Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists

Tel Aviv University, School of Computer Science, TR-2005-01-191212

R. Manevich[1,⋆], E. Yahav[2], G. Ramalingam[2], and M. Sagiv[1]

[1] Tel Aviv University, {rumster,msagiv}@tau.ac.il
[2] IBM T.J. Watson Research Center, {rama,eyahav}@watson.ibm.com

Abstract. Predicate abstraction and canonical abstraction are two finitary abstractions used to prove properties of programs. We study the relationship between these two abstractions by considering a very limited case: abstraction of (potentially cyclic) singly-linked lists.

We provide a new and rather precise family of abstractions for potentially cyclic singly-linked lists. The main observation behind this family of abstractions is that the number of shared nodes in linked lists can be statically bounded. Therefore, the number of possible "heap shapes" is also bounded. We present the new abstraction in both predicate abstraction form as well as in canonical abstraction form.

As we illustrate in the paper, given any canonical abstraction, it is possible to define a predicate abstraction that is equivalent to the canonical abstraction. However, with this straightforward simulation, the number of predicates used for the predicate abstraction is exponential in the number of predicates used by the canonical abstraction.

An important feature of the family of abstractions we present in this paper is that the predicate abstraction representation we define is far more practical as it uses a number of predicates that is quadratic in the number of predicates used by the corresponding canonical abstraction representation. In particular, for the most abstract abstraction in this family, the number of predicates used by the canonical abstraction is linear in the number of program variables, while the number of predicates used by the predicate abstraction is quadratic in the number of program variables.

We have encoded this particular predicate abstraction and corresponding transformers in TVLA, and used this implementation to successfully verify safety properties of several list manipulating programs, including programs that were not previously verified using predicate abstraction or canonical abstraction.

1 Introduction

Abstraction and abstract interpretation [7] are essential techniques for automatically proving properties of programs. The main challenge in abstract interpretation is to develop abstractions that are precise enough to prove the required property and efficient enough to be applicable to realistic applications.

⋆ This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No 304/03).

Predicate abstraction [11] abstracts the program into a Boolean program which conservatively simulates all potential executions. Every safety property which holds for the Boolean program is guaranteed to hold for the original program. Furthermore, abstraction refinement [6, 2] can be used to refine the abstraction when the analysis produces a "false alarm". When the process terminates, it yields a concrete error trace in which the property is violated, or successfully verifies the property. In principle, the whole process can be fully mechanized given a sufficiently powerful theorem prover. This process was successfully used in SLAM [19] and BLAST [12] to prove safety properties of device drivers.

Canonical abstraction [23] is a finitary abstraction that was specially developed to model properties of unbounded memory locations (inspired by [16]). This abstraction has been implemented in TVLA [17], and successfully used to prove various properties of heap-manipulating programs (e.g., [21, 25, 24]).

1.1 Main Results

In this paper, we study the utility of predicate abstraction to prove properties of programs operating on singly-linked lists. We also compare the expressive power of predicate abstraction and canonical abstraction.

The results in this paper can be summarized as follows:

- We show that current state-of-the-art iterative refinement techniques fail to prove interesting properties of singly-linked lists such as pointer equalities and absence of null dereferences in a fully automatic manner. This means that on many simple programs the process of refinement will diverge when the program is correct. This result is inline with the experience of Blanchet et al. [4].
- We show that predicate abstraction can simulate arbitrary finitary abstractions and, in particular, canonical abstraction. This trivial result is not immediately useful because of the number of predicates used. The number of predicates required to simulate canonical abstraction is, in the worst case, exponential in the number of predicates used by the canonical abstraction (usually, this means exponential in the number of program variables).
- We develop a new family of abstractions for heaps containing (potentially cyclic) singly-linked lists. The main idea is to summarize list elements on unshared list segments not pointed-to by local variables. For programs manipulating singly-linked lists, this abstraction is finitary since the number of shared list elements reachable from program variables is bounded. Abstractions in this family vary in their level of precision, which is controlled by the level of sharing-relationships recorded.
- We show that the abstraction recording only one-level sharing relationships (i.e., the least precise member of the family that records sharing) is sufficient for successfully verifying all our example programs, including programs that were not verified earlier using predicate abstraction or canonical abstraction.
- We show how to code the one-level-sharing abstraction using both canonical abstraction (with a linear number of unary predicates) and predicate abstraction (with a quadratic number of nullary predicates).

```
      //head points to the first element of an acyclic list
      //tail points to the last element of the same list
1     curr = head;
2     while (curr != tail) {
3        assert (curr != null);
4        curr = curr.n;
5     }
```

Fig. 1. A simple program on which counterexample-guided refinement diverges

1.2 Motivating Examples

Fig. 1 shows a program that traverses a singly-linked list with a head-pointer `head` and a tail-pointer `tail`. This is a trivial program since it only uses an acyclic linked list, and does not contain destructive pointer updates. When counterexample-guided iterative refinement is applied to this program to assure that the assertion at line 3 is never violated, it will diverge. At the i-th iteration it will generate an assertion of the form $\texttt{curr}(\texttt{.n})^i! = \texttt{null}$. However, no finite value of i will suffice. Indeed, the problem of proving the absence of null-dereferences is undecidable even in programs manipulating singly-linked lists and even under the (non-realistic) assumption that all control flow paths are executable [5].

In contrast, the TVLA abstract interpreter [17] proves the absence of null dereferences in this program in 2 seconds, consuming 0.6MB of memory. TVLA uses canonical abstraction which generalizes predicate abstraction by allowing first-order predicates (relation symbols) that can have arguments. Thus, nullary (0-arity) predicates correspond to predicates in the program and in predicate abstractions. Unary predicates (1-arity) are used to denote sets of unbounded locations and binary (2-arity) predicates are used to denote relationships between unbounded locations.

A curious reader may ask herself: *Are there program properties that can be verified with canonical abstractions but not with predicate abstractions?*

It is not hard to see that the answer is negative, since any finitary abstraction can be simulated by a suitable predicate abstraction. For example, consider an abstraction mapping $\alpha : C \to A$, from a concrete domain C to a finite abstract domain of indexed elements $A = \{1, \ldots, n\}$. Define the predicate BIT$[j]$ to hold for the set of concrete states $\{c \mid \text{the } j\text{th bit of } \alpha(c), \text{ in its binary representation, is } 1\}$. Now, the set of predicates $\{\text{BIT}[j]\}_{j=1}^{\lceil \log n \rceil}$ yields a predicate abstraction that simulates A. This simulation is usually not realistic, since it contains too many predicates. The number of predicates required by predicate abstraction to simulate canonical abstraction can be exponential in the number of predicates used by the canonical abstraction.

Fortunately, the only nullary predicate crucial to prove the absence of null dereferences in this program is the fact that `tail` is reachable from `curr` by a path of `n` selectors (of some length). Similar observations were suggested independently in [15, 3, 14]. In this paper, we define a quadratic set of nullary predicates that captures the invariants in many programs manipulating (potentially cyclic) singly-linked lists.

Fig. 2 shows a simple program removing a contiguous segment from a cyclic singly-linked list pointed-to by `x`. For this example program, we would like to verify that the resulting structure pointed-to by `x` remains a cyclic singly-linked list. Unfortunately, using TVLA's canonical abstraction with the standard set of predicates turns out to

```
     // x points to a cyclic singly-linked list
     // low and high are two integer values, low < high
1    t = null;
2    y = x;
3    while (t != x && y.data < low) {
4       t = y.n; y = t;
5    }
6    z = y;
7    while (z != x && z.data < high) {
8       t = z.n; z = t;
9    }
10   t = null;
11   if (y != z) {
12      y.n = null;
13      y.n = z;
14   }
```

Fig. 2. A simple program that removes the segment between low and high from a linked list

be insufficient. The problem stems from the fact that canonical abstraction with the standard set of predicates loses the ordering between the 3 reference variables that point to that cyclic singly-linked list (this is further explained in the next section).

In this paper, we provide two abstractions — a predicate abstraction, and a canonical abstraction — that are able to correctly determine that the result of this program is indeed a cyclic singly-linked list.

The rest of this paper is organized as follows: Sec. 2 provides background on the basic concrete semantics we are using, canonical abstraction, and predicate abstraction. Sec. 3 presents an instrumented concrete semantics that records list interruptions. Sec. 4 shows a quite precise predicate abstraction for singly-linked lists. Sec. 5 shows a quite precise canonical abstraction of singly-linked lists. In Sec. 6, we show that the predicate abstraction of Sec. 4 and the canonical abstraction of Sec. 5 are equivalent. Sec. 7 describes our experimental results. Proofs of claims and additional technical details can be found in the respective appendices.

2 Background

In this section, we provide basic definitions that we will use throughout the paper. In particular, we define canonical abstraction and predicate abstraction.

2.1 Concrete Program States

We represent the state of a program using a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects.

Definition 1. *A 2-valued logical structure over a vocabulary (set of predicates) $\mathcal{P}$ is a pair $S = \langle U, \iota \rangle$ where U is the universe of the 2-valued structure, and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota(p) : U^k \to \{0, 1\}$.*

We denote the set of all 2-valued logical structures over a set of predicates $\mathcal{P}$ by 2-STRUCT$_\mathcal{P}$. In the sequel, we assume that the vocabulary $\mathcal{P}$ is fixed, and abbreviate 2-STRUCT$_\mathcal{P}$ to 2-STRUCT.

Table 1. Predicates used for representing concrete program states

Predicates	**Intended Meaning**
$eq(v_1, v_2)$	v_1 is equal to v_2
$\{\, x(v) : x \in \mathit{PVar} \,\}$	reference variable x points to the object v
$n(v_1, v_2)$	next field of the object v_1 points to the object v_2

Table 1 shows the predicates we use to record properties of individuals. A unary predicate $x(v)$ holds when the object v is pointed-to by the reference variable x. We assume that the set of predicates includes a unary predicate for every reference variable in a program. We use *PVar* to denote the set of all reference variables in a program. A binary predicate $n(v_1, v_2)$ records the value of the reference field n.

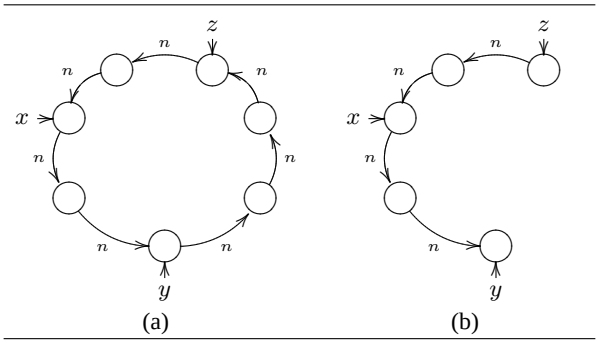

Fig. 3. The effect of the statement `y.n=null` in the concrete semantics. (a) a possible state of the program of Fig. 2 at line 12; (b) the result of applying `y.n=null` to (a)

Concrete Semantics Program statements are modelled by *actions* that specify how statements transform an incoming logical structure into an outgoing logical structure. This is done primarily by defining the values of the predicates in the outgoing structure using formulae of first-order logic with transitive closure over the incoming structure [23]. The update formulae for heap-manipulating statements are shown in Table 2. For brevity, we omit the treatment of the allocation statement `new T()`, the interested reader may find the details in [23].

To simplify update formulae, we assume that every assignment to the n field of an object is preceded by first assigning null to it. Therefore, the statement at line 12 of the example program of Fig. 2 assigns null to `y.n` before the next statement assigns it the new value z.

Example 1. Applying the action `y.n = null` to the concrete structure of Fig. 3(a), results with the concrete structure of Fig. 3(b). Throughout this paper we assume that all heaps are garbage-free, i.e., every element is reachable from some program variable, and that the concrete program semantics reclaims garbage elements immediately after

Table 2. Predicate-update formulae that define the semantics of heap-manipulating statements

Statement	**Update formulae**
`x = null`	$x'(v) = 0$
`x = t`	$x'(v) = t(v)$
`x = t.n`	$x'(v) = \exists v_1 : t(v_1) \wedge n(v_1, v)$
`x.n = null`	$n'(v_1, v_2) = n(v_1, v_2) \wedge \neg x(v_1)$
`x.n = t` (assuming x.n == null)	$n'(v_1, v_2) = n(v_1, v_2) \vee (x(v_1) \wedge t(v_2))$

executing program statements. Thus, the two objects between `y` and `z` are collected when `y.n` is set to null, as they become unreachable.

2.2 Canonical Abstraction

The goal of an abstraction is to create a finite representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size. The abstractions we use are based on 3-valued logic [23], which extends boolean logic by introducing a third value $1/2$ denoting values that may be 0 or 1.

We represent an abstract state of a program using a 3-valued first-order structure.

Definition 2. *A 3-valued logical structure over a set of predicates $\mathcal{P}$ is a pair $S = \langle U, \iota \rangle$ where U is the universe of the 3-valued structure (an individual in U may represent multiple heap-allocated objects), and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota(p) : U^k \to \{0, 1, 1/2\}$.*

An abstract state may include summary nodes*, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state. A summary node u has $eq(u, u) = 1/2$, indicating that it may represent more than a single individual.*

Embedding We now formally define how states are represented using abstract states. The idea is that each individual from the (concrete) state is mapped into an individual in the abstract state. More generally, it is possible to map individuals from an abstract state into an individual in another, less precise, abstract state.

Formally, let $S = \langle U, \iota \rangle$ and $S' = \langle U', \iota' \rangle$ be abstract states. A function $f : U \to U'$ such that f is surjective is said to *embed S into S'* if for each predicate p of arity k, and for each $u_1, \ldots, u_k \in U$, one of the following holds:

$$\iota(p(u_1, \ldots, u_k)) = \iota'(p(f(u_1), \ldots, f(u_k))) \quad \text{or} \quad \iota'(p(f(u_1), \ldots, f(u_k))) = 1/2$$

We say that S' *represents* S when there exists such an embedding f.

One way of creating an embedding function f is by using *canonical abstraction*. Canonical abstraction maps concrete individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by f to the same abstract individual.

Table 3. Predicates used for the canonical abstraction in Fig. 4, and their meaning.The notation n^* stands for the reflexive-transitive closure of the predicate n, and n^+ stands for the transitive closure of n

Predicates	**Intended Meaning**	**Defining formulae**
$\{\, x(v) : x \in \mathit{PVar} \,\}$	reference variable x points to v	
$n(u, v)$	next field of u points to v	
$\{\, r_x(v) : x \in \mathit{PVar} \,\}$	v is reachable from x by dereferencing n fields	$\exists v_x . x(v_x) \wedge n^*(v_x, v)$
$c_n(v)$	v resides on a cycle of n fields	$n^+(v, v)$
$is(v)$	v is heap-shared	$\exists v_1, v_2 . n(v_1, v) \wedge n(v_2, v) \wedge (v_1 \neq v_2)$

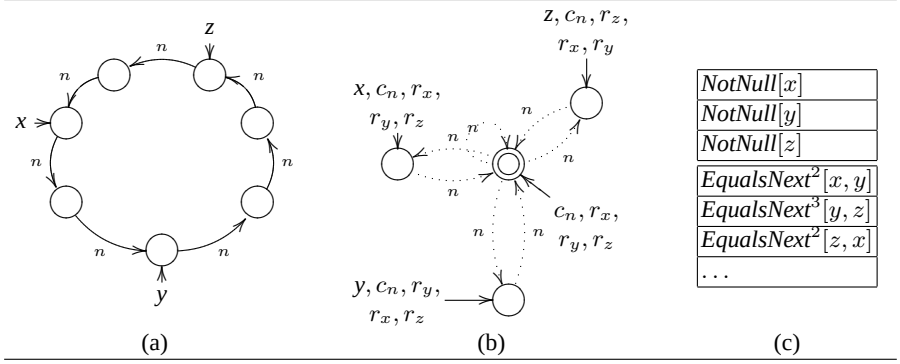

Fig. 4. (a) a concrete possible state of the program of Fig. 2 at line 12, (b) its canonical abstraction in TVLA, (c) its predicate abstraction with the set of predicates in Table 4

Table 3 presents the set of predicates used in [23] to abstract singly-linked lists. The predicates $r_x(v)$, $c_n(v)$, and $is(v)$, referred to in [23] as *instrumentation predicates*, record derived information and are used to refine the abstraction.

This set of predicates has been used for successfully verifying many programs manipulating singly-linked lists, but is insufficient for verifying that the output of the example program of Fig. 2 is a cyclic singly-linked list pointed-to by x.

Example 2. Fig. 4(b) shows the canonical abstraction of the concrete state of Fig. 4(a), using the predicates of Table 3. The node with double-line boundaries is a *summary node*, possibly representing more than a single concrete node. The dashed edges are $1/2$ edges, a dashed edge exists between v_1 and v_2 when $n(v_1, v_2) = 1/2$. The abstract state of Fig. 4(b) records the fact that x,y, and z point to a cyclic list (using the $c_n(v)$ predicate), and that all list elements are reachable from all 3 reference variables (using the $r_x(v)$,$r_y(v)$, and $r_z(v)$ predicates). This abstract state, however, does not record the order between the reference variables. In particular, it does not record that x does not reside between y and z (the segment that is about to be removed by the program statement at line 12). As a result, applying the abstract effect of y.n=z to this abstract state results with a possible abstract state in which the cyclic list is broken.

2.3 Predicate Abstraction

Predicate abstraction abstracts a concrete state into a truth-assignment for a finite set of propositional (nullary) predicates.

A predicate abstraction is defined by a vocabulary $P^A = \{P_1, \ldots, P_m\}$, where each P_i is associated with a defining formula φ_i that can be evaluated over concrete states. An abstract state is a truth assignment to the predicates in P^A. Given an abstract state A, we denote the value of P_i in A by A_i.

A concrete state S over a vocabulary P^C, is mapped to an abstract state A by an abstraction mapping β: 2-STRUCT$[\mathcal{P}^C] \rightarrow$ 2-STRUCT$[\mathcal{P}^A]$. The abstraction mapping evaluates the defining formulae of the predicates in $\mathcal{P}^A$ over S and sets the appropriate values to the respective predicates in A. Formally, for every $1 \leq i \leq m$, $A_i = [\![\varphi_i]\!]^S_2$.

Table 4. Predicates used for the predicate abstraction in Fig. 4, and their meaning. Note that the maximal tracked length K is fixed a priori

Predicates	**Intended meaning**	**Defining formulae**
$\{$ *NotNull*$[x]: x \in$ *PVar* $\}$	x is not null	$\exists v_x.x(v_x)$
$\{$ *EqualsNext*$^k[x, y]$ $: x, y \in$ *PVar*, $0 \leq k \leq K\}$	the node pointed-to by y is reachable by k n fields from the node pointed-to by x	$\exists v_0, \ldots, v_k.x(v_0) \wedge y(v_k) \wedge \bigwedge_{0 \leq i < k} n(v_i, v_{i+1})$

Table 4 shows an example set of predicates similar to the ones used in [1, 8].

Example 3. Fig. 4(c) shows the predicate abstraction of the concrete state shown in Fig. 4(a) using the predicates of Table 4. A predicate of the form *NotNull*$[x]$ records the fact that x is not null. In Fig. 4(c), all three variables x,y,and z are not null. A predicate of the form *EqualsNext*$^k[x, y]$ records that the node pointed-to by y is reachable by k steps over the n fields from the node pointed-to by x (Note that K, the maximal tracked length, is fixed a priori). For example, in Fig. 4(c), the list element pointed-to by y is reachable from the list element pointed-to by x in 2 steps over the n field, and therefore *EqualsNext*$^2[x, y]$ holds.

3 Recording List Interruptions

In this section, we instrument the concrete semantics to record a designated set of nodes, called *interruptions*, in singly-linked lists. The instrumented concrete semantics presented in this section serves as the basis for the predicate abstraction and the canonical abstraction presented in the following sections.

3.1 The Intuition

The intuition behind our instrumented concrete is that a garbage-free heap, containing only singly-linked lists, is characterized by two factors: (i) the "shape" of the heap, i.e., the connectivity relations between a set of designated nodes (interruptions); and (ii) the length of "simple" list segments connecting interruptions, but not containing interruptions themselves. This intuition is similar to proofs of small model properties.

Considering this characterization, we observe that the number of shapes that are equivalent, up to lengths of simple list segments, is bounded. We therefore instrument our concrete semantics to record interruptions, which are an essential ingredient of the sharing patterns.

The abstractions presented in the next sections, abstract the lengths of simple list segments into a fixed set of abstract lengths (thereby obtaining a finite representation). These abstractions retain the general shape of the heap but lose any correlations between the actual lengths of different simple list segments. Our experience indicates that the correctness of program properties usually depends on the shape of heap, rather than on the lengths of simple list segments.

In the rest of this section, we formally define the notions of interruptions and simple list segments, and formally define the information recorded by our instrumented concrete semantics.

3.2 Basic Definitions

We say that a list node v is an *interrupting node*, or simply an *interruption*, if it is pointed-to by a program variable or it is heap-shared. Fig. 5 shows a heap with 4 interruptions: (i) the node pointed-to by *x*, (ii) the node pointed-to by *y*, (iii) the node pointed-to by $x_{s,1}$ and $y_{s,1}$, and (iv) the node pointed-to by $x_{s,2}$ and $y_{s,2}$.

Definition 3 (Uninterrupted Lists). *We say that there is an* uninterrupted list *between list node u and list node v, denoted by UList(u, v), when there is a non-empty path between them, such that, every node on the path between them (i.e., not including u and v) is non-interrupting.*

We also say that there is an uninterrupted list between list node v and null, denoted by UListNULL(v), when there is a non-empty path from v to null, such that, every node on the path, except possibly v, is non-interrupting.

Table 5 formulates UList(u, v) and UListNULL(v) as formulae in FO^{TC}.

Given a heap, we are actually interested in a subset of its uninterrupted lists. We say that an uninterrupted list is *maximal* when it is not contained in a longer uninterrupted list.

The heap in Fig. 5 contains 4 maximal uninterrupted lists: (i) from the node pointed-to by *x* and the node pointed-to by $x_{s,1}$ and $y_{s,1}$, (ii) from the node pointed-to by *y* and the node pointed-to by $x_{s,1}$ and $y_{s,1}$, (iii) from the node pointed-to by $x_{s,1}$ and $y_{s,1}$ to the node pointed-to by $x_{s,2}$ and $y_{s,2}$, and (iv) from the node pointed-to by $x_{s,2}$ and $y_{s,2}$ to itself.

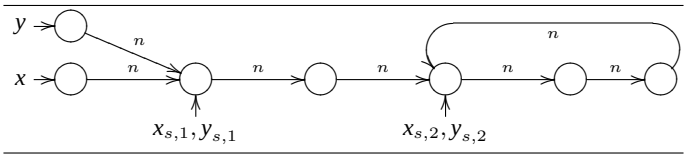

Fig. 5. Two lists sharing the same tail, and their representation in the instrumented concrete semantics

Table 5. Shorthand notations used throughout this paper

Shorthand	**Meaning**	**Formula**
HeapShared(v)	v is heap-shared	$\exists a, b. n(a, v) \wedge n(b, v) \wedge (a \neq b)$
PtByVar(v)	v is pointed-to by some variable	$\bigvee_{var \in PVar} var(v)$
Interruption(v)	v is an interrupting list node	*HeapShared*$(v) \vee$ *PtByVar*(v)
UList$_1(u, v)$	there is an uninterrupted list of length 1 from u to v	$n(u, v)$
UList$_2(u, v)$	there is an uninterrupted list of length 2 from u to v	$\exists m. \neg$*Interruption*$(m) \wedge$ $n(u, m) \wedge n(m, v)$
UList$_{>2}(u, v)$	there is an uninterrupted list of length > 2 from u to v	$\exists m_1, m_2 : n(u, m_1) \wedge n(m_2, v) \wedge$ (TC $a, b : n(a, b) \wedge \neg$*Interruption*$(a) \wedge$ $\neg$*Interruption*$(b))(m_1, m_2)$
UList(u, v)	there is an uninterrupted list of some length from u to v	*UList*$_1(u, v) \vee$ *UList*$_2(u, v) \vee$ *UList*$_{>2}(u, v)$
UListNULL$_1(v)$	there is an uninterrupted list of length 1 from v to null	$\forall w. \neg n(v, w)$
UListNULL$_2(v)$	there is an uninterrupted list of length 2 from v to null	$\exists m. n(v, m) \wedge \neg$*Interruption*$(m) \wedge$ *UListNULL*$_1(m)$
UListNULL$_{>2}(v)$	there is an uninterrupted list of length > 2 from v to null	$\exists m_1, m_2 : n(v, m_1) \wedge$ *UListNULL*$_1(m_2)$ (TC $a, b : n(a, b) \wedge \neg$*Interruption*$(a) \wedge$ $\neg$*Interruption*$(b))(m_1, m_2)$
UListNULL(v)	there is a list of some length from v to null	*UListNULL*$_1(v) \vee$ *UListNULL*$_2(v) \vee$ *UListNULL*$_{>2}(v)$

3.3 Statically Naming Heap-Shared Nodes

We now explain how to use a quadratic number of auxiliary variables to statically name all heap-shared nodes. This will allow us to name all maximal uninterrupted lists using nullary predicates for the predicate abstraction, and using unary predicates for the canonical abstraction.

Proposition 1. *A garbage-free heap, consisting of only singly-linked lists with n program variables, contains at most n heap-shared nodes and at most $2n$ interruptions.*

Proof. See Appendix B.

Corollary 1. *In a garbage-free heap, consisting of only singly-linked lists with n program variables, list node v is reachable from list node u if and only if it is reachable by a sequence of $k < n$ uninterrupted lists. Similarly, there is a path from node v to null if and only if there is a path from v to null by a sequence of $k < n$ uninterrupted lists.*

Proof. By Proposition 1, every simple path (from u to v or from v to null) contains at most n interruptions, and, therefore, at most n maximal uninterrupted lists. □

For every program variable *x*, we define a set of auxiliary variables $\{x_{s,k} | k = 1 \ldots n-1\}$. Auxiliary variable $x_{s,k}$ points to a heap-shared node u when there exists a

simple path consisting of k maximal uninterrupted lists from the node pointed by x-to to u, such that all of the interrupting nodes on the path are not pointed-to by program variables (i.e., they are heap-shared). Formally, we define the set of auxiliary variables derived for program variable *x* by using the following set of formulae in FO^{TC}.

$$
\begin{aligned}
x_{s,1}(v) \equiv \quad & \exists v_x.x(v_x) \wedge \mathit{UList}(v_x, v) \wedge \mathit{HeapShared}(v) \wedge \neg \mathit{PtByVar}(v), \\
& \ldots \\
x_{s,k+1}(v) \equiv \; & \exists v_k.x_{s,k}(v_k) \wedge \mathit{UList}(v_k, v) \wedge \mathit{HeapShared}(v) \wedge \\
& \neg \mathit{PtByVar}(v) \wedge \neg(\textstyle\bigvee_{m=1\ldots k} x_{s,m}(v)) \quad .
\end{aligned}
$$

We denote the set of auxiliary variables by *AuxVar* and the set of all (program and auxiliary) variables by $\mathit{Var} = \mathit{PVar} \cup \mathit{AuxVar}$.

Proposition 2. *Every heap-shared node is pointed-to by a variable in Var. Also, $x_{s,k}(v)$ holds for at most one node, for every reference variable x and every index k.*

Proof. See Appendix B.

3.4 Parameterizing the Concrete Semantics

Let n denote the number of (regular) program variables. Notice that $|AuxVar| = O(n^2)$. In the following sections, we will see that using the full set of auxiliary variables yields a canonical abstraction with a quadratic ($O(n^2)$) number of unary predicates, and a predicate abstraction with a bi-quadratic ($O(n^4)$) number of predicates.

We use a parameter k to define different subsets of *Var* as follows: $\mathit{Var}_k = \mathit{PVar} \cup \{x_{s,i}(v) | x \in \mathit{PVar}, i \leq k\}$. By varying the "heap-shared depth" parameter k, we are able to distinguish between different sets of heap-shared nodes. We discovered that, in practice, heap-shared nodes with depth > 1 rarely exist (they never appear in our examples), and, therefore, restricting k to 1 is usually enough to capture all maximal uninterrupted lists. Using Var_1 as the set of variables to record, we obtain a canonical abstraction with a linear number of unary predicates $(O(n))$ and a predicate abstraction with a quadratic $(O(n^2))$ number of variables.

Fig. 5 shows a heap containing a heap-shared node of depth 2 (pointed by $x_{s,2}$ and $y_{s,2}$). By setting the heap-shared depth parameter k to 1, we are able to record the following facts about this heap: (i) there is a list of length 1 from the node pointed-to by *x* to a heap-shared node, (ii) there is a list of length 1 from the node pointed-to by *y* to a heap-shared node, (iii) the heap-shared node mentioned in (i) and (ii) is the same (we record aliasing between variables), and (iv) there is a partially cyclic list (i.e., a non-cyclic list connected to a cyclic list) from the heap-shared node mentioned in (iii). We know that the list from the first heap-shared node does not reach null (since we record lists from interruptions to null) and it is not a cycle from the first-heap shared node to itself (otherwise there would be no second heap-shared node and the cycle would be recorded). The information lost, due to the fact that $x_{s,2}$ and $y_{s,2}$ are not recorded, is that the list from the first heap-shared node to second has length 2 and the cycle from the second heap-shard node to itself is also of length 2.

The Instrumented Concrete Semantics. The instrumented concrete semantics operates by using the update formulae presented in Table 2 and then using the defining formulae of the auxiliary variables to update their values.

4 A Predicate Abstraction for Singly-Linked Lists

We now describe the abstraction used to create a finite (bounded) representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size.

4.1 The Abstraction

We start by defining a vocabulary P^A of nullary predicates, which we use in our abstraction. The predicates are shown in Table 6.

Table 6. Predicates used for the predicate abstraction and their meaning

Predicates	**Defining formulae and intended meaning**
$\{$ *Aliased*$[x, y] : x, y \in$ *Var* $\}$	$\exists v : x(v) \wedge y(v)$ variables x and y point to the same object
$\{$ *UList*$_1[x, y] : x, y \in$ *Var* $\}$	$\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)$ the n field of the object pointed-to by x and the variable y point to the same object
$\{$ *UList*$_2[x, y] : x, y \in$ *Var* $\}$	$\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge$ *UList*$_2(v_x, v_y)$ there is an uninterrupted list of length 2 from the object pointed-to by x to the object pointed-to by y
$\{$ *UList*$[x, y] : x, y \in$ *Var* $\}$	$\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge$ *UList*(v_x, v_y) there is an uninterrupted list of length 1 or more from the object pointed-to by x to the object pointed-to by y
$\{$ *UList*$_1[x,$ null$] : x \in$ *Var* $\}$	$\exists v_x : x(v_x) \wedge$ *UListNULL*$_1(v_x)$ there n field of the object pointed-to by x points to null
$\{$ *UList*$_2[x,$ null$] : x \in$ *Var* $\}$	$\exists v_x . x(v_x) \wedge$ *UListNULL*$_2(v_x)$ there is an uninterrupted list of length 2 from the object pointed-to by x to null
$\{$ *UList*$[x,$ null$] : x \in$ *Var* $\}$	$\exists v_x . x(v_x) \wedge$ *UListNULL*(v_x) there is an uninterrupted list of length 1 or more from the object pointed-to by x to null

Intuitively, the heap is partitioned into a linear number of uninterrupted list segments and each list segment is delimited by some variables. The predicates in Table 6 abstract the path length of list segments into one of the following abstract lengths: 0 (via the *Aliased*$[x, y]$ predicates), 1 (via the *UList*$_1[x, y]$ predicates), 2 (via the *UList*$_2[x, y]$ predicates), or any length ≥ 1 (via the *UList*$[x, y]$ predicates), and infinity (i.e., there is no uninterrupted path and thus all of the previously mentioned predicates are 0).

The abstraction function $\beta_{\mathit{PredAbs}} : 2\text{-STRUCT}[P^C] \rightarrow 2\text{-STRUCT}[P^A]$ operates as described Sec. 2.3 where P^A is the set of predicates in Table 6.

Example 4. Fig. 6(a) shows an abstract state abstracting the concrete state of Fig. 3(a). The predicates *Aliased*$[x, x]$,*Aliased*$[y, y]$, *Aliased*$[z, z]$ represent the fact that the reference variables x, y, and z are not null. The predicate *UList*$_2[x, y]$ represents the fact

$Aliased[x,x], Aliased[y,y], Aliased[z,z]$
$UList_2[x,y], UList_2[z,x]$
$UList[x,y], UList[y,z], UList[z,x]$

(a)

$Aliased[x,x], Aliased[y,y], Aliased[z,z]$
$UList_1[y, \text{null}]$
$UList_2[x,y], UList_2[z,x]$
$UList[x,y], UList[z,x], UList[y, \text{null}]$

(b)

Fig. 6. The abstract effect of `y.n=null` under predicate abstraction. (a) predicate abstraction of the state of Fig. 3(a); (b) result of applying the abstract transformer of `y.n=null` to (a)

that there is an uninterrupted list of length exactly 2 from the object pointed-to by `x` to the object pointed-to by `y`. This adds on the information recorded by the predicate $UList[x,y]$, which represents the existence of a list of length 1 or more. Similarly, the predicate $UList_2[z,x]$ records the fact that a list of exactly length 2 exists from `z` to `x`. Note that the uninterrupted list between `y` and `z` is of length 3, a length that is abstracted away and recorded as a uninterrupted list of an arbitrary length by $UList[y,z]$.

4.2 Abstract Semantics

Rabin [20] showed that monadic second-order logic of theories with one function symbol is decidable. This immediately implies that first-order logic with transitive closure of singly-linked lists is decidable, and thus the best transformer can be computed as suggested in [22]. Moreover, Rabin also proved that every satisfiable formula has a small model of limited size, which can be employed by the abstraction. For simplicity and efficiency, we directly define the abstractions and the abstract transformer. The reader is referred to [13] which shows that reasonable extensions of this logic become undecidable. We believe that our techniques can be employed even for undecidable logics but the precision may vary. In particular, the transformer we provide here operates in polynomial time.

Example 5. In order to simplify the definition of the transformer for `y.n = null`, we split it to 5 different cases (shown in [18]) based on classification of the next list interruption. The abstract state of Fig. 6(a) falls into the case in which the next list interruption is a node pointed-to by some regular variable (`z` in this case) and not heap-shared (case 3). The update formulae for this case are the following:

$$\begin{aligned}
UList_1[z_1,z_2]' &= UList_1[z_1,z_2] \wedge \neg Aliased[z_1,y] \\
UList_1[z_1,\text{null}]' &= UList_1[z_1,\text{null}] \vee Aliased[z_1,y] \\
UList_2[z_1,z_2]' &= UList_2[z_1,z_2] \wedge \neg Aliased[z_1,y] \\
UList[z_1,z_2]' &= UList[z_1,z_2] \wedge \neg Aliased[z_1,y] \\
UList[z_1,\text{null}]' &= UList[z_1,\text{null}] \vee Aliased[z_1,y]
\end{aligned}$$

Applying this update to the abstract state of Fig. 6(a) yields the abstract state of Fig. 6(b).

In Appendix A, we show that how to produce these formulae manually by applying rewrite rules.

5 Canonical Abstraction for Singly-Linked Lists

In this section, we show how canonical abstraction, with an appropriate set of predicates, provides a rather precise abstraction for (potentially cyclic) singly-linked lists.

5.1 The Abstraction

As in Sec. 4, the idea is to partition the heap into a linear number of uninterrupted list segments, where each segment is delimited by a pair of variables (possibly including auxiliary variables). The predicates we use for canonical abstraction are shown in Table 7. The predicates of the form *cul*$[x](v)$, for $x \in$ *Var*, record uninterrupted lists starting from the node pointed-to by x.

Table 7. Predicates used for the canonical abstraction and their meaning. We use the shorthand *UList*(u, v) as defined in Def. 3

Predicates	**Intended Meaning**	**Defining Formulae**
$\{\, x(v) : x \in \mathit{Var} \,\}$	object v is pointed-to by x	
$\{\, \mathit{cul}[x](v) : x \in \mathit{Var} \,\}$	there exists an uninterrupted list to v, starting from the node pointed-to by x	$\exists v_x : x(v_x) \wedge \mathit{UList}(v_x, v)$

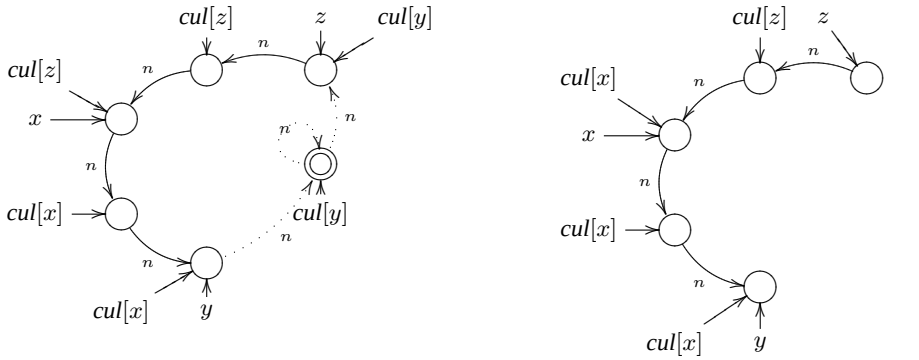

Fig. 7. The abstract effect of `y.n=null` under canonical abstraction. (a) canonical abstraction of the state of Fig. 3(a); (b) result of applying the abstract transformer of `y.n=null` to (a)

Example 6. Fig. 7(a) shows an abstract state abstracting the concrete state of Fig. 3(a). The predicates *cul*$[x](v)$, *cul*$[y](v)$, and *cul*$[z](v)$ record uninterrupted list segments. Note that, in contrast to the abstract state of Fig. 4(b) (which uses the standard TVLA predicates), the abstract configuration of Fig. 7(a) records the order between the reference variables, and is therefore able to observe that x is not pointing to an object on the list from y to z.

6 Discussion

Equivalence of the Canonical Abstraction and the Predicate Abstraction. We first show that the two abstractions — the predicate abstraction of Sec. 4, and the canonical

abstraction of Sec. 5 — are equivalent. That is, both observe the same set of distinctions between concrete heaps.

Theorem 1. *The abstractions presented in Section 4 and in Section 5 are equivalent.*

Proof. See Appendix B.

The Number of Predicates Used by the Abstractions. The next proposition shows that in fact only a logarithmic number of auxiliary variables is required for every regular program variable, in order to name all heap-shared nodes.

Proposition 3. *The* heap-sharing depth *in any heap is bounded from above by* $m = \lfloor \log n \rfloor + 1$. *In other words, auxiliary variables* $x_{s,k}$ *where* $k > m$ *never point to nodes.*

Proof. See Appendix B.

Using Proposition 3 we can reduce the number of unary predicates needed for the canonical abstraction to $O(n \log n)$, and the number of predicates needed for the predicate abstraction to $O((n \log n)^2)$, without affecting precision.

In general, the number of predicates needed by a predicate abstraction to simulate a given canonical abstraction is exponential in the number of unary predicates used by the canonical abstraction. It is interesting to note that, in this case, we were able to simulate the canonical abstraction using a sub-exponential number of nullary predicates.

We note that there exist predicate abstractions and canonical abstractions that are equivalent to the most precise member of the family of abstractions presented in the previous sections (i.e., with the full set of auxiliary variables) but require less predicates. We give the intuition to the principles underlying those abstractions and refer the reader to [18] for the technical details.

In heaps that do not contain cycles, the predicates in Table 3 are sufficient for keeping different uninterrupted lists from being merged. We can "reduce" general heaps to heaps without cycles by considering only interruptions that occur on cycles:

$$\mathit{Interruption}_c(v) \equiv \mathit{Interruption}(v) \wedge \mathit{OnCycle}(v) \ ,$$

and use these interruptions to break cycles by redefining the formulae for uninterrupted lists to use $\mathit{Interruption}_c$ instead of *Interruption*. Now, a linear number of auxiliary variables can be used to syntactically capture those interruptions. For every reference variable x, we add an auxiliary variable x_c, which is captured by the formula

$$\begin{aligned} x_c(v) \equiv\ & x(v) \wedge \mathit{OnCycle}(v) \vee \\ & \exists v_1, v_2 . x(v_1) \wedge n^*(v_1, v_2) \wedge \neg \mathit{OnCycle}(v_2) \wedge n(v_2, v) \ . \ . \end{aligned}$$

The set of all variables is defined by $\mathit{Var}' = \mathit{PVar} \cup \{x_c \mid x \in \mathit{PVar}\}$, and the predicates in Table 8 define the new canonical abstraction.

Recording Numerical Relationships. We believe that our abstractions can be generalized along the lines suggested by Deutsch in [9], by capturing numerical relationships between list lengths. This will allow us to prove properties of programs which traverse correlated linked lists, while maintaining the ability to conduct strong updates, which could not be handled by Deutsch. Indeed, in [10] numerical and canonical abstractions were combined in order to handle such programs.

Table 8. Predicates used for the new canonical abstraction with linear number of predicates. The shorthand *UList*$_c$ denotes an uninterrupted list where interruptions are defined by *Interruption*$_c$

Predicates	**Intended Meaning**	**Defining Formulae**
$\{ x(v) : x \in \mathit{Var}' \}$	object v is pointed-to by x	
$\{ cul_c[x](v) : x \in \mathit{Var}' \}$	there exists an uninterrupted list to v, starting from the node pointed-to by x	$\exists v_x : x(v_x) \wedge \mathit{UList}_c(v_x, v)$
$is(v)$	u is heap-shared	*HeapShared*(v)

7 Experimental Results

We implemented in TVLA the analysis based on the predicates and abstract transformers described in Section 2.3. We applied it to verify various specifications of programs operating on lists, described in Table 9. For all examples, we checked the absence of null dereferences and memory leaks. For the running example and reverse_cyclic we also verified that the output list is cyclic and partially cyclic, respectively.

The experiments were conducted using TVLA version 2, running with SUN's JRE 1.4, on a laptop computer with a 796 MHZ Intel Pentium Processor with 256 MB RAM.

The results of the analysis are shown in Table 9. In all of the examples, the analysis produced no false alarms. In contrast, TVLA, with the abstraction predicates in Table 1, is unable to prove that the output of reverse_cyclic is a partially cyclic list and that the output of removeSegment is a cyclic list.

The dominating factor in the running times and memory consumption is the loading phase, in which the predicates and update formulae are created (and explicitly represented). For example, the time and space consumed during the chaotic iteration of the `merge` example is 8 seconds and 7.4 MB, respectively.

Table 9. Time, space and number of errors measurements. Rep. Err. is the number of errors reported by the analysis, and Act. Err. is the number of real errors

Benchmark	**Description**	**Time (sec)**	**Space (MB)**	**Rep. Err./ Act. Err.**
create	Dynamically allocates a new linked list	3	1.8	0/0
delete	Removes an element from a list	7	9.1	0/0
deleteAll	Deallocates a list	3	2.7	0/0
getLast	Retrieves the last element in a list	4	4	0/0
insert	Inserts an element into a sorted list	9	13.5	0/0
merge	Merges two sorted lists into a single list	15	29.6	0/0
removeSegment	The running example	7	8.4	0/0
reverse	Reverses an acyclic list in-place	5	6	0/0
reverse_cyclic	reverse, applied to a partially cyclic list	2	7.1	0/0
rotate	Moves the first element after the last element	6	7.9	0/0
search	Searches for an element with a specified value	3	2.1	0/0
search_nullderef	Erroneous implementation of search that dereferences a null pointer	3	2.4	1/1
swap	Swaps the first two elements in a list	6	8.8	0/0

Acknowledgements

The authors wish to thank Alexey Loginov, Thomas Reps, and Noam Rinetzky for their contribution to this paper.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 203–213, June 2001.
2. T. Ball and S. Rajamani. Generating abstract explanations of spurious counterexamples in c programs. Report MSR-TR-2002-09, Microsoft Research, Microsoft Redmond, Jan. 2002. http://research.microsoft.com/slam/.
3. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *Proceedings of the 1999 European Symposium On Programming*, pages 2–19, Mar. 1999.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, M. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In J. J. B. Fenwick and C. Norris, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)*, volume 38, 5 of *ACM SIGPLAN Notices*, pages 196–207, New York, June 9–11 2003. ACM Press.
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. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. Computer Aided Verification*, pages 154–169, 2000.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Symp. on Principles of Prog. Languages*, pages 269–282, New York, NY, 1979. ACM Press.
8. 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.
9. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.
10. D. Gopan, F. DiMaio, N.Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algs. for the Construct. and Anal. of Syst.*, pages 512–529, 2004.
11. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *LNCS*, 1254:72–83, 1997.
12. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
13. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. *Proc. Computer Science Logic*, pages 160–174, Sept. 2004.
14. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. *ACM SIGPLAN Notices*, 36(3):14–26, Mar. 2001.
15. J. Jensen, M. Joergensen, N.Klarlund, and M. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proc. Conf. on Prog. Lang. Design and Impl.*, 1997.
16. N. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

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. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. Technical Report TR-2005-01-191212, Tel Aviv University, 2005.
19. Microsoft Research. The SLAM project. http://research.microsoft.com/slam/, 2001.
20. M. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc*, 141(1):1–35, 1969.
21. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. Conf. on Prog. Lang. Design and Impl.*, volume 37, 5, pages 83–94, June 2002.
22. 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.
23. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
24. R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proc. of the 10th International Static Analysis Symposium, SAS 2003*, volume 2694 of *LNCS*, June 2003.
25. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 25–34. ACM Press, 2004.

A Deriving the Best Transformer for `y.n=null`

In order to simplify the definition of the transformer for `y.n = null`, we split it to five different cases, shown in Table 10, based on classification of the next list interruption. The table uses the following shorthand notations:

$$\begin{aligned}
\mathit{ListToInDegree2}[y] &= \bigvee_{z_1 \in \mathit{Var}} \mathit{UList}[z_1, y_s] \wedge \neg \mathit{Aliased}[y, z_1] \wedge \\
&\quad \bigwedge_{z_2 \in \mathit{Var}} \mathit{UList}[z_2, y_s] \rightarrow (\mathit{Aliased}[z_2, y] \vee \mathit{Aliased}[z_2, z_1]) \\
\mathit{ListRegularVar}[y] &= \bigvee_{w \in \mathit{PVar}} \mathit{UList}[y, w] \\
\mathit{ListToHeapShared}[y] &= \bigvee_{w \in \mathit{PVar}} \mathit{UList}[y, w_s]
\end{aligned}$$

Table 10. The different cases considered when defining the abstract transformer for the statement `y.n = null`

Case	Next List Interruption	Precondition
1	is a heap-shared node not pointed by any regular variable, with in-degree $= 2$	$\neg(\mathit{UList}[y, \text{null}] \vee \mathit{ListRegularVar}[y]) \wedge$ $\mathit{ListToInDegree2}[y]$
2	is null	$\mathit{UList}[y, \text{null}]$
3	is a node pointed by some regular variable and not heap shared	$\mathit{ListRegularVar}[y] \wedge \neg \mathit{ListToHeapShared}[y]$
4	is a heap-shared node with in-degree > 2	$\mathit{ListToHeapShared}[y] \wedge \neg \mathit{ListToInDegree2}[y]$
5	is a node pointed by a regular variable and heap shared, with in-degree $= 2$	$\mathit{ListRegularVar}[y] \wedge \mathit{ListToHeapShared}[y] \wedge$ $\mathit{ListToInDegree2}[y]$

We show that manual construction of the best transformer results with the same formulae provided in Sec. 4. The derivation is shown Table 11. For each predicate, we first show its defining formula after applying the concrete effect of the statement `y.n=null`. We then rewrite this formula to an equivalent formula that is folded into the nullary predicates of our predicate-abstraction vocabulary (of Table 6). In the process of rewriting, we use transformations of FO^{TC} under the assumption that formulae describe heap configurations satisfying the integrity rules of the following definition:

Definition 4 (Integrity Rules). *We require that every heap configuration satisfies the following integrity rules:*

1. *for every unary predicate $x(v)$ representing a reference variable,* $\forall v_1, v_2 . x(v_1) \wedge x(v_2) \rightarrow v_1 = v_2$
2. *for the predicate $n(v_1, v_2)$ representing the `n` field,* $\forall v, v_1, v_2 . n(v, v_1) \wedge n(v, v_2) \rightarrow v_1 = v_2$

In the process of rewriting, we also use the rewrite rules of the following lemma. When a rule from the lemma is used in the rewriting, we note its number in brackets. We use $[*]$ to denote a rewrite using FO^{TC} transformations (assuming formulae describe heap configurations that satisfy the above consistency rules).

Lemma 1. *The following always hold:*

(I) $\neg PtByVar(u) \Rightarrow \neg y(u)$
(II) $\neg Interruption(u) \Rightarrow \neg y(u)$

The following hold under the precondition of case 3*:*

(III) $\mathit{Interruption}'(u) = \mathit{Interruption}(u)$
(IV) $\mathit{UList}'(v_1, v_2) = \mathit{UList}(v_1, v_2) \wedge \neg y(v_1)$
(V) $\mathit{UListNULL}'(v_1, v_2) = \mathit{UListNULL}(v_1, v_2) \vee y(v_1)$

where the primed values of shorthands denote their value after a after applying the effect of the statement `y.n = null`*.*

Proof. in Sec. B.

Table 11. Derivation of the transformer for `y.n=`null for case 3.

$\mathit{UList}_1[z_1, z_2]'$	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge n(v_1, v_2) \wedge \neg y(v_1)$	$[*]$
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge n(v_1, v_2) \wedge \exists v_3.z_1(v_3) \wedge \neg y(v_3)$	$[*]$
	$\mathit{UList}_1[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y]$	
$\mathit{UList}_1[z_1, \text{null}]'$	$\exists v.z_1(v) \wedge \forall u.\neg(n(v, u) \wedge \neg y(v))$	$[*]$
	$\exists v.z_1(v) \wedge \forall u.(\neg n(v, u) \vee y(v))$	$[*]$
	$\exists v.y(v) \vee z_1(v) \wedge \forall u.\neg n(v, u)$	$[*]$
	$\mathit{UList}_1[z_1, \text{null}] \vee \mathit{Aliased}[z_1, y]$	
$\mathit{UList}_2[z_1, z_2]'$	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \exists m.\neg \mathit{Interruption}'(m) \wedge$ $\wedge (n(v_1, m) \wedge \neg y(v_1)) \wedge (n(m, v_2) \wedge \neg y(m))$	$[III]$
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \exists m.\neg \mathit{Interruption}(m) \wedge$ $\wedge (n(v_1, m) \wedge \neg y(v_1)) \wedge (n(m, v_2) \wedge \neg y(m))$	$[II]$
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \exists m.\neg \mathit{Interruption}(m) \wedge$ $\wedge n(v_1, m) \wedge \neg y(v_1) \wedge n(m, v_2)$	$[*]$
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \exists m.\neg \mathit{Interruption}(m) \wedge$ $\wedge n(v_1, m) \wedge n(m, v_2) \wedge z_1(v_1) \wedge \neg y(v_1)$	$[*]$
	$\mathit{UList}_2[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y]$	
$\mathit{UList}[z_1, z_2]'$	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \mathit{UList}'(v_1, v_2)$	$[IV]$
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \neg y(v_1) \wedge \mathit{UList}(v_1, v_2)$	$[*]$
	$\mathit{UList}[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y]$	
$\mathit{UList}[z_1, \text{null}]'$	$\exists v_1.z_1(v_1) \wedge \mathit{UListNULL}(v_1)$	$[V]$
	$\exists v_1.z_1(v_1) \wedge \mathit{UListNULL}(v_1) \vee y(v_1)$	$[*]$
	$\mathit{UList}[z_1, \text{null}] \vee \mathit{Aliased}[z_1, y]$	

B Proofs

Proof (of Proposition 1). A program variable points to at most 1 element, and therefore the number of list elements pointed by all program variables is at most n. The proof that the number of heap-shared elements is at most n is done by induction on the number of non-null variables.
Basis: Suppose the only non-null program variable is `x`. The proof is split into the following cases.

Case 1: The path from the element pointed by x reaches null. In this case, there are no heap-shared elements.

Case 2: The path from the element pointed by x reaches the element pointed by x, thereby forming a cycle. In this case, there are no heap-shared elements.

Case 3: The path from the element pointed by x reaches an element other than the one pointed by x. In this case, there is exactly 1 heap-shared element. *Induction hypothesis*: Assume that the proposition holds for $k \geq 0$ non-null program variables.
Induction step: Suppose there are $k+1$ non-null program variables $x_1, \ldots, x_{k+1}$. Let H_k be the sub-heap consisting of only the elements reachable from $x_1, \ldots, x_k$ and the links between them. The proof is split into the following cases, according to the interaction between variable x_{k+1} and H_k.

Case 1: The set of elements in H_k and the set of elements reachable from variables x_{k+1} do not intersect. By the induction hypothesis, the sub-heap H_k contains at most k heap-shared elements, and the sub-heap containing elements reachable from x_{k+1} contains at most a single heap-shared element. Therefore, the entire heap contains at most $k+1$ heap-shared elements.

Case 2: Variable x_{k+1} points to an element in H_k. Since the variable x_{k+1} in itself does not contribute to the in-degree of the element it points to, setting x_{k+1} to null does no change the number of heap-shared elements. Therefore, by the induction hypothesis, the heap contains at most k heap-shared elements.

Case 3: Variable x_{k+1} connects to the sub-heap H_k via the path $[u_1, \ldots, u_m]$ (none of $u_1, \ldots, u_m$ is heap-shared). By the induction hypothesis, H_k contains at most k heap-shared elements. The link from u_m to an element in H_k contributes at most a single heap-shared element to the entire heap, and therefore the entire heap contains at most $k+1$ heap-shared elements.

Proof (of Proposition 2). To prove the first part of the claim, suppose u is heap-shared. If u is pointed-to by a program variable then the claim trivially holds. Since we assume that the heap is garbage-free, node u is reachable from some program variable. Let x be the program variable that reaches u on the shortest path. Obviously no node on the path is pointed-to by a program variable (otherwise there would be a shorter path from a different variable). By Corollary 1, the path from x to u consists of k maximal uninterrupted lists, for some $k < n$. Therefore, by definition, auxiliary variable $x_{s,k}(v)$ points to u.

The second part of the claim is proved by induction on the sharing-depth k.
Basis: The term $\mathit{HeapShared}(v) \wedge \neg \mathit{PtByVar}(v)$ means that $x_{s,1}(v)$ can hold only for a subset of interruptions that are heap shared but not pointed by any (regular) program variable. The term $\exists v_x.x(v_x) \wedge \mathit{UList}(v_x, v)$ further restricts the set of nodes to only ones that are reachable by an uninterrupted list from a node pointed by the variable x. Since x is a reference variable, it can point to at most one node, which means that $\exists v_x.x(v_x) \wedge \mathit{UList}(v_x, v)$ holds for at most one interruption. Therefore, the entire conjunction $\exists v_x.x(v_x) \wedge \mathit{UList}(v_x, v) \wedge \mathit{HeapShared}(v) \wedge \neg \mathit{PtByVar}(v)$ holds for at most one node.

Induction hypothesis: Assume that the proposition holds for every reference variables and sharing-depth $i \leq k$.
Induction step: By the induction hypothesis $x_{s,k}(v)$ holds for at most one node. There-

fore, the arguments that were used to prove the basis hold (with x replaced by $x_{s,k}$) for the sub-formula

$$\exists v_k.x_{s,k}(v_k) \wedge \mathit{UList}(v_k, v) \wedge \mathit{HeapShared}(v) \wedge \neg \mathit{PtByVar}(v) \ .$$

The conjunction $\neg(\bigvee_{m=1\ldots k} x_{s,m}(v))$ can only further restrict the set of nodes for which the sub-formula above holds, and therefore the claim holds for the entire formula.

Proof (of Proposition 3). Fig. 8 shows a representative case of a concrete heap where the heap-sharing depth reaches the upper bound.

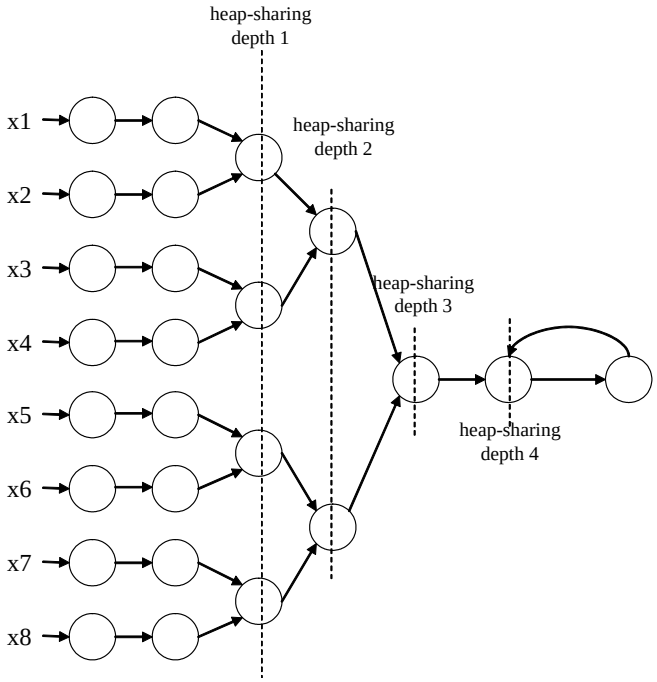

Fig. 8. A representative case for a heap sharing depth that reaches the upper bound. The vertical dashed lines are used to show the different levels of heap-sharing depth

We will use the simple fact that, since the out-degree of any node in the heap is at most 1, every connected component of the heap (considering the undirected version of the relation $n(u, v)$) contains at most one simple cycle.

Let u be a heap-shared node of depth $k > 1$. There are two cases:

Node u does not reside on a cycle. Consider the part of heap containing nodes that reach u. These nodes, along with u form a connected component without cycles where the out-degree of every node is at most 1. This is a tree with program variables at the leaves and u as a root.

The fact that u has heap-sharing depth k means that u is reachable from at least two distinct nodes a and b of heap-sharing depth $k - 1$. In addition, a and b do not reside on a cycle.

The same reasoning can now be applied to a and b, obtaining the fact that u is reachable from at least 4 nodes of heap-sharing depth $k-2$. The reasoning goes on until we get to the leaves of the tree, and have that u is reachable from 2^k nodes that are pointed by program variables. This means that $2^k \leq n$ and therefore $k \leq \lfloor \log n \rfloor$.

Node u resides on a cycle. Since node u is heap-shared and found on a cycle, there are two distinct interrupting nodes a and b such that the lists from a to u and from b to u are maximal uninterrupted lists, and a is on the cycle where u is and b is out of the cycle.

Since b does not reside on a cycle, we have already shown that b can have a heap-sharing depth of at most $\lfloor \log n \rfloor$. Therefore, node u has heap-sharing depth of at most $\lfloor \log n \rfloor + 1$.

Proof (Lemma 1). The first claims in the Lemma are mostly immediate from the definitions of the shorthand notations.

(I)
$$\neg \mathit{PtByVar}(u) = \neg \bigvee_{\mathit{var} \in \mathit{PVar}} \mathit{var}(u) \Rightarrow \neg y(u)$$
(II)
$$\neg Interruption(u) = \neg \mathit{HeapShared}(u) \wedge \neg \mathit{PtByVar}(u) \Rightarrow \neg y(u)$$
(III) we begin by showing that $\mathit{HeapShared}'(u) = \mathit{HeapShared}(u)$
$$\mathit{HeapShared}'(u) = \exists a, b. n(a,u) \wedge \neg y(a) \wedge n(b,u) \wedge \neg y(b) \wedge (a \neq b)$$
by the precondition to this case
$$\mathit{HeapShared}'(u) = \exists a, b. n(a,u) \wedge n(b,u) \wedge (a \neq b) = \mathit{HeapShared}(u)$$
since $PtByVar(u)$ does not change under the action `y.n=null`,
it follows that $\mathit{Interruption}'(u) = \mathit{Interruption}(u)$.

(IV)

$\mathit{UList}'(v_1, v_2)$ $(n(v_1, v_2) \wedge \neg y(v_1) \vee$
$(\exists m. \neg \mathit{Interruption}'(m) \wedge (((n(v_1, m) \wedge \neg y(v_1)) \wedge (n(m, v_2) \wedge \neg y(m))) \vee$
$(\exists m_1, m_2. (n(v_1, m_1) \wedge \neg y(v_1) \wedge \neg y(v_1)) \wedge (n(m_2, v_2) \wedge \neg y(m_2)) \wedge$
$(\text{TC } a, b : n(a, b) \wedge \neg y(a) \wedge \neg \mathit{Interruption}'(a) \wedge \neg \mathit{Interruption}'(b))(m_1, m_2))) \; [III]$
$\exists v_1, v_2. z_1(v_1) \wedge z_2(v_2) \wedge (n(v_1, v_2) \wedge \neg y(v_1) \vee$
$(\exists m. \neg \mathit{Interruption}(m) \wedge (n(v_1, m) \wedge \neg y(v_1)) \wedge (n(m, v_2) \wedge \neg y(m))) \vee$
$(\exists m_1, m_2. n(v_1, m_1) \wedge \neg y(v_1) \wedge n(m_2, v_2) \wedge$
$(\text{TC } a, b : n(a, b) \wedge \neg y(a) \wedge \neg \mathit{Interruption}(a) \wedge \neg \mathit{Interruption}(b))(m_1, m_2))) \; [II]$
$\exists v_1, v_2. z_1(v_1) \wedge z_2(v_2) \wedge (n(v_1, v_2) \wedge \neg y(v_1) \vee$
$(\exists m. \neg \mathit{Interruption}(m) \wedge n(v_1, m) \wedge \neg y(v_1) \wedge n(m, v_2)) \vee$
$(\exists m_1, m_2. n(v_1, m_1) \wedge \neg y(v_1) \wedge n(m_2, v_2) \wedge$
$(\text{TC } a, b : n(a, b) \wedge \neg \mathit{Interruption}(a) \wedge \neg \mathit{Interruption}(b))(m_1, m_2))) \; [*]$
$\exists v_1, v_2. z_1(v_1) \wedge z_2(v_2) \wedge \neg y(v_1) \wedge (n(v_1, v_2) \vee$
$(\exists m. \neg \mathit{Interruption}(m) \wedge n(v_1, m) \wedge n(m, v_2)) \vee$
$(\exists m_1, m_2. n(v_1, m_1) \wedge n(m_2, v_2) \wedge$
$(\text{TC } a, b : n(a, b) \wedge \neg \mathit{Interruption}(a) \wedge \neg \mathit{Interruption}(b))(m_1, m_2))) \; [*]$
$\mathit{UList}'(v_1, v_2) \wedge \neg y(v_1)$

(V)

$\mathit{UListNULL}'(u)$ $\forall v. \neg(n(u, v) \wedge \neg y(u)) \vee$
$\exists m. (n(u, m) \wedge \neg y(u)) \wedge \neg \mathit{Interruption}'(m) \wedge \mathit{UListNULL}'_1(m) \vee$
$\exists m_1, m_2 : (n(u, m_1) \wedge \neg y(u)) \wedge \mathit{UListNULL}'_1(m_2) \wedge$
$(\text{TC } a, b : (n(a, b) \wedge \neg y(a)) \wedge \neg \mathit{Interruption}'(a) \wedge$
$\neg \mathit{Interruption}'(b))(m_1, m_2) \; [III]$
$\forall v. \neg(n(u, v) \wedge \neg y(u)) \vee$
$\exists m. (n(u, m) \wedge \neg y(u)) \wedge \neg \mathit{Interruption}(m) \wedge (\mathit{UListNULL}_1(m) \vee y(m)) \vee$
$\exists m_1, m_2 : (n(u, m_1) \wedge \neg y(u)) \wedge (\mathit{UListNULL}_1(m_2) \vee y(m_2)) \wedge$
$(\text{TC } a, b : (n(a, b) \wedge \neg y(a)) \wedge \neg \mathit{Interruption}(a) \wedge$
$\neg \mathit{Interruption}(b))(m_1, m_2) \; [II] + [*]$
$\forall v. \neg(n(u, v) \wedge \neg y(u)) \vee$
$\exists m. (n(u, m) \wedge \neg y(u)) \wedge \neg \mathit{Interruption}(m) \wedge (\mathit{UListNULL}_1(m) \vee y(m)) \vee$
$\exists m_1, m_2 : (n(u, m_1) \wedge \neg y(u)) \wedge (\mathit{UListNULL}_1(m_2) \vee y(m_2)) \wedge$
$(\text{TC } a, b : (n(a, b) \wedge \neg y(a)) \wedge \neg \mathit{Interruption}(a) \wedge$
$\neg \mathit{Interruption}(b))(m_1, m_2) \; [II]$
$\forall v. (\neg n(u, v) \vee y(u)) \vee$
$\exists m. (n(u, m) \wedge \neg y(u)) \wedge \neg \mathit{Interruption}(m) \wedge \mathit{UListNULL}_1(m) \vee$
$\exists m_1, m_2 : (n(u, m_1) \wedge \neg y(u)) \wedge \mathit{UListNULL}_1(m_2) \wedge$
$(\text{TC } a, b : n(a, b) \wedge \neg \mathit{Interruption}(a) \wedge$
$\neg \mathit{Interruption}(b))(m_1, m_2) \; [*]$
$\forall v. (\neg n(u, v)) \vee y(u) \vee$
$\neg y(u) \wedge \exists m. n(u, m) \wedge \neg \mathit{Interruption}(m) \wedge \mathit{UListNULL}_1(m) \vee$
$\neg y(u) \wedge \exists m_1, m_2 : n(u, m_1) \wedge \mathit{UListNULL}_1(m_2) \wedge$
$(\text{TC } a, b : n(a, b) \wedge \neg \mathit{Interruption}(a) \wedge$
$\neg \mathit{Interruption}(b))(m_1, m_2) \; [*](resolution)$
$\forall v. (\neg n(u, v)) \vee y(u) \vee$
$\exists m. n(u, m) \wedge \neg \mathit{Interruption}(m) \wedge \mathit{UListNULL}_1(m) \vee$
$\exists m_1, m_2 : n(u, m_1) \wedge \mathit{UListNULL}_1(m_2) \wedge$
$(\text{TC } a, b : n(a, b) \wedge \neg \mathit{Interruption}(a) \wedge$
$\neg \mathit{Interruption}(b))(m_1, m_2) \; [*]$
$\mathit{UListNULL}'(u) \vee y(u)$

□

B.1 Proving Theorem 1

We want to prove that the predicate abstraction, β_{PredAbs}, presented in Sec. 4 and the canonical abstraction, β_{Canonic}, presented in Sec. 5 are equivalent. Before delving into the details, we make the claim more precise.

Recall that both abstractions are parameterized by an index k ranging from 1 to n (the number of program variables). The proof here is for $k = n$ (i.e., the full set of auxiliary variables is used).

By equivalence of abstractions, we mean that, for any two concrete heaps C_1 and C_2 (2-valued structures), the following holds:

$$\beta_{\mathit{PredAbs}}(C_1) = \beta_{\mathit{PredAbs}}(C_2) \text{ when } \beta_{\mathit{Canonic}}(C_1) = \beta_{\mathit{Canonic}}(C_2) \ .$$

We will use the following shorthand notations

$$\begin{aligned} A_1^P &= \beta_{\mathit{PredAbs}}(C_1) \ , \\ A_2^P &= \beta_{\mathit{PredAbs}}(C_2) \ , \\ A_1^C &= \beta_{\mathit{Canonic}}(C_1) \ , \\ A_2^C &= \beta_{\mathit{Canonic}}(C_2) \ , \end{aligned}$$

and make use of the embedding functions f and g such that $C_1 \sqsubseteq^f A_1^C$ and $C_2 \sqsubseteq^g A_2^C$. With these notations we rephrase the equivalence claim: $A_1^P = A_2^P$ when $A_1^C = A_2^C$. Note that by $A_1^C = A_2^C$ we mean that structures A_1^C and A_2^C are isomorphic, i.e., $A_1^C \sqsubseteq A_2^C$ and $A_2^C \sqsubseteq A_1^C$. The semantics of formulae for 3-valued structures is explained in [23].

We will sometimes write the name of a predicate from Table 6 as shorthand for its defining formula. The exact meaning, however, should be clear from the context, depending on whether the structure referred to is concrete (2-valued) or abstract (3-valued).

Since the join operator used in both kinds of abstractions is the same—set union—the equivalence of the abstractions carries over from single concrete heaps to sets of concrete heaps.

Proof Structure. We want to show that both abstractions are able to make exactly the same distinctions about any two concrete heaps. We start by showing that whenever C_1 and C_2 assign different interpretations to a predicate in P^A (indicating that their predicate abstraction is different), A_1^C is different from A_2^C. This is shown by a case analysis according the 7 predicate types that appear in Table 6 in order of appearance. In each case we assume that the predicates considered in previous cases have the same interpretation in both C_1 and C_2. Finally, we consider the case where all predicates in P^A have the same interpretation in C_1 and C_2 (indicating that their predicate abstraction is the same), and show that $A_1^C = A_2^C$.

We use the following lemmas to show that two concrete heaps are different under canonical abstraction[3]. In the lemmas, we use the shorthand notations introduced above.

In the proofs of the lemma we will use use the fact that, by the Embedding Theorem of [23], if two 3-valued structures are isomorphic then the value of every closed formula evaluates to the value in both structures.

Lemma 2. *Let C_1 and C_2 be a pair of 2-valued structures, and let $\varphi(v)$ be a conjunction of unary predicates and negations of unary predicates.*

If $[\![\exists v : \varphi(v)]\!]^{C_1} = 1$ and $[\![\exists v : \varphi(v)]\!]^{C_2} = 0$ then $A_1^C \neq A_2^C$.

Proof. Let v be a node in U^{C_1} for which $\varphi(v)$ holds. Since canonical abstraction preserves the definite values of unary predicates, $\varphi(v)$ evaluates to a definite value for $f(v)$ (the same value as in C_1). Therefore, $[\![\exists v : \varphi(v)]\!]^{A_1^C} = 1$. Since there is no node in U^{C_1} for which $\varphi(v)$ holds, there is also node node in A_2^C for which $\varphi(v)$ holds, and therefore $[\![\exists v : \varphi(v)]\!]^{A_2^C} = 0$.

We conclude that $A_1^C \neq A_2^C$.

□

Lemma 3. *Let C_1 and C_2 be a pair of 2-valued structures, and let $\varphi(v)$ be a conjunction of unary predicates and negations of unary predicates.*

If $\varphi(v)$ holds for exactly one individual u in C_1, and $\varphi(v)$ holds for more than one individual in C_2 then $A_1^C \neq A_2^C$.

Proof. Since $\varphi(v)$ holds for exactly one individual u in C_1, we have that $\varphi(v)$ holds for exactly one individual $v = f(u)$ in A_1^C. Therefore, $[\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_1^C} = 1$.

Let V_2 be the set of nodes in U^{C_2} for which $\varphi(v)$ holds. If $w = g(u) = g(v)$ for some pairs of nodes $u, v \in V_2$ then $eq(w, w) = 1/2$ and $[\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_2^C} = 1/2$. Otherwise, there $eq^{A_2^C}(g(u), g(v)) = 0$ for every distinct nodes $u, v \in V_2$, and therefore $[\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_2^C} = 0$.

In both cases $[\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_1^C} \neq [\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_2^C}$ and we conclude that $A_1^C \neq A_2^C$.

□

For intuition, Fig. 9 shows the different cases of concrete lists and their canonical abstraction, along with the values of the predicates from Table 6.

[3] The lemmas are stated for the canonical abstraction from Sec. 5, but they are actually true for canonical abstraction with any set of predicates.

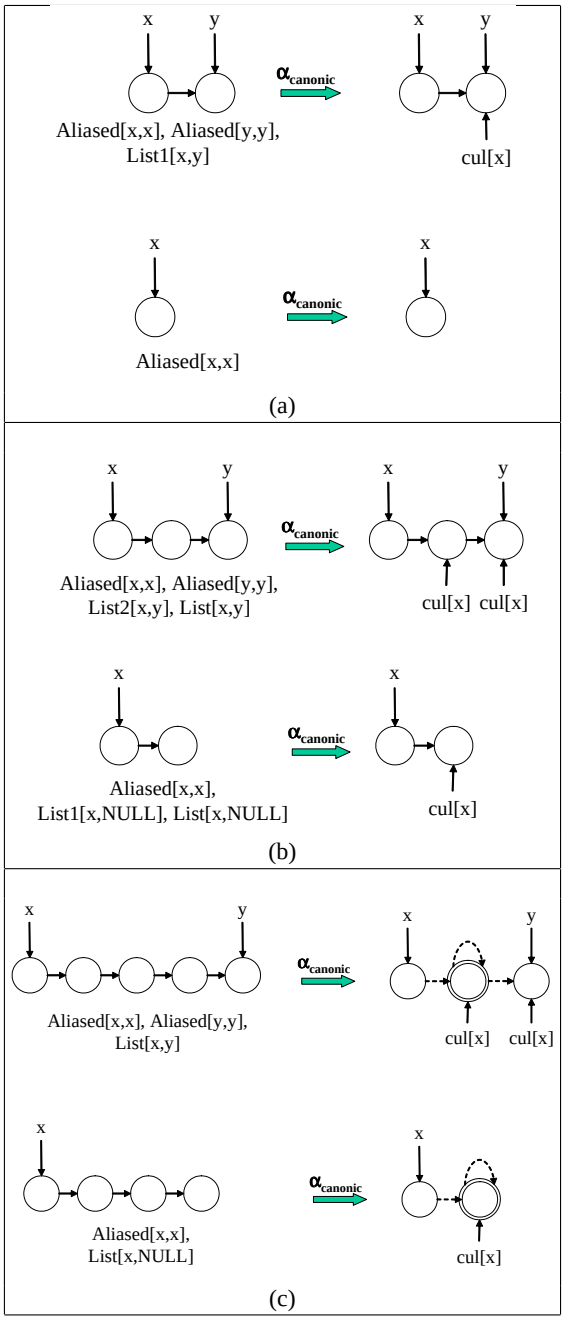

Fig. 9. Applying canonical abstraction to lists of different lengths: (a) lists of length 1, (b) lists of length 2, and (c) lists of length greater than 2

Proof (of Theorem 1).

Case 1 : Distinction by *Aliased*$[x, y]$ predicates. Assume that for two variables $x, y \in$ *Var* we have $[\![\textit{Aliased}[x, y]]\!]^{C_1} = 1$ and $[\![\textit{Aliased}[x, y]]\!]^{C_2} = 0$. Substituting the predicate *Aliased*$[x, y]$ with its defining formula from Table 6, we get $[\![\exists v : x(v) \wedge y(v)]\!]^{C_1} = 1$ and $[\![\exists v : x(v) \wedge y(v)]\!]^{C_2} = 0$. Therefore, by Lemma 2, $A_1^C \neq A_2^C$.

Case 2 : Distinction by *UList*$_1[x, y]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, and that for some $x, y \in$ *Var* we have $[\![\textit{UList}_1[x, y]]\!]^{C_1} = 1$ and $[\![\textit{UList}_1[x, y]]\!]^{C_2} = 0$.

Substituting the predicate *UList*$_1[x, y]$ with its defining formula from Table 6, we get $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)]\!]^{C_1} = 1$ and $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)]\!]^{C_2} = 0$. Let $u_x, y_u \in U^{C_1}$ be the unique (Proposition 2) nodes such that $x^{C_1}(u_x) = 1$ and $y^{C_1}(u_y) = 1$. From the assumption that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, we have that there exist unique nodes $v_x, v_y \in\in U^{C_2}$ such that $x^{C_2}(v_x) = 1$ and $y^{C_2}(v_y) = 1$.

We now have that there exists unique nodes $u'_x = f(u_x) \in A_1^C$ and $u'_y = f(u_y) \in A_1^C$ such that $x^{A_1^C}(u'_x) = 1$ and $x^{A_1^C}(u'_y) = 1$. Therefore, $n^{A_1^C}(u'_x, u'_y) = n^{C_1}(u_x, u_y) = 1$ and $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)]\!]^{A_1^C} = 1$.

Furthermore, there exists unique nodes $v'_x = g(v_x) \in A_2^C$ and $v'_y = g(v_y) \in A_2^C$ such that $x^{A_2^C}(v'_x) = 1$ and $x^{A_2^C}(v'_y) = 1$. Therefore, $n^{A_2^C}(v'_x, v'_y) = n^{C_2}(u_x, u_y) = 0$ and $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)]\!]^{A_2^C} = 0$.

We conclude that $A_1^C \neq A_2^C$.

Case 3 : Distinction by *UList*$_2[x, y]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$ and *UList*$_1[x, y]$, and that for some $x, y \in$ *Var* we have $[\![\textit{UList}_2[x, y]]\!]^{C_1} = 1$ and $[\![\textit{UList}_2[x, y]]\!]^{C_2} = 0$.

The meaning of $[\![\textit{UList}_2[x, y]]\!]^{C_1} = 1$ is that there exist two nodes v_x and v_y in U^{C_1}, which are pointed-to by variables x and y, respectively, and a third node v_m, such that v_x, v_m, v_y is a maximal uninterrupted list in C_1. Therefore, *cul*$[x](v) \wedge \neg y(v)$ holds uniquely for v_m in C_1. In addition, $[\![\textit{UList}_2[x, y]]\!]^{C_1} = 1$ implies $[\![\textit{UList}_1[x, y]]\!]^{C_1} = 0$, since a maximal uninterrupted list has a determined integer length. Now, since C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$ then there exist two nodes u_x and u_y in U^{C_2} that are pointed-to by variables x and y, respectively.

We consider the following three sub-cases: (i) There is no uninterrupted list between u_x and u_y. Therefore, $[\![\exists v : \textit{cul}[x](v) \wedge \neg y(v)]\!]^{C_2} = 0$, and by Lemma 2, $A_1^C \neq A_2^C$; (ii) There exists a maximal uninterrupted list between u_x and u_y of length 1. This possibility is ruled out since it contradicts the fact that $[\![\textit{UList}_1[x, y]]\!]^{C_1} = 0$ with our assumption that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$ and *UList*$_1[x, y]$; and (iii) There exists a maximal uninterrupted list between u_x and u_y of length > 2. This means that *cul*$[x](v) \wedge \neg y(v)$ holds for more than one node in C_2 (but only for v_m in C_1, and so by Lemma 3, $A_1^C \neq A_2^C$.

Case 4 : Distinction by *UList*$[x, y]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, *UList*$_1[x, y]$, and *UList*$_2[x, y]$; and that for some $x, y \in$ *Var* we have $[\![\textit{UList}[x, y]]\!]^{C_1} = 1$ and $[\![\textit{UList}[x, y]]\!]^{C_2} = 0$.

Since $[\![\textit{UList}[x, y]]\!]^{C_1} = 1$ we can substitute the definition of *cul*$[x](v)$ in the definition of *UList*$[x, y]$ and get $[\![\exists v : y(v) \wedge \textit{cul}[x](v)]\!]^{C_1} = 1$. Applying this substitution For

$[\![\mathit{UList}[x,y]]\!]^{C_2} = 0$ gives us $[\![\exists v : y(v) \wedge \mathit{cul}[x](v)]\!]^{C_2} = 0$. Therefore, by Lemma 2, $A_1^C \neq A_2^C$.

Case 5 : Distinction by $\mathit{UList}_1[x, \mathbf{null}]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form $\mathit{Aliased}[x,y]$, $\mathit{UList}_1[x,y]$, $\mathit{UList}_2[x,y]$, and $\mathit{UList}[x,y]$; and that for some $x \in \mathit{Var}$ we have $[\![\mathit{UList}_1[x, \text{null}]]\!]^{C_1} = 1$ and $[\![\mathit{UList}_1[x, \text{null}]]\!]^{C_2} = 0$.

Since $[\![\mathit{UList}_1[x, \text{null}]]\!]^{C_1} = 1$, we have that there is no list emanating from the node pointed-to by x in C_1, and $[\![\exists v : \mathit{cul}[x](v)]\!]^{C_1} = 0$. Since $[\![\mathit{UList}_1[x, \text{null}]]\!]^{C_2} = 0$, we have that there is a non-empty list emanating from the node pointed-to by x in C_2, and $[\![\exists v : \mathit{cul}[x](v)]\!]^{C_2} = 1$. Therefore, by Lemma 2, $A_1^C \neq A_2^C$.

Case 6 : Distinction by $\mathit{UList}_2[x, \mathbf{null}]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form $\mathit{Aliased}[x,y]$, $\mathit{UList}_1[x,y]$, $\mathit{UList}_2[x,y]$, $\mathit{UList}[x,y]$, and $\mathit{UList}_1[x, \text{null}]$; and that for some $x \in \mathit{Var}$ we have $[\![\mathit{UList}_2[x, \text{null}]]\!]^{C_1} = 1$ and $[\![\mathit{UList}_2[x, \text{null}]]\!]^{C_2} = 0$.

We consider the following sub-cases: (i) There exists a maximal uninterrupted list of length 1 from the node pointed-to by x to null, in C_1, i.e., $[\![\mathit{UList}_2[x, \text{null}]]\!]^{C_2} = 1$. This case is ruled out, since by the assumption that C_1 and C_2 identify on all predicates of the form $\mathit{UList}_1[x, \text{null}]$ this would mean that $[\![\mathit{UList}_1[x, \text{null}]]\!]^{C_1} = 1$, which is not possible since there exists a maximal uninterrupted list of length 2 from that node to null and any maximal uninterrupted list has a determined integer length; (ii) There exists a maximal uninterrupted list of length > 2 from the node pointed-to by x to null, in C_1. This means that in C_1 the predicate $\mathit{cul}[x](v)$ holds for exactly one node (the one following the node pointed-to by x), and in C_2 the predicate $\mathit{cul}[x](v)$ holds for more than one node (all of the nodes following the node pointed-to by x). Therefore, by Lemma 3, $A_1^C \neq A_2^C$; and (iii) There is no maximal uninterrupted list from x to null in C_2, which means that there exists a maximal uninterrupted list from x to a (possible the same) variable y, i.e., $[\![\exists v : \mathit{cul}[x](v) \wedge y(v)]\!]^{C_2} = 1$. However, since in C_1 there is no maximal uninterrupted list from x to any variable, $[\![\exists v : \mathit{cul}[x](v) \wedge y(v)]\!]^{C_1} = 0$, and therefore, by Lemma 2, $A_1^C \neq A_2^C$.

Case 7 : Distinction by $\mathit{UList}[x, \mathbf{null}]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form $\mathit{Aliased}[x,y]$, $\mathit{UList}_1[x,y]$, $\mathit{UList}_2[x,y]$, $\mathit{UList}[x,y]$, $\mathit{UList}_1[x, \text{null}]$, and $\mathit{UList}_2[x, \text{null}]$; and that for some $x \in \mathit{Var}$ we have $[\![\mathit{UList}[x, \text{null}]]\!]^{C_1} = 1$ and $[\![\mathit{UList}[x, \text{null}]]\!]^{C_2} = 0$. (This reasoning here is the same as the third sub-case in the previous case.)

This means that in C_2 there exists a maximal uninterrupted list from x to a (possible the same) variable y, i.e., $[\![\exists v : \mathit{cul}[x](v) \wedge y(v)]\!]^{C_2} = 1$. However, since in C_1 there is no maximal uninterrupted list from x to any variable, $[\![\exists v : \mathit{cul}[x](v) \wedge y(v)]\!]^{C_1} = 0$, and therefore, by Lemma 2, $A_1^C \neq A_2^C$.

Case 8 : No distinctions by predicates from Table 6. Assume that C_1 and C_2 identify on all predicates from Table 6.

We show that A_1^C is isomorphic to A_2^C by showing that: (i) for every node $u_1 \in U^{A_1^C}$ there exists a unique corresponding node $u_2 \in U^{A_2^C}$ such that for every unary predicate $p(v)$ from Table 7 $p(u_1)^{A_1^C} = p(u_2)^{A_2^C}$ (i.e., A_1^C and A_2^C have the same set of canonic names); and (ii) for every pair of nodes $u_1, v_1 \in U^{A_1^C}$ and corresponding pair of nodes (with respect to the values of unary predicates) $u_2, v_2 \in U^{A_2^C}$, the equalities $n(u_1, v_1)^{A_1^C} = n(u_2, v_2)^{A_2^C}$ and $eq(u_1, v_1)^{A_1^C} = eq(u_2, v_2)^{A_2^C}$ hold.

Universe to universe bijection and preservation of unary predicates. Let u_1 be a node in $U^{A_1^C}$, and let X and L be subsets of *Var* such that the unary predicates that hold for u_1 in A_1^C are $x(u_1)$ for every $x \in X$ and $cul[x](v)$ for every $x \in L$.

We consider two cases separately according to the emptiness of X.

X **is non-empty.** From Proposition 2 we have that $f^{-1}(u_1) = \{v_1\}$, and $x^{C_1}(v_1) = 1$ for every $x \in X$. Thus, $[\![\exists v : x(v) \wedge y(v)]\!]^{C_1} = 1$ for every $x, y \in X$. From our assumption that C_1 and C_2 identify on all predicates of the form $Aliased[x, y]$ we get that $[\![\exists v : x(v) \wedge y(v)]\!]^{C_2} = 1$ for every $x, y \in X$. Using Proposition 2 we get that there exists a unique node $v_2 \in U^{C_2}$ such that $x^{C_2}(v_2) = 1$ for every $x \in X$. We denote by u_2 the node $g(v_2)$, which is designated as the corresponding node for u_1 in the isomorphism map, and using the definition of canonical abstraction we get that $x^{A_2^C}(u_2) = x^{A_1^C}(u_1)$ for every $x \in X$.

From the definition of the predicates $cul[x](v)$, we have that $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge UList(v_y, v_x)]\!]^{C_1}$ for every $y \in L$ and $x \in X$. From our assumption that C_1 and C_2 identify on all predicates of the form $UList[x, y]$ we get that $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge UList(v_y, v_x)]\!]^{C_2}$ for every $y \in L$ and $x \in X$. Using the definition of canonical abstraction we get that $cul[y]^{A_2^C}(u_2) = cul[y]^{A_1^C}(u_1)$ for every $y \in L$.

X **is empty.** Let $f^{-1}(u_1) = V_1$ be the set of nodes mapped by f to u_1. Since X is empty we have that for every node $v_1 \in V_1$: $x^{C_1}(v_1) = 0$ for every $x \in Var$ and $cul[x]^{C_1}(v_1) = 1$ for every $x \in L$. Either V_1 is part of a maximal uninterrupted list from x to null, or V_1 is part of a maximal uninterrupted list from x to some variable y. In either case, from our assumption that C_1 and C_2 identify on all predicates of the form $Aliased[x, y]$, $UList[x, y]$, and $UListNULL[x]$, we have that there exists a non-empty set of nodes $V_2 \subseteq U^{C_2}$ such that for every $v_2 \in V_2$: $x^{C_2}(v_2) = 0$ for every $x \in Var$ and $cul[x]^{C_2}(v_2) = 1$ for every $x \in L$. Therefore, if we denote by u_2 the image of V_2 under g, we get from the definition of canonical abstraction that $x^{A_2^C}(u_2) = x^{A_1^C}(u_1)$ for every $x \in X$ and $cul[y]^{A_2^C}(u_2) = cul[y]^{A_1^C}(u_1)$ for every $y \in L$. The uniqueness of u_2 is determined by the fact that the values of all unary predicates are considered for the nodes of V_2.

The correspondence by values of unary predicates defines a bijection $h : U^{A_1^C} \rightarrow U^{A_2^C}$ such that $h(u) = v$ when $p(u_1)^{A_1^C} = p(u_2)^{A_2^C}$ for every unary predicate $p(v)$ from Table 7.

Preservation of the binary predicate $eq(u, v)$**.** Since $eq(u, v)$ is interpreted as 0 in every 3-valued structure for distinct u and v, we are only interested in $eq(u, u)$.

Recall that by the meaning of the predicate $eq(u, v)$ its interpretation can either be 1 or $1/2$ (but never 0).

Let u_1 be a node in $U^{A_1^C}$ and let u_2 be $h(u_1)$. Assume that $eq(u_1, u_1)^{A_1^C} = 1/2$ (i.e., u_1 is a summary node). Let X be the set variables such that $x^{A_1^C}(u_1) = 1$ for every $x \in X$ and L be the set of variables such that $cul[y]^{A_1^C}(u_1) = 1$ for every $y \in L$. From Proposition 2 we get that $X = \emptyset$.

Denote by V_1 the set $f^{-1}(u_1)$. We have that $|V_1| > 1$, which means that V_1 are part of an uninterrupted list in C_1 containing more than two elements, which emanates from the node pointed-to by the variables in L.

Denote by V_2 the set $g^{-1}(u_2)$. Since we assumed that C_1 and C_2 identify on all predicates from Table 6, we get that from the node pointed-to by the variables in L emanates an uninterrupted list containing more than two elements in C_1. Hence, $|V_2| > 1$. Therefore, $eq^{A_2^C}(u_2, u_2) = 1/2$.

Preservation of the binary predicate $n(u, v)$**.** We will show that, for a structure in the image of canonical abstraction with the predicates from Table 7, the values of unary predicates together with the value of the predicate $eq(u, v)$, determine the value of the predicate $n(u, v)$. Since A_1^C and A_2^C are isomorphic with respect to those predicates, this completes the proof.

Let S be a structure in the image of canonical abstraction with the predicates from Table 7 and let u_1 and u_2 be two nodes in U^A. Furthermore, let X_1 and L_1 be the sets of variables such that the unary predicates that hold for u_1 are $x(v)$ for every $x \in X_1$ and $cul[x](v)$ for every $x \in L_1$, and let X_2 and L_2 be the sets of variables such that the unary predicates that hold for u_2 are $x(v)$ for every $x \in X_2$ and $cul[x](v)$ for every $x \in L_2$.

We consider the following sub-cases (the symmetric cases are not discussed):

X_1 **and** X_2 **are non-empty.** If $X_1 \subseteq L_2$ it means that u_1 and u_2 represent the end-points of a maximal uninterrupted list. If there is no node in U^A such that $cul[x](v)$ holds for some $x \in X_1$ then the list is of length 1 and therefore $n^S(u_1, u_2) = 1$. Otherwise, the length of the list is greater than 1 and $n^S(u_1, u_2) = 0$.

X_1 **is empty,** X_2 **is non-empty, and** $eq^S(u_1, u_1) = 0$**.** If $L_1 \subseteq L_2$ it means that u_2 represents the last node of a maximal uninterrupted list containing the nodes represented by u_1. Therefore, $n^S(u_1, u_2) = eq^S(u_1, u_1)$ and $n^S(u_2, u_1) = 0$.. If $L_2 \subseteq L_1$ it means that u_2 represents the first node of a maximal uninterrupted list containing the nodes represented by u_1. Therefore, $n^S(u_2, u_1) = eq^S(u_1, u_1)$ and $n^S(u_1, u_2) = 0$. Otherwise, u_1 and u_2 represent nodes belonging to distinct uninterrupted list and so $n^S(u_1, u_2) = n^S(u_2, u_1) = 0$.

X_1 **and** X_2 **are both empty.** If $u_1 = u_2$ then $n^S(u_1, u_1) = eq^S(u_1, u_1)$. Otherwise, this means that u_1 and u_2 represent distinct uninterrupted lists and therefore $n^S(u_1, u_2) = n^S(u_2, u_1) = 0$.

□