) \\
& (n^{+}(x, x) \text{ does not hold since lists are acyclic}) \\
\iff & \mathit{Reach}[\mathtt{w},\mathtt{x}] \\
& \text{(Folding)}
\end{aligned}
$$

B.8 Deriving Update Formulae for `x.n=y`

We assume the this statement is preceded by `x.n = null`, and therefore $x \neq \mathit{null}$ and $n(x) = \mathit{null}$ hold. We also assume (and check) the following assumptions: (i) there are no incoming edges to `y`, i.e., $\forall u : \neg n(u, y)$; and (ii) $\neg n^{*}(y, x)$.

Since the concrete semantics only changes the function n, the predicates of the form $\mathit{Aliased}[\mathtt{w},\mathtt{z}]$ and $\mathit{NotNull}[\mathtt{z}]$ retain their current values.

Assume that $w, z \in \mathit{PVar} \setminus \{x, y\}$ and that w and z are different ($\mathit{Next}[\mathtt{w},\mathtt{z}]$ is defined for different variables).

$$
\begin{aligned}
\mathit{Next}[\mathtt{w},\mathtt{z}]' \iff & w' \neq \mathit{null} \wedge z' \neq \mathit{null} \wedge n'(w') = z' \\
& \text{(Unfolding the defining formula of } \mathit{Next}[\mathtt{w},\mathtt{z}]'\text{)} \\
\iff & w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (\lambda\, v.v = x?y : n(v))(w) = z \\
& \text{(Unfolding the concrete semantics of } \mathtt{x.n=y}\text{)} \\
\iff & w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (w = x \wedge y = z \vee w \neq x \wedge n(w) = z) \\
& (\lambda \text{ application}) \\
\iff & (w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge w = x \wedge y = z) \vee \\
& (w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge w \neq x \wedge n(w) = z) \\
& \text{(Distributing } \vee \text{ over } \wedge\text{)} \\
\iff & (w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge w = x \wedge y = z) \vee \\
& (w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n(w) = z) \\
& \text{(since } n(x) = \mathit{null} \text{ we have that } w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n(w) = z \\
& \text{implies } w \neq x\text{)} \\
\iff & \mathit{Next}[\mathtt{w},\mathtt{z}] \vee \mathit{Aliased}[\mathtt{w},\mathtt{x}] \wedge \mathit{Aliased}[\mathtt{z},\mathtt{y}] \\
& \text{(Folding)}
\end{aligned}
$$

When w is y, we get

$$
\begin{aligned}
\mathit{Next}[\mathtt{y},\mathtt{z}]' \iff & y' \neq \mathit{null} \wedge z' \neq \mathit{null} \wedge n'(y') = z' \\
& \text{(Unfolding the defining formula of } \mathit{Next}[\mathtt{y},\mathtt{z}]'\text{)} \\
\iff & y \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (\lambda\, v.v = x?y : n(v))(y) = z \\
& \text{(Unfolding the concrete semantics of } \mathtt{x.n=y}\text{)} \\
\iff & y \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (y = x \wedge y = z \vee y \neq x \wedge n(y) = z) \\
& (\lambda \text{ application}) \\
\iff & y \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n(y) = z \\
& \text{(Recall our assumption that } x \neq y\text{)} \\
\iff & y \neq \mathit{null} \wedge z \neq \mathit{null} \wedge y \neq x \wedge n(y) = z \\
\iff & \mathit{Next}[\mathtt{y},\mathtt{z}] \\
& \text{(Folding)}
\end{aligned}
$$

Since we assume that $\neg n^*(y, x)$ the current value of $\mathit{Next}[\mathtt{y},\mathtt{x}]$ is 0. When w is y and z is x, we get

$$
\begin{aligned}
\mathit{Next}[\mathtt{y},\mathtt{x}]' \iff & y' \neq \mathit{null} \wedge x' \neq \mathit{null} \wedge n'(y') = x' \\
& \text{(Unfolding the defining formula of } \mathit{Next}[\mathtt{y},\mathtt{x}]'\text{)} \\
\iff & y \neq \mathit{null} \wedge x \neq \mathit{null} \wedge (\lambda\, v.v = x?y : n(v))(y) = x \\
& \text{(Unfolding the concrete semantics of } \mathtt{x.n=y}\text{)} \\
\iff & y \neq \mathit{null} \wedge x \neq \mathit{null} \wedge (y = x \wedge y = x \vee y \neq x \wedge n(y) = x) \\
& (\lambda \text{ application}) \\
\iff & 0 \\
& \text{(Recall our assumption that } x \neq y \text{ and} \neg n^*(y, x)\text{)}
\end{aligned}
$$

When w is x, after applying the λ, we get

$$\begin{aligned}
\mathit{Next}[\mathtt{x},\mathtt{z}]' &\iff x \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (x = x \wedge y = z \vee x \neq x \wedge n(x) = z) \\
&\quad (\lambda \text{ application}) \\
&\iff x \neq \mathit{null} \wedge z \neq \mathit{null} \wedge y = z \\
&\quad (\text{Canceling trivial sub-terms}) \\
&\iff \mathit{Aliased}[\mathtt{z},\mathtt{y}] \\
&\quad (\text{Folding})
\end{aligned}$$

When w is x and z is y, after applying the λ, we get

$$\begin{aligned}
\mathit{Next}[\mathtt{x},\mathtt{y}]' &\iff x \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (x = x \wedge y = y \vee x \neq x \wedge n(x) = y) \\
&\quad (\lambda \text{ application}) \\
&\iff x \neq \mathit{null} \wedge y \neq \mathit{null} \\
&\quad (\text{Canceling trivial sub-terms}) \\
&\iff y \neq \mathit{null} \\
&\quad (x \text{ is known to be non-null}) \\
&\iff \mathit{NotNull}[\mathtt{y}] \\
&\quad (\text{Folding})
\end{aligned}$$

Assume that $z \in \mathit{PVar} \setminus \{x\}$.

$$\begin{aligned}
\mathit{NextNull}[\mathtt{z}]' &\iff z' \neq \mathit{null} \wedge n'(z') = \mathit{null} \\
&\quad (\text{Unfolding the defining formula of } \mathit{NextNull}[\mathtt{z}]') \\
&\iff z \neq \mathit{null} \wedge (\lambda\ v.v = x?y : n(v))(z) = \mathit{null} \\
&\quad (\text{Unfolding the concrete semantics of } \mathtt{x.n=y}) \\
&\iff z \neq \mathit{null} \wedge (z = x \wedge y = \mathit{null} \vee z \neq x \wedge n(z) = \mathit{null}) \\
&\quad (\lambda \text{ application}) \\
&\iff z \neq \mathit{null} \wedge z = x \wedge y = \mathit{null} \vee z \neq \mathit{null} \wedge z \neq x \wedge n(z) = \mathit{null} \\
&\quad (\text{Distributing } \wedge \text{ over } \vee) \\
&\iff \mathit{Aliased}[\mathtt{z},\mathtt{x}] \wedge \neg \mathit{NotNull}[\mathtt{y}] \vee z \neq \mathit{null} \wedge z \neq x \wedge n(z) = \mathit{null} \\
&\quad (\text{Folding first disjunct}) \\
&\iff \mathit{Aliased}[\mathtt{z},\mathtt{x}] \wedge \neg \mathit{NotNull}[\mathtt{y}] \vee z \neq \mathit{null} \wedge n(z) = \mathit{null} \wedge \\
&\quad (z \neq x \vee x \neq \mathit{null}) \\
&\quad (\text{Using the assumption that } x \neq \mathit{null}) \\
&\iff \mathit{Aliased}[\mathtt{z},\mathtt{x}] \wedge \neg \mathit{NotNull}[\mathtt{y}] \vee z \neq \mathit{null} \wedge n(z) = \mathit{null} \wedge \\
&\quad (z \neq x \vee x \neq \mathit{null} \vee z \neq \mathit{null}) \\
&\quad (\text{Using the Boolean algebra rule } A \wedge B \leftrightarrow A \wedge (\neg A \vee B) \\
&\quad \text{where } A = (z \neq \mathit{null}) \text{ and } B = (z \neq x \vee x \neq \mathit{null})) \\
&\iff \mathit{Aliased}[\mathtt{z},\mathtt{x}] \wedge \neg \mathit{NotNull}[\mathtt{y}] \vee \neg \mathit{Aliased}[\mathtt{z},\mathtt{x}] \wedge \mathit{NextNull}[\mathtt{z}] \\
&\quad (\text{Folding})
\end{aligned}$$

When z is x, we get

$$
\begin{aligned}
\mathit{NextNull}[\mathtt{x}]' &\iff x' \neq \mathit{null} \wedge n'(x') = \mathit{null} \\
&\qquad \text{(Unfolding the defining formula of } \mathit{NextNull}[\mathtt{x}]'\text{)} \\
&\iff x \neq \mathit{null} \wedge (\lambda\, v.v = x ? y : n(v))(x) = \mathit{null} \\
&\qquad \text{(Unfolding the concrete semantics of } \mathtt{x.n=y}\text{)} \\
&\iff x \neq \mathit{null} \wedge (x = x \wedge y = \mathit{null} \vee x \neq x \wedge n(x) = \mathit{null}) \\
&\qquad (\lambda \text{ application)} \\
&\iff y = \mathit{null} \\
&\qquad \text{(Canceling trivial sub-terms)} \\
&\iff \mathit{IsNull}[\mathtt{y}] \\
&\qquad \text{(Folding and using the } \mathit{IsNull}[\mathtt{y}] \text{ macro)}
\end{aligned}
$$

When z is y, we get

$$
\begin{aligned}
\mathit{NextNull}[\mathtt{y}]' &\iff y' \neq \mathit{null} \wedge n'(y') = \mathit{null} \\
&\qquad \text{(Unfolding the defining formula of } \mathit{NextNull}[\mathtt{y}]'\text{)} \\
&\iff y \neq \mathit{null} \wedge (\lambda\, v.v = x ? y : n(v))(y) = \mathit{null} \\
&\qquad \text{(Unfolding the concrete semantics of } \mathtt{x.n=y}\text{)} \\
&\iff y \neq \mathit{null} \wedge (y = x \wedge y = \mathit{null} \vee y \neq x \wedge n(y) = \mathit{null}) \\
&\qquad (\lambda \text{ application)} \\
&\iff y \neq \mathit{null} \wedge n(y) = \mathit{null} \\
&\qquad \text{(Recall our assumption that } x \neq y\text{)} \\
&\iff \mathit{NextNull}[\mathtt{y}]
\end{aligned}
$$

Assume that $w, z \in \mathit{PVar} \setminus \{x, y\}$ and that w and z are different ($\mathit{Reach}[\mathtt{w},\mathtt{z}]$ is defined for different variables).

$$
\begin{aligned}
\mathit{Reach}[\mathtt{w},\mathtt{z}]' &\iff w' \neq \mathit{null} \wedge z' \neq \mathit{null} \wedge {n'}^{+}(w', z') \\
&\qquad \text{(Unfolding the defining formula of } \mathit{Reach}[\mathtt{w},\mathtt{z}]'\text{)} \\
&\iff w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (\lambda\, v.v = x ? y : n(v))^{+}(w, z) \\
&\qquad \text{(Unfolding the concrete semantics of } \mathtt{x.n=y}\text{)} \\
&\iff w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (n^{+}(w, z) \vee n^{*}(w, x) \wedge n^{*}(y, z)) \\
&\qquad \text{(Lemma 1)} \\
&\iff w \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (n^{+}(w, z) \vee n^{*}(w, x) \wedge n^{*}(y, z) \wedge y \neq \mathit{null}) \\
&\qquad (z \neq \mathit{null} \wedge n^{*}(y, z) \implies y \neq \mathit{null}) \\
&\iff \mathit{Reach}[\mathtt{w},\mathtt{z}] \vee \mathit{ReachOrAliased}[\mathtt{w},\mathtt{x}] \wedge \mathit{ReachOrAliased}[\mathtt{y},\mathtt{z}] \\
&\qquad \text{(Folding and using macros)}
\end{aligned}
$$

When w is x, we get

$$\begin{aligned}
\mathit{Reach}[\mathtt{x},\mathtt{z}]' &\iff x' \neq \mathit{null} \land z' \neq \mathit{null} \land n'^{+}(x', z') \\
&\quad \text{(Unfolding the defining formula of } \mathit{Reach}[\mathtt{x},\mathtt{z}]'\text{)} \\
&\iff x \neq \mathit{null} \land z \neq \mathit{null} \land (\lambda\, v.v = x?y : n(v))^{+}(x, z) \\
&\quad \text{(Unfolding the concrete semantics of } \mathtt{x.n=y}\text{)} \\
&\iff x \neq \mathit{null} \land z \neq \mathit{null} \land (n^{+}(x, z) \lor n^{*}(x, x) \land n^{*}(y, z)) \\
&\quad \text{(Lemma 1)} \\
&\iff x \neq \mathit{null} \land z \neq \mathit{null} \land (n^{+}(x, z) \lor n^{*}(x, x) \land n^{*}(y, z) \land y \neq \mathit{null}) \\
&\quad (z \neq \mathit{null} \land n^{*}(y, z) \implies y \neq \mathit{null}) \\
&\iff z \neq \mathit{null} \land (n^{+}(x, z) \land n^{*}(y, z) \land y \neq \mathit{null}) \\
&\quad \text{(Canceling the trivial term } n^{*}(x, x)\text{)} \\
&\iff x \neq \mathit{null} \land z \neq \mathit{null} \land (n^{*}(y, z) \land y \neq \mathit{null}) \\
&\quad \text{(Canceling } n^{+}(x, z) \text{ since } n(x) = \mathit{null}\text{)} \\
&\iff \mathit{ReachOrAliased}[\mathtt{y},\mathtt{z}] \\
&\quad \text{(Folding and using macros)}
\end{aligned}$$

When z is y, we get

$$\begin{aligned}
\mathit{Reach}[\mathtt{w},\mathtt{y}]' &\iff w' \neq \mathit{null} \land y' \neq \mathit{null} \land n'^{+}(w', y') \\
&\quad \text{(Unfolding the defining formula of } \mathit{Reach}[\mathtt{w},\mathtt{y}]'\text{)} \\
&\iff w \neq \mathit{null} \land y \neq \mathit{null} \land (\lambda\, v.v = x?y : n(v))^{+}(w, y) \\
&\quad \text{(Unfolding the concrete semantics of } \mathtt{x.n=y}\text{)} \\
&\iff w \neq \mathit{null} \land y \neq \mathit{null} \land (n^{+}(w, y) \lor n^{*}(w, x) \land n^{*}(y, y)) \\
&\quad \text{(Lemma 1)} \\
&\iff w \neq \mathit{null} \land y \neq \mathit{null} \land (n^{+}(w, y) \lor n^{*}(w, x)) \\
&\quad \text{(Canceling the trivial term } n^{*}(y, y)\text{)} \\
&\iff w \neq \mathit{null} \land y \neq \mathit{null} \land n^{*}(w, x) \\
&\quad \text{(Canceling } n^{+}(w, y) \text{ since we assume that y has no} \\
&\quad \text{incoming edges)} \\
&\iff \mathit{NotNull}[\mathtt{y}] \land \mathit{ReachOrAliased}[\mathtt{w},\mathtt{x}] \\
&\quad \text{(Folding and using macros)}
\end{aligned}$$

When w is x and z is y, we get from the last line before folding for the predicate $\mathit{Reach}[\mathtt{w},\mathtt{y}]'$

$$\begin{aligned}
\mathit{Reach}[\mathtt{x},\mathtt{y}]' &\iff x \neq \mathit{null} \land y \neq \mathit{null} \land n^{*}(x, x) \\
&\iff x \neq \mathit{null} \land y \neq \mathit{null} \\
&\quad \text{(Canceling the trivial term } n^{*}(x, x)\text{)} \\
&\iff \mathit{NotNull}[\mathtt{y}] \\
&\quad \text{(Folding)}
\end{aligned}$$

B.9 Deriving Update Formulae for `x=y.n`

We assume the this statement is preceded by `assert(y!=null)`, and therefore $y \neq \mathit{null}$ holds. We also assume that the statement is preceded by `x=null`, and therefore $x = \mathit{null}$ holds.

Since the concrete semantics only changes `x`, values of predicates that do not involve `x` remain unchanged.

Lemma 3. *When $y \neq null$ holds the following equivalence also holds:*

$$n(y) \neq null \iff \neg NextNull[\mathtt{y}] \ .$$

Proof. The following equivalence-preserving transformations proves the claim:

$$\begin{aligned}
&n(y) \neq null \iff \\
&\neg(n(y) = null) \iff \\
&\neg(y \neq null \wedge n(y) = null) \iff \\
&\neg NextNull[\mathtt{y}] \ .
\end{aligned}$$

□

We know that $y \neq null$ holds for all inputs for this transformers and thus we replace $n(y) \neq null$ with $\neg NextNull[\mathtt{y}]$ in the following derivations.

$$\begin{aligned}
NotNull[\mathtt{x}]' &\iff x' \neq null \\
&\quad \text{(Unfolding the defining formula of } NotNull[\mathtt{x}]'\text{)} \\
&\iff n(y) \neq null \\
&\quad \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
&\iff \neg NextNull[\mathtt{y}] \\
&\quad \text{(Lemma 3)}
\end{aligned}$$

Assume that $z \in PVar \setminus \{x, y\}$.

$$\begin{aligned}
Aliased[\mathtt{x}, \mathtt{z}]' &\iff x' \neq null \wedge z' \neq null \wedge x' = z') \\
&\quad \text{(Unfolding the defining formula of } Aliased[\mathtt{x}, \mathtt{z}]'\text{)} \\
&\iff n(y) \neq null \wedge z \neq null \wedge n(y) = z \\
&\quad \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
&\iff z \neq null \wedge n(y) = z \\
&\quad (z \neq null \wedge n(y) = z \implies n(y) \neq null) \\
&\iff y \neq null \wedge z \neq null \wedge n(y) = z \\
&\quad \text{(We know that } y \neq null \text{ holds)} \\
&\quad Next[\mathtt{y}, \mathtt{z}] \\
&\quad \text{(Folding)}
\end{aligned}$$

Since The definition of $Aliased[\mathtt{x}, \mathtt{z}]$ and $Aliased[\mathtt{z}, \mathtt{x}]$ is symmetric the same update is used for $Aliased[\mathtt{z}, \mathtt{x}]$.

Lemma 4. *If u and v are elements of unshared lists such that $u \neq null$, $v \neq null$, then $n(u) = n(v) \leftrightarrow u = v$.*

Proof. Follows from Definition 2. □

Assume that $z \in PVar \setminus \{x, y\}$.

$$\begin{aligned}
\mathit{Next}[\mathtt{z},\mathtt{x}]' \iff & z' \neq \mathit{null} \wedge x' \neq \mathit{null} \wedge n'(z') = x') \\
& \text{(Unfolding the defining formula of } \mathit{Next}[\mathtt{z},\mathtt{x}]'\text{)} \\
\iff & z \neq \mathit{null} \wedge n(y) \neq \mathit{null} \wedge n(z) = n(y)) \\
& \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
\iff & z \neq \mathit{null} \wedge y \neq \mathit{null} \wedge n(y) \neq \mathit{null} \wedge n(z) = n(y) \\
& \text{(We know that } y \neq \mathit{null} \text{ holds)} \\
\iff & z \neq \mathit{null} \wedge y \neq \mathit{null} \wedge n(y) \neq \mathit{null} \wedge z = y \\
& \text{(}\mathit{Lemma}\text{ 4)} \\
& \neg \mathit{NextNull}[\mathtt{y}] \wedge \mathit{Aliased}[\mathtt{z},\mathtt{y}] \\
& \text{(Folding, using Lemma 3)}
\end{aligned}$$

When z is y, we get

$$\begin{aligned}
\mathit{Next}[\mathtt{y},\mathtt{x}]' \iff & y' \neq \mathit{null} \wedge x' \neq \mathit{null} \wedge n'(y') = x') \\
& \text{(Unfolding the defining formula of } \mathit{Next}[\mathtt{y},\mathtt{x}]'\text{)} \\
\iff & y \neq \mathit{null} \wedge n(y) \neq \mathit{null} \wedge n(y) = n(y)) \\
& \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
\iff & y \neq \mathit{null} \wedge n(y) \neq \mathit{null} \\
& \text{(Removing } n(y) = n(y)\text{)} \\
\iff & \neg \mathit{NextNull}[\mathtt{y}] \\
& \text{(Folding)}
\end{aligned}$$

Lemma 5. *For acyclic lists, and two list elements y and z, such that $z, y, n(y) \neq null$, the following holds:*

$$n^+(n(y), z) \iff n^+(y, z) \wedge n(y) \neq z \ \ .$$

Proof. Assume that $n^+(n(y), z)$ holds. Then there exists the path $u_1, \ldots, u_k$ such that $u_1 = n(y)$, $u_k = z$, and $k > 1$. Therefore, there exists the path $y, u_1, \ldots, u_k$. Hence, $n^+(y, z)$ holds. Since all paths are acyclic, the elements in $y, u_1, \ldots, u_k$ are all distinct. Hence, $n(y) \neq z$.

Now assume that $n^+(y, z) \wedge n(y) \neq z$ holds. Then, there exists a path $u_1, \ldots, u_k$, such that $u_1 = y$, $u_k = z$, and $k > 1$. Since $n(y) \neq z$ holds, we know that $k > 2$. Therefore, there exists the path $u_2, \ldots, u_k$ where $u_2 = n(y)$ and the length of the path is at least 1 (i.e., $k - 2 > 1$). Therefore, $n^+(n(y), z)$ holds. □

Assume that $z \in PVar \setminus \{x, y\}$.

$$\begin{aligned}
\mathit{Reach}[\mathtt{x},\mathtt{z}]' \iff & x' \neq \mathit{null} \wedge z' \neq \mathit{null} \wedge n'^+(x', z') \\
& \text{(Unfolding the defining formula of } \mathit{Reach}[\mathtt{x},\mathtt{z}]'\text{)} \\
\iff & n(y) \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n^+(n(y), z) \\
& \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
\iff & n(y) \neq \mathit{null} \wedge z \neq \mathit{null} \wedge (n^+(y, z) \wedge \neg n(y) = z) \\
& \text{(}\mathit{Lemma}\text{ 5)} \\
& \neg \mathit{NextNull}[\mathtt{y}] \wedge \mathit{Reach}[\mathtt{y},\mathtt{z}] \wedge \neg \mathit{Next}[\mathtt{y},\mathtt{z}] \\
& \text{(Folding, using Lemma 3)}
\end{aligned}$$

Lemma 6. *When all lists are acyclic, given two list elements y and z, such that $z, y, n(y) \neq null$, the following holds:*

$$n^+(z, n(y)) \Longleftrightarrow n^+(z, y) \vee z = y \enspace .$$

Proof. Assume that $n^+(z, n(y))$ holds. Then, there exists the path $u_1, \ldots, u_k$ where $u_1 = z$, $u_k = n(y)$, and $k > 1$. From the assumption that all lists are acyclic, we also know that $u_{k-1} = y$. Therefore, there exists a path between z and y of length greater or equal to 0. Hence, $n^+(z, y) \vee z = y$ holds.

Assume that $n^+(z, y) \vee z = y$ holds. Then, there exists a path of length greater or equal to 0 from z to y. Therefore, there exists a path of length greater than 0 from z to $n(y)$. Hence, $n^+(z, n(y))$ holds. □

Assume that $z \in PVar \setminus \{x, y\}$.

$$\begin{aligned}
Reach[\mathtt{z},\mathtt{x}]' \Longleftrightarrow\; & z' \neq null \wedge x' \neq null \wedge {n'}^+(z', x') \\
& \text{(Unfolding the defining formula of } Reach[\mathtt{z},\mathtt{x}]'\text{)} \\
\Longleftrightarrow\; & z \neq null \wedge n(y) \neq null \wedge n^+(z, n(y)) \\
& \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
& z \neq null \wedge n(y) \neq null \wedge (n^+(z, y) \vee z = y) \\
& (\textit{Lemma } 6) \\
& \neg NextNull[\mathtt{y}] \wedge ReachOrAliased[\mathtt{z},\mathtt{y}] \\
& \text{(Folding, using Lemma 3 and macros)}
\end{aligned}$$

When z is y, we get

$$\begin{aligned}
Reach[\mathtt{y},\mathtt{x}]' \Longleftrightarrow\; & y' \neq null \wedge x' \neq null \wedge {n'}^+(y', x') \\
& \text{(Unfolding the defining formula of } Reach[\mathtt{y},\mathtt{x}]'\text{)} \\
\Longleftrightarrow\; & y \neq null \wedge n(y) \neq null \wedge n^+(y, n(y)) \\
& \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
& y \neq null \wedge n(y) \neq null \wedge (n^+(y, y) \vee y = y) \\
& (\textit{Lemma } 6) \\
& y \neq null \wedge n(y) \neq null \\
& \neg NextNull[\mathtt{y}] \\
& \text{(Folding)}
\end{aligned}$$

Lemma 7. *The condition formulae shown in Fig. 3 are mutually-exclusive, and exactly one of them is enabled.*

Proof. The condition formulae are:

Case 1: $NextNull[\mathtt{y}]$,
Case 2: $\neg NextNull[\mathtt{y}] \wedge NextPtByVar[\mathtt{y}]$,
Case 3: $\neg NextNull[\mathtt{y}] \wedge ReachNull[\mathtt{y}]$, and
Case 4: $\neg NextNull[\mathtt{y}] \wedge \neg ReachNull[\mathtt{y}] \wedge \neg NextPtByVar[\mathtt{y}]$.

Recall that, for the statement `x=y.n` to be executed, we require that $y \neq null$ hold, hence $NotNull[\mathtt{y}]$ holds for all of the cases.

The formula $ReachNull[\mathtt{y}]$ means that by following `n` links, we do not encounter any element that is pointed-to by some variable. Therefore, $ReachNull[\mathtt{y}]$ implies $\neg NextPtByVar[\mathtt{y}]$ (which means that the next element after `y` is not pointed-to by any variable). Therefore, the condition formula of case 3 rewrites

to

$$NotNull[\mathtt{y}] \wedge \neg NextNull[\mathtt{y}] \wedge ReachNull[\mathtt{y}] \wedge \neg NextPtByVar[\mathtt{y}] \enspace .$$

Now it is evident from the structure of the condition formulae that they partition the set of concrete states since they contain complementary sub-formulae. □

We continue to show the derivation of the update formulae for the predicates $NextNull[\mathtt{x}]$ and $Next[\mathtt{x},\mathtt{z}]$ by considering each case separately.

Case 1: the condition $\boldsymbol{NextNull}[\mathtt{y}]$ holds. Since a statement `x=y.n` is preceded by a statement `x=null`, the fact that the condition formula $NotNull[\mathtt{y}] \wedge NextNull[\mathtt{y}]$ holds means that $n(y) = null$, and thus no change occurs in the concrete semantics. Therefore, the values of all predicates remain unchanged.

Case 2 : the condition $\boldsymbol{NotNull}[\mathtt{y}] \wedge \neg \boldsymbol{NextNull}[\mathtt{y}] \wedge \boldsymbol{NextPtByVar}[\mathtt{y}]$ holds.

Lemma 8. *Under the condition of case* 2*, the following holds:*

$$n(n(y)) = null \Longleftrightarrow \bigvee_{w \in PVar \setminus \{x,y\}} Next[\mathtt{y},\mathtt{w}] \wedge NextNull[\mathtt{w}] \enspace .$$

Proof. We first show that the right-hand side implies the left-hand side:

$\bigvee_{w \in PVar \setminus \{x,y\}} Next[\mathtt{y},\mathtt{w}] \wedge NextNull[\mathtt{w}] \Longrightarrow$
(Assuming the left-hand side)
$\bigvee_{w \in PVar \setminus \{x,y\}} y \neq null \wedge w \neq null \wedge n(y) = w \wedge n(w) = null \Longrightarrow$
(Unfolding the definitions of the predicates)
$\exists u : y \neq null \wedge u \neq null \wedge n(y) = u \wedge n(u) = null \Longleftrightarrow$
(Applying a valid first-order logic transformation:
Let c be a constant. Then $\phi(c) \implies \exists v.\phi(v)$)
$\exists u : n(y) = u \wedge n(u) = null \Longleftrightarrow$
(Using the condition to eliminate trivial terms)
$n(n(y)) = null \Longleftrightarrow$
(Folding by using the meaning of applying a function twice)

We now show that the left-hand side implies the right-hand side:
$n(n(y)) = null \Longleftrightarrow$
(Assuming the left-hand side)
$\exists u : n(y) = u \wedge n(u) = null \Longleftrightarrow$
(Unfolding the meaning of applying a function twice)
$\exists u : n(y) = u \wedge (\bigvee_{z \in PVar \setminus \{x,y\}} u = z) \wedge n(u) = null \Longleftrightarrow$
(Since $NextPtByVar[\mathtt{y}]$ and $x = null$ hold, $u = n(y) \implies \bigvee_{z \in PVar \setminus \{x,y\}} u = z$)
$\exists u : \bigvee_{z \in PVar \setminus \{x,y\}} (n(y) = u \wedge u = z \wedge n(u) = null) \Longleftrightarrow$
(Distributivity of $\vee$ over $\wedge$)
$\exists u : \bigvee_{z \in PVar \setminus \{x,y\}} (n(y) = z \wedge u = z \wedge n(z) = null) \Longrightarrow$
(Using $u = z$ to substitute u by z)
$\exists u : \bigvee_{z \in PVar \setminus \{x,y\}} (n(y) = z \wedge n(z) = null) \Longleftrightarrow$
(Removing a conjunct weakens the formula)
$\bigvee_{z \in PVar \setminus \{x,y\}} (n(y) = z \wedge n(z) = null) \Longleftrightarrow$
(Removing the quantifier with empty binding)
$\bigvee_{z \in PVar \setminus \{x,y\}} (Next[\mathtt{y}, \mathtt{z}] \wedge NextNull[\mathtt{z}])$

□

$NextNull[\mathtt{x}]' \Longleftrightarrow x' \neq null \wedge n'(x') = null$
(Unfolding the defining formula of $NextNull[\mathtt{x}]'$)
$n(y) \neq null \wedge n(n(y)) = null$
(Unfolding the concrete semantics of `x=y.n`)
$\bigvee_{w \in PVar \setminus \{x,y\}} Next[\mathtt{y}, \mathtt{w}] \wedge NextNull[\mathtt{w}]$
(*Lemma* 8)

Lemma 9. *Under the condition of case* 2*, the following holds for any* $z \in PVar \setminus y$*:*

$$n(n(y)) = z \Longleftrightarrow \bigvee_{w \in PVar \setminus \{x,y\}} Next[\mathtt{y}, \mathtt{w}] \wedge Next[\mathtt{w}, \mathtt{z}] \ .$$

Proof. We first show that the right-hand side implies the left-hand side:
$\bigvee_{w \in PVar \setminus \{x,y\}} Next[\mathtt{y}, \mathtt{w}] \wedge Next[\mathtt{w}, \mathtt{z}] \Longrightarrow$
(Assuming the left-hand side)
$\bigvee_{w \in PVar \setminus \{x,y\}} y \neq null \wedge w \neq null \wedge n(y) = w \wedge n(w) = z \Longrightarrow$
(Unfolding the definitions of the predicates)
$\exists u : y \neq null \wedge u \neq null \wedge n(y) = u \wedge n(u) = z \Longleftrightarrow$
(Applying a valid first-order logic transformation:
Let c be a constant. Then $\phi(c) \implies \exists v. \phi(v)$)
$\exists u : n(y) = u \wedge n(u) = z \Longleftrightarrow$
(Using the condition to eliminate trivial terms)
$n(n(y)) = z$
(Folding by using the meaning of applying a function twice)

We now show that the left-hand side implies the right-hand side:

$n(n(y)) = z \iff$
(Assuming the left-hand side)
$\exists u : n(y) = u \wedge n(u) = z \iff$
(Unfolding the meaning of applying a function twice)
$\exists u : n(y) = u \wedge (\bigvee_{w \in PVar \setminus \{x,y\}} u = w) \wedge n(u) = z \iff$
(Since $NextPtByVar[\mathtt{y}]$, $x = null$, and $n(u) \neq null$ hold,
$u = n(y) \implies \bigvee_{w \in PVar \setminus \{x,y\}} u = w$)
$\exists u : \bigvee_{w \in PVar \setminus \{x,y\}} (n(y) = u \wedge u = w \wedge n(u) = z) \iff$
(Distributivity of $\vee$ over $\wedge$)
$\exists u : \bigvee_{w \in PVar \setminus \{x,y\}} (n(y) = w \wedge u = w \wedge n(w) = z) \implies$
(Using $u = w$ to substitute u by w)
$\exists u : \bigvee_{w \in PVar \setminus \{x,y\}} (n(y) = w \wedge n(w) = z) \iff$
(Removing a conjunct weakens the formula)
$\bigvee_{w \in PVar \setminus \{x,y\}} (n(y) = w \wedge n(w) = z) \iff$
(Removing the quantifier with empty binding)
$\bigvee_{w \in PVar \setminus \{x,y\}} (Next[\mathtt{y},\mathtt{w}] \wedge Next[\mathtt{w},\mathtt{z}])$

□

Assume that $z \in PVar \setminus \{x, y\}$.

$Next[\mathtt{x},\mathtt{z}]' \iff x' \neq null \wedge z' \neq null \wedge n'(x') = z'$
(Unfolding the defining formula of $Next[\mathtt{x},\mathtt{z}]'$)
$\iff n(y) \neq null \wedge z \neq null \wedge n(n(y)) = z$
(Unfolding the concrete semantics of `x=y.n`)
$\bigvee_{w \in PVar \setminus \{x,y\}} Next[\mathtt{y},\mathtt{w}] \wedge Next[\mathtt{w},\mathtt{z}]$
(*Lemma* 9)

Case 3 : *NotNull*[y] ∧ ¬*NextNull*[y] ∧ *ReachNull*[y] holds. The next lemma shows that there are concrete states that are indistinguishable by the abstraction but are distinguishable by the focus formula $NextNextNull[\mathtt{y}] \equiv n(n(y)) = null$.

Lemma 10. *When the condition of case* 3 *holds, the predicate NextNextNull*[y] *is a focus predicate.*

In other words, let σ_0 be a concrete state that satisfies the condition formula. Then, there exist two concrete states (containing only acyclic unshared lists) σ_1, σ_2 such that $\sigma_1 \models NextNextNull[\mathtt{y}]$ and $\sigma_2 \models \neg NextNextNull[\mathtt{y}]$, and $\alpha(\sigma_0) = \alpha(\sigma_1) = \alpha(\sigma_2)$.

Proof. We show that we can modify σ_0 to obtain two concrete states, σ_1 and σ_2, such that $\alpha(\sigma_0) = \alpha(\sigma_1) = \alpha(\sigma_2)$, and $\sigma_1 \models NextNextNull[\mathtt{y}]$ and $\sigma_2 \models \neg NextNextNull[\mathtt{y}]$.

From the fact that σ_0 satisfies the condition, we have that σ_0 contains a path $u_1, \ldots, u_k$ such that $u_1 = y$, $u_k = null$ (since $y \neq null$ holds and there exists a path from every non-null variable to $null$, $k > 1$ (since $\neg NextNull[\mathtt{y}]$ holds), and $u_2, \ldots, u_{k-1}$ are not pointed-to by any variable (since $ReachNull[\mathtt{y}]$ holds).

To obtain σ_1, we replace the path $u_1, \ldots, u_k$ with the path u_1, u_2, u_k, by removing the elements $u_3, \ldots, u_{k-1}$ from the heap and adjusting the `n` links accordingly. Thus, $n(n(y)) = \mathit{null}$ in σ_1, i.e., $\sigma_1 \models \mathit{NextNextNull}[\mathtt{y}]$.

To obtain σ_2, we replace the path $u_1, \ldots, u_k$ with the path u_1, u', u_2, u_k where u' is a new element added to the heap between u_1 and u_2. Since $k > 1$ in σ_0, we have that $u_2 \neq \mathit{null}$. Thus, $n(n(y)) \neq \mathit{null}$ in σ_2, i.e., $\sigma_2 \models \neg\mathit{NextNextNull}[\mathtt{y}]$.

Intuitively, to obtain σ_1 and σ_2, we have changed σ_0 by modifying the length of the path from y to *null*, which used to be greater than 1. However, the abstraction we use does not distinguish between lengths of lists greater than 1, and thus we get 3 concrete states that are equivalent under the abstraction.

Now, since in σ_0, none of the elements $u_2, \ldots, u_{k-1}$ are pointed-to by variables, every logical constant z representing a variable `z`, evaluates to the same element in σ_0, σ_1, and σ_2. Therefore, $[\![\mathit{NotNull}[\mathtt{z}]]\!]^{\sigma_0} = [\![\mathit{NotNull}[\mathtt{z}]]\!]^{\sigma_1} = [\![\mathit{NotNull}[\mathtt{z}]]\!]^{\sigma_2}$, for every $z \in \mathit{PVar}$. As well, $[\![\mathit{Aliased}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_0} = [\![\mathit{Aliased}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_1} = [\![\mathit{Aliased}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_2}$ for every $z, w \in \mathit{PVar}$.

Let z be in PVar such that $z = \mathit{null}$ in σ_0. Then, $z = \mathit{null}$ in σ_1 and σ_2. Therefore, $[\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_0} = [\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_1} = [\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_2} = 0$. This also means that for every $w \in \mathit{PVar}$, we have $[\![\mathit{Next}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_0} = [\![\mathit{Next}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_1} = [\![\mathit{Next}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_2} = 0$, $[\![\mathit{Next}[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![\mathit{Next}[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![\mathit{Next}[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2} = 0$, $[\![\mathit{Reach}[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![\mathit{Reach}[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![\mathit{Reach}[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2} = 0$, and $[\![\mathit{Reach}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_0} = [\![\mathit{Reach}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_1} = [\![\mathit{Reach}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_2} = 0$.

We now have to show that the equality holds for predicates of the form $\mathit{NextNull}[\mathtt{z}]$, $\mathit{Next}[\mathtt{z},\mathtt{w}]$, and $\mathit{Reach}[\mathtt{z},\mathtt{w}]$ such that both z and w are different from *null* in σ_0.

The only element pointed-to by a variable for which the outgoing edge has changed is the one pointed-to by `y`. Therefore, for every $z \in \mathit{PVar} \setminus \{y\}$ such that $y \neq z$, $[\![n(z) = \mathit{null}]\!]^{\sigma_0} = [\![n(z) = \mathit{null}]\!]^{\sigma_1} = [\![n(z) = \mathit{null}]\!]^{\sigma_2}$. Hence, $[\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_0} = [\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_1} = [\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_2}$.

For every $z \in \mathit{PVar} \setminus \{y\}$ such that $y = z$, we have $[\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_0} = 0$ (since the statement is preceded by `assert(y!=null)`). In σ_1 and σ_2, we have $n(z) = u_1$. Thus, $[\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_0} = [\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_1} = [\![\mathit{NextNull}[\mathtt{z}]]\!]^{\sigma_2} = 0$.

Let w and z be two variables in PVar such that w and z are distinct and both are different from *null*. The path between w and z is the same in σ_0 as in σ_1 and σ_2. Therefore, $[\![\mathit{Reach}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_0} = [\![\mathit{Reach}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_1} = [\![\mathit{Reach}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_2} = 0$, and $[\![\mathit{Next}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_0} = [\![\mathit{Next}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_1} = [\![\mathit{Next}[\mathtt{z},\mathtt{w}]]\!]^{\sigma_2} = 0$. □

$$
\begin{aligned}
\mathit{NextNull}[\mathtt{x}]' \iff & x' \neq \mathit{null} \wedge n'(x') = \mathit{null} \\
& \text{(Unfolding the defining formula of } \mathit{NextNull}[\mathtt{x}]'\text{)} \\
& n(y) \neq \mathit{null} \wedge n(n(y)) = \mathit{null} \\
& \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
& \mathit{NextNextNull}[\mathtt{y}] \\
& \text{(Folding)}
\end{aligned}
$$

Assume that $z \in PVar \setminus \{x, y\}$.

$$
\begin{aligned}
Next[\mathtt{x},\mathtt{z}]' &\iff x' \neq null \wedge z' \neq null \wedge n'(x') = z' \\
&\qquad \text{(Unfolding the defining formula of } Next[\mathtt{x},\mathtt{z}]'\text{)} \\
&\iff x \neq null \wedge z \neq null \wedge n(n(y)) = z \\
&\qquad \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
&\iff 0 \\
&\qquad \text{(Since } y \neq null z \neq null \wedge n(n(y)) = z \implies Reach[\mathtt{y},\mathtt{z}] \implies \\
&\qquad \neg ReachNull[\mathtt{y}]\text{, yet we have that } ReachNull[\mathtt{y}] \text{ holds} \\
&\qquad \text{in the condition.)}
\end{aligned}
$$

Case 4 : the condition *NotNull*[y] $\wedge$ $\neg$*NextNull*[y] $\wedge$ $\neg$*ReachNull*[y] $\wedge$ $\neg$*NextPtByVar*[y] holds.

Lemma 11. *Under the condition of case 4, the predicate NextNextPtByVar[y] is a focus predicate.*

In other words, let σ_0 be a concrete state that satisfies the condition formula. Then, there exist two concrete states (containing only acyclic unshared lists) σ_1, σ_2 such that $\sigma_1 \models NextNextNull[y]$ and $\sigma_2 \models \neg NextNextNull[y]$, and $\alpha(\sigma_0) = \alpha(\sigma_1) = \alpha(\sigma_2)$.

Proof. We show that we can modify σ_0 to obtain two concrete states, σ_1 and σ_2, such that $\alpha(\sigma_0) = \alpha(\sigma_1) = \alpha(\sigma_2)$, and $\sigma_1 \models NextNextPtByVar[\mathtt{y}]$ and $\sigma_2 \models \neg NextNextPtByVar[\mathtt{y}]$.

From the fact that σ_0 satisfies the condition, we have that σ_0 contains a path $u_1, \ldots, u_k$ such that $u_1 = y$, u_k is pointed-to by some variable (since $\neg ReachNull[\mathtt{y}]$ holds), and $u_2, \ldots, u_{k-1}$ are not pointed-to by any variable, and $k > 1$ (since $\neg NextPtByVar[\mathtt{y}]$ holds).

To obtain σ_1, we replace the path $u_1, \ldots, u_k$ with the path u_1, u_2, u_k, by removing the elements $u_3, \ldots, u_{k-1}$ from the heap and adjusting the `n` links accordingly. Thus, $n(n(y)) = u_k$ in σ_1, i.e., $\sigma_1 \models NextNextPtByVar[\mathtt{y}]$.

To obtain σ_2, we replace the path $u_1, \ldots, u_k$ with the path u_1, u', u_2, u_k where u' is a new element added to the heap between u_1 and u_2. Since neither of u' and u_2 are pointed-to by a variable, $\sigma_2 \models \neg NextNextPtByVar[\mathtt{y}]$.

Intuitively, to obtain σ_1 and σ_2, we have changed σ_0 by modifying the length of the path from y to u_k, which used to be greater than 1. However, the abstraction we use does not distinguish between lengths of lists greater than 1, and thus we get 3 concrete states that are equivalent under the abstraction.

Now, since in σ_0, none of the elements $u_2, \ldots, u_{k-1}$ were are pointed-to by variables, every logical constant z representing a variable `z`, evaluates to the same element in σ_0, σ_1, and σ_2. Therefore, $[\![NotNull[\mathtt{z}]]\!]^{\sigma_0} = [\![NotNull[\mathtt{z}]]\!]^{\sigma_1} = [\![NotNull[\mathtt{z}]]\!]^{\sigma_2}$, for every $z \in PVar$. As well, $[\![Aliased[\mathtt{z},\mathtt{w}]]\!]^{\sigma_0} = [\![Aliased[\mathtt{z},\mathtt{w}]]\!]^{\sigma_1} = [\![Aliased[\mathtt{z},\mathtt{w}]]\!]^{\sigma_2}$ for every $z, w \in PVar$.

Let z be in $PVar$ such that $z = null$ in σ_0. Then, $z = null$ in σ_1 and σ_2. Therefore, $[\![NextNull[\mathtt{z}]]\!]^{\sigma_0} = [\![NextNull[\mathtt{z}]]\!]^{\sigma_1} = [\![NextNull[\mathtt{z}]]\!]^{\sigma_2} = 0$. This also means that for every $w \in PVar$, we have $[\![Next[\mathtt{z},\mathtt{w}]]\!]^{\sigma_0} = [\![Next[\mathtt{z},\mathtt{w}]]\!]^{\sigma_1} =$

$[\![Next[\mathtt{z},\mathtt{w}]]\!]^{\sigma_2} = 0$, $[\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2} = 0$, $[\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2} = 0$, and $[\![Reach[\mathtt{z},\mathtt{w}]]\!]^{\sigma_0} = [\![Reach[\mathtt{z},\mathtt{w}]]\!]^{\sigma_1} = [\![Reach[\mathtt{z},\mathtt{w}]]\!]^{\sigma_2} = 0$.

We now have to show that the equality holds for predicates of the form $NextNull[\mathtt{z}]$, $Next[\mathtt{z},\mathtt{w}]$, and $Reach[\mathtt{z},\mathtt{w}]$ such that both z and w are different from *null* in σ_0.

The only element pointed-to by a variable for which the outgoing edge has changed is the one pointed-to by `y`. Therefore, for every $z \in PVar \setminus \{y\}$ such that $y \neq z$, $[\![n(z) = null]\!]^{\sigma_0} = [\![n(z) = null]\!]^{\sigma_1} = [\![n(z) = null]\!]^{\sigma_2}$. Hence, $[\![NextNull[\mathtt{z}]]\!]^{\sigma_0} = [\![NextNull[\mathtt{z}]]\!]^{\sigma_1} = [\![NextNull[\mathtt{z}]]\!]^{\sigma_2}$.

For every $z \in PVar \setminus \{y\}$ such that $y = z$, we have $[\![NextNull[\mathtt{z}]]\!]^{\sigma_0} = 0$ (since the statement is preceded by `assert(y!=null)`). In σ_1 and σ_2, we have $n(z) = u_1$. Thus, $[\![NextNull[\mathtt{z}]]\!]^{\sigma_0} = [\![NextNull[\mathtt{z}]]\!]^{\sigma_1} = [\![NextNull[\mathtt{z}]]\!]^{\sigma_2} = 0$.

Let w and z be two variables in $PVar$ such that w and z are distinct and both are different from *null*. If any of w and z are not on the same list as y in σ_0 then $[\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2}$ and $[\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2}$, since only the list containing y is possibly different in these states. Assume that, in σ_0, w and z are on the list containing y in, and without loss of generality let z be reachable from w. If the path between w and z does not contain the path $u_0, \ldots, u_k$ then it remains the same in all three states, and thus $[\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2}$ and $[\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2}$. Assume that, in σ_0, the path from w to z is $v_1, \ldots, v_g, u_1, \ldots, u_k, h_1, \ldots, j_p$ where $k > 1$, $g \geq 1$, and $p \geq 1$. Then, in σ_1 the path from w to z is $v_1, \ldots, v_g, u_1, u_2, u_k, h_1, \ldots, j_p$ and in σ_1 the path from w to z is $v_1, \ldots, v_g, u_1, u', \ldots, u_k, h_1, \ldots, j_p$. Therefore, $[\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![Next[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2} = 0$ and $[\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_0} = [\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_1} = [\![Reach[\mathtt{w},\mathtt{z}]]\!]^{\sigma_2}$. □

Lemma 12. *Under the condition of case* 3, $n(n(y)) \neq null$ *holds:*

Proof. By Contradiction. Assume that the condition of case 3 and $n(n(y)) = null$ hold simultaneously.

Since $n(n(y)) = null$ holds, we have that $n^j(y) = null$ for every $j \geq 2$. Therefore, for every $w \in PVar$, if $w \neq null$ then $n^+(y, w)$ holds when $n(y) = w$ holds.

Now, for every $w \in PVar$, we have

$Reach[\mathtt{y}, \mathtt{w}] \iff$
$y \neq null \land w \neq null \land n^+(y, w) \iff$
(Unfolding the predicate definition)
$y \neq null \land w \neq null \land n(y) = w \iff$
(Since $w \neq null$ and $n^j(y) = null$ for $j \geq 2$)
$Next[\mathtt{y}, \mathtt{w}] \implies$
(Folding)
$\bigvee_{z \in PVar \setminus \{y\}} Next[\mathtt{y}, \mathtt{z}] \iff$
(Weakening)
$NextPtByVar[\mathtt{y}] \iff$
(Folding)
0
(Since, by the condition of case 3, $\neg NextPtByVar[\mathtt{y}]$ holds)

We now have that $\neg Reach[\mathtt{y}, \mathtt{q}]$ holds for every $w \in PVar$, i.e., $\bigwedge_{w \in PVar \setminus y} \neg Reach[\mathtt{y}, \mathtt{w}]$ holds. Therefore, $ReachNull[\mathtt{y}]$ holds. However, by the condition of case 3, $\neg ReachNull[\mathtt{y}]$ also holds. We have arrived to a contradiction. □

$NextNull[\mathtt{x}]' \iff x' \neq null \land n'(x') = null$
(Unfolding the defining formula of $NextNull[\mathtt{x}]'$)
$n(y) \neq null \land n(n(y)) = null$
(Unfolding the concrete semantics of `x=y.n`)
0
(*Lemma* 12)

Lemma 13. *Under the condition of case* 3*, the following holds for every* $z \in PVar \setminus \{y\}$*:*

$$z \neq null \land n(n(y)) = z \iff NextNextPtByVar[y] \land ReachFirst[y, z] \ .$$

Proof. Assume that $z \neq null \land n(n(y)) = z$ holds. This implies that $\bigvee_{w \in PVar \setminus \{y\}} w \neq null \land n(n(y)) = w$ holds. Hence, $NextNextPtByVar[\mathtt{y}]$ holds.

We now show that $ReachFirst[\mathtt{y}, \mathtt{z}]$ also holds. The following calculation shows that $Reach[\mathtt{y}, \mathtt{z}]$ holds.

$z \neq null \land n(n(y)) = z \implies$
$z \neq null \land n^+(y, z) \iff$
$y \neq null \land z \neq null \land n^+(y, z) \iff$
(By the condition of case 4)
$Reach[\mathtt{y}, \mathtt{z}]$
(Folding)

Now assume that $Reach[\mathtt{y}, \mathtt{w}]$ for $w \in PVar \setminus \{y\}$. This means that $w \neq null$ and there exists the unique path $u_1, \ldots, u_k$ such that $u_1 = y$ and $u_k = w$. By the condition of case 4, we have that $\neg NextPtByVar[\mathtt{y}]$ holds. Therefore, $k \geq 2$. Since $z \neq null \land n(n(y)) = z$ also holds and n is deterministic, we have that $u_2 = z$ and therefore, $n^*(z, w)$ holds. This shows that $ReachFirst[\mathtt{y}, \mathtt{z}]$ holds.

Now assume that $\mathit{NextNextPtByVar}[\mathtt{y}] \wedge \mathit{ReachFirst}[\mathtt{y}, \mathtt{z}]$ holds. From the first conjunct, we have that there exists $w \in PVar \backslash \{y\}$ for which $w \neq \mathit{null} \wedge n(n(y)) = w$ holds. Choose such a w without loss of generality.

Since $\mathit{ReachFirst}[\mathtt{y}, \mathtt{z}]$ holds, we have that $n^+(y, z)$ holds, i.e., there exists the unique path $u_1, \ldots, u_k$ such that $u_1 = y$, $u_k = z$, and $k \geq 1$. From the conjunct $\neg \mathit{NextPtByVar}[\mathtt{y}]$ in the condition, we know that $k \geq 2$.

Also, from the fact $\mathit{ReachFirst}[\mathtt{y}, \mathtt{z}]$ and $n^+(y, w)$ hold, we have that $n^*(z, w)$ also holds. However, $w = u_2$, and therefore $z = w$. Substituting w with z, we get that $z \neq \mathit{null} \wedge n(n(y)) = z$ holds. □

Assume that $z \in PVar \setminus \{x, y\}$.

$$
\begin{aligned}
\mathit{Next}[\mathtt{x}, \mathtt{z}]' \iff & x' \neq \mathit{null} \wedge z' \neq \mathit{null} \wedge n'(x') = z' \\
& \text{(Unfolding the defining formula of } \mathit{Next}[\mathtt{x}, \mathtt{z}]'\text{)} \\
\iff & x \neq \mathit{null} \wedge z \neq \mathit{null} \wedge n(n(y)) = z \\
& \text{(Unfolding the concrete semantics of } \mathtt{x=y.n}\text{)} \\
& \mathit{NextNextPtByVar}[\mathtt{y}] \wedge \mathit{ReachFirst}[\mathtt{y}, \mathtt{z}] \\
& (\mathit{Lemma}\ 13)
\end{aligned}
$$