Static Analysis of Lists by Combining Shape and Numerical Abstractions

Liqian Chen*, Renjian Li, Xueguang Wu, Ji Wang

National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China

Abstract

We present an approach in the framework of abstract interpretation to analyze list-manipulating programs by combining shape and numerical abstractions. The analysis automatically divides a list into non-overlapping list segments according to the reachability property of pointer variables to list nodes. The list nodes in each segment are abstracted by a bit-vector wherein each bit corresponds to a pointer variable and indicates whether the nodes can be reached by that pointer variable. Moreover, for each bit-vector, we introduce an auxiliary integer variable, namely a counter variable, to record the number of nodes in the segment abstracted by that bit-vector. On this basis, we leverage the power of numerical abstractions to discover numerical relations among counter variables, so as to infer relational length properties among list segments. Furthermore, we show how our approach works for circular lists. Our approach stands out in its ability to find intricate properties that involve both shape and numerical information, which are important for checking program properties such as memory safety. A prototype is implemented and preliminary experimental results are encouraging.

Keywords: Static analysis, Abstract interpretation, Lists, Abstract domains, Shape analysis

1. Introduction

Invariants involving both shape and numerical information are crucial for checking nontrivial program properties in heap manipulating programs, such as mem-

*Corresponding author
Email address: lqchen@nudt.edu.cn (Liqian Chen)

ory safety, termination, bounded size of heap memory. However, automatically inferring such invariants is challenging, especially for programs manipulating dynamic linked data structures. In this paper, we consider the problem of analyzing programs manipulating lists. And inferring such invariants over lists requires considering both the shape of lists and the numerical information over the number of the list nodes.

Shape analysis provides a powerful approach to generate shape invariants, and much progress has been achieved in shape analysis in the past two decades [1][2]. However, shape analysis itself has limited ability in inferring non-trivial properties involving numerical information such as "sum of traversed and remained list sizes equals to the input list size". On the other hand, numerical static analysis by abstract interpretation [3] is widely adopted to automatically generate numerical invariants for programs. However, most of existing abstract domains focus on purely numerical properties and thus are specific for analyzing numerical programs. A recent interesting trend is to combine these two techniques, using shape analysis to generate shape invariants and using numerical abstract domains to track numerical relationships [4][5][6][7]. The key technical issue here is how to interact effectively between the shape aspect and the numerical aspect. Although several generic frameworks for the combination have been proposed [4][6], tighter bidirectional coupling between the two aspects still needs further considerations for the selected shape abstraction and numerical abstraction. For instance, shape abstraction needs to be enhanced to support numerical aspects while numerical abstraction also needs to be adapted with respect to the semantics of shape abstraction. And transfer functions for the combined domain should be designed by taking into account both the shape and numerical information at the same time.

In this paper, we present an approach in the framework of abstract interpretation to combine shape and numerical abstractions for analyzing programs manipulating lists. First, for the shape of a list, we propose a lightweight shape abstraction based on bit-vectors, upon the insight that list nodes in a list can be naturally grouped into non-overlapping list segments according to the reachability property of pointer variables to list nodes. Each list segment which includes those list nodes that can be reached by the same set of pointer variables, is abstracted by one bit-vector wherein each bit corresponds to a pointer variable in the program. From the numerical aspect, in order to track the number of list nodes in each list segment, we introduce an auxiliary (nonnegative) integer counter variable for each bit-vector. And we apply numerical abstract domains to infer numerical relations among counter variables. Then, transfer functions for the combined domain are constructed in terms of transfer functions of the numerical domain upon the

semantics of the shape abstraction. Specifically, in this paper we instantiate our approach by using a combination of intervals [8] and affine equalities [9] to conduct numerical abstractions. For the sake of better understanding, we first focus on non-circular lists and then consider extensions for circular lists. On this basis, a prototype is implemented and preliminary experimental results are presented on benchmark programs.

This paper is an extended version of our SAC 2013 paper in the Software Verification and Testing track [10]. On top of [10], we have added formal descriptions of maintaining points-to sets for pointer variables (Section 5.2). We have extended our approach to fit for circular lists (Section 6). We have conducted more experiments over programs manipulating circular lists and programs that may cause memory errors (Section 7). We have also provided details of proofs (in the Appendix).

The rest of the paper is organized as follows. Section 2 describes a simple list-manipulating programming language. Section 3 presents a shape abstraction approach for non-circular lists based on bit-vectors. Section 4 presents a combined domain of intervals and affine equalities to conduct numerical abstractions over counter variables. Section 5 shows how to perform analysis of programs manipulating non-circular lists based on the proposed abstractions. Section 6 extends the approach to fit for circular lists. Section 7 presents our prototype implementation together with preliminary experimental results. Section 8 discusses related work before Section 9 concludes.

2. List-Manipulating Programming Language

We first present a small language that manipulates lists. The syntax of our language is depicted in Fig. 1. It is a simple procedure-less sequential language with dynamic allocation and deallocation but no recursion. There is only one type of variables, i.e., pointer variables of LIST type, denoted as *PVar*. For the sake of simplicity, we first focus on non-circular singly-linked list.

The structure for list nodes contains a *next* field pointing to the successive list node, while all other fields are considered as data fields. The data fields are ignored in this paper, since we assume that operations over data fields have no influence on the shape of lists. We assume that there is at most one *next* operator in a statement and a pointer variable appears at most once in an assignment statement. All other cases could be transformed into this form by introducing temporary variables.

$$
\begin{array}{rcl}
p, q \in PVar & & \\
AsgnStmnt & := & p := null \mid p := q \mid p := q \rightarrow next \mid \\
 & & p \rightarrow next := null \mid p \rightarrow next := q \mid \\
 & & p := \text{malloc}() \mid \text{free}(p) \\
Cond & := & p == q \mid p == null \mid \neg Cond \mid \\
 & & Cond_1 \vee Cond_2 \mid Cond_1 \wedge Cond_2 \mid \\
 & & \text{true} \mid \text{false} \mid \text{brandom} \\
BranchStmnt & := & \textbf{if } Cond \textbf{ then } \{Stmnt;\}^* \textbf{ [else } \{Stmnt;\}^* \textbf{] fi} \\
WhileStmnt & := & \textbf{while } Cond \textbf{ do } \{Stmnt;\}^* \textbf{ od} \\
Stmnt & := & AsgnStmnt \mid BranchStmnt \mid WhileStmnt \\
Program & := & \{Stmnt;\}^*
\end{array}
$$

Figure 1: Syntax of a list-manipulating program

3. Shape Abstraction for Non-Circular Lists

First, we recall the definition of classic shape graph that is a graph used to represent the allocated memory in heap.

Definition 1. A shape graph for lists is a tuple $SG = \langle N, V, E \rangle$, where:

- N denotes the set of pointer variables and list nodes, and we utilize N_{nil} to denote $N \cup \{NULL\}$,
- $V \subseteq N$ denotes the set of pointer variables in the program,
- $E \subseteq N \times (N_{nil} - V)$ denotes the set of edges, which describes the points-to relations of pointer variables as well as successive relations between list nodes through the "*next*" field.

From the above definition, we can see that when using a standard shape graph to describe lists, we have to name explicitly all list nodes and store all the successive relations between nodes. Hence, using shape graphs may cause heavy memory costs. To this end, we propose a lightweight approach to encode the shape information contained in a shape graph. First, we introduce a binary predicate $Reach(n, n')$ to describe the reachability property between two nodes $n, n' \in N$:

$$
\begin{array}{rcl}
Reach(n, n') & \triangleq & \exists k \in \mathbb{N}. \forall 0 \leq i \leq k. n_i \in N \wedge n_0 = n \wedge n_k = n' \wedge \\
 & & \forall 0 \leqslant j < k. \langle n_j, n_{j+1} \rangle \in E
\end{array}
$$

Obviously, $Reach(n, n') = true$ holds if and only if there exists a path from n to n' in the shape graph. We maintain a variable ordering for all pointer variables V in the program and use V_i to denote the i-th variable in V where $0 \leq i \leq |V| - 1$.

Definition 2. For each node $n \in (N - V)$, we define a so-called *Variable Reachability Vector (VRV)* $\mathbf{vec}_n \in \{0,1\}^{|V|}$ that is a bit-vector of length $|V|$, where

$$\mathbf{vec}_n[i] = 1 \quad \textit{iff} \quad \textit{Reach}(V_i, n) = \textit{true}$$

We say V_i *reaches* list node n (or VRV $\mathbf{vec}_n$) if $\textit{Reach}(V_i, n) = \textit{true}$. We use bit-vector $\mathbf{0}$ as the VRV for those list nodes that cannot be reached by any pointer variables. Let Γ denote the set of VRVs for all list nodes $n \in (N - V)$. For every $\mathbf{vec} \in \Gamma$, let $\mathcal{I}_{\mathbf{vec}}$ denote the set of the 1-bits in $\mathbf{vec}$: $\mathcal{I}_{\mathbf{vec}} \triangleq \{i \in \mathbb{N} \mid \mathbf{vec}[i] = 1\}$. In other words, $\mathcal{I}_{\mathbf{vec}}$ describes the set of the indices of those pointer variables that can reach $\mathbf{vec}$. If $i \in \mathcal{I}$, it means that V_i can reach $\mathbf{vec}$ (and the corresponding nodes). We use $\Gamma_i \triangleq \{\mathbf{vec} \mid \mathbf{vec}[i] = 1\}$ to denote the set of VRVs that the variable V_i can reach. In fact, Γ describes the reachability properties of all pointer variables to list nodes. Each VRV $\mathbf{vec}_n$ can be considered as an abstract node that represents the set of nodes which can be reached by the same set of pointer variables as node n. By default, we use the lexicographic order as the variable ordering in VRVs throughout this paper, such as $\flat^v\flat^u\flat^q\flat^p$ where $\flat \in \{0,1\}$ (which means that p corresponds to the rightest bit in the bit vector according to the lexicographic order $p \prec q \prec u \prec v$)

Example 1. For the shape graph shown in Fig. 2 (a), suppose the variable ordering is $p \prec q \prec u \prec v$ (which means that the bit vectors are in the form of $\flat^v\flat^u\flat^q\flat^p$). Then the VRVs for this shape graph are shown in Fig. 2 (b). And we have $\Gamma = \{0011, 0100, 0111, 1111\}$; $\mathcal{I}_{0011} = \{0,1\}, \mathcal{I}_{0100} = \{2\}, \mathcal{I}_{0111} = \{0,1,2\}, \mathcal{I}_{1111} = \{0,1,2,3\}$; $\Gamma_0 = \{0011, 0111, 1111\}$, $\Gamma_1 = \{0011, 0111, 1111\}$, $\Gamma_2 = \{0111, 1111\}$, $\Gamma_3 = \{1111\}$.

Definition 3. Given two non-zero VRVs $\mathbf{vec}_1, \mathbf{vec}_2$,

- if $\mathcal{I}_{\mathbf{vec}_1} \subseteq \mathcal{I}_{\mathbf{vec}_2}$, we say $\mathbf{vec}_1$ can reach $\mathbf{vec}_2$, denoted as $\mathbf{vec}_1 \subseteq \mathbf{vec}_2$,
- if $\mathcal{I}_{\mathbf{vec}_1} \subset \mathcal{I}_{\mathbf{vec}_2}$, we say $\mathbf{vec}_1$ can strictly reach $\mathbf{vec}_2$, denoted as $\mathbf{vec}_1 \subset \mathbf{vec}_2$,
- if $\mathcal{I}_{\mathbf{vec}_1} \cap \mathcal{I}_{\mathbf{vec}_2} = \emptyset$, we say $\mathbf{vec}_1$ and $\mathbf{vec}_2$ cannot reach each other, denoted as $\mathbf{vec}_1 \cap \mathbf{vec}_2 = \emptyset$.

For the example shown in Fig. 2, we have: $\mathbf{vec}_{0011} \subset \mathbf{vec}_{0111}$; $\mathbf{vec}_{0100} \subset \mathbf{vec}_{0111}$; $\mathbf{vec}_{0100} \cap \mathbf{vec}_{0011} = \emptyset$.

Theorem 1. *Given two list nodes n_1, n_2 such that $\mathbf{vec}_{n_1} \neq \mathbf{vec}_{n_2}$ and $\mathbf{vec}_{n_1} \neq \mathbf{0}$, there exists one path from n_1 to n_2 if and only if $\mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2}$ holds.*

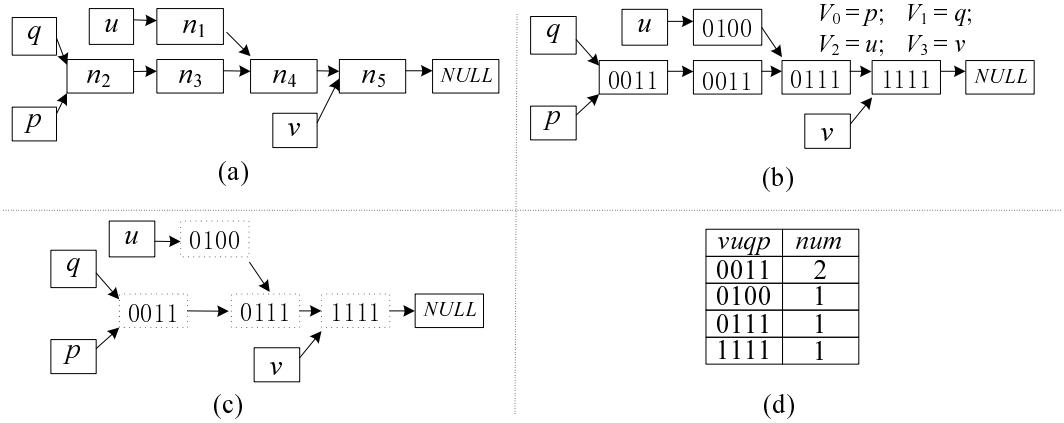

Figure 2: Example of variable reachability vectors for non-circular lists

PROOF. Given in the appendix. □

The bitwise set relations among VRVs implicitly characterize the reachability relations among nodes. All VRVs in Γ_i form a total order over $\subseteq$. Let $\mathbf{vec}_i^0$ denote the minimum element in Γ_i, then $\mathbf{vec}_i^0$ represents the VRV of the list node that variable V_i directly points to. For the example in Fig. 2, we have $\Gamma_0 = \{0011, 0111, 1111\}$, from which we can see that p directly points to 0011, since 0011 is the minimum element in Γ_0. Furthermore, from $\Gamma = \{0011, 0100, 0111, 1111\}$, we can see that

- p, q are aliases, since in each VRV from Γ the bit corresponding to p is 1 if and only if the bit corresponding to q is 1;
- p cannot reach the node that is directly pointed to by u, since the bit corresponding to p in the minimum element of Γ_2 (i.e., 0100) is 0.

Definition 4. A set of VRVs Γ is *consistent*, denoted as $wf(VRV(x))$, if for arbitrary two distinct VRVs $\mathbf{vec}_{n_1}, \mathbf{vec}_{n_2} \in \Gamma$, $\mathbf{vec}_{n_1} \cap \mathbf{vec}_{n_2} = \emptyset \vee \mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2} \vee \mathbf{vec}_{n_2} \subset \mathbf{vec}_{n_1}$ holds.

Theorem 2. *The set of VRVs of a singly-linked list is consistent.*

PROOF. Given in the appendix. □

Theorem 3. *A consistent set of VRVs* Γ *satisfies* $|\Gamma| \leq 2|V|$.

Proof. Given in the appendix. □

Definition 5. A set of VRVs with Counters (VRVCs) $\Gamma^+ \subseteq \Gamma \times \mathbb{N}$ is defined as a set of 2-tuples $\langle \mathbf{vec}, num \rangle$ where $\mathbf{vec} \in \Gamma$, $num \in \mathbb{N}$ standing for the number of the list nodes whose VRV is **vec**.

Example 2. For the shape graph shown in Fig. 2 (b), the VRVCs for it is $\Gamma^+ = \{\langle 0011, 2\rangle, \langle 0100, 1\rangle, \langle 0111, 1\rangle, \langle 1111, 1\rangle\}$, as shown in Fig. 2 (d).

The (consistent) set of VRVs with counters provides an exact abstraction for the shape of lists when ignoring the data contents. The list nodes are abstracted via VRVs (i.e., the **vec** component), the edges (i.e., the successive relations between nodes) are abstracted via the implicit bitwise subset relations of VRVs, and the number of the nodes that are reachable by the same set of pointer variables are described by the counters (i.e., the *num* component).

Utilizing bit vectors (i.e., VRVs) to represent the shape of linked lists brings a clear benefit in terms of memory. Also, shape operations over VRVs can benefit a lot from the efficient bitwise operations in terms of efficiency. E.g., checking whether the list segment abstracted by $\mathbf{vec}_1$ can reach the list segment abstracted by $\mathbf{vec}_2$ (i.e., checking $\mathbf{vec}_1 \subseteq \mathbf{vec}_2$) can be cheaply implemented via the bitwise AND operation by checking whether it holds that $\mathbf{vec}_1 \,\&\, \mathbf{vec}_2 = \mathbf{vec}_1$. Checking whether only one of two variables p and q will reach the list segment abstracted by **vec** can be implemented by checking whether it holds that $(((\mathbf{vec} >> p) \oplus (\mathbf{vec} >> q)) \,\&\, \mathtt{0x1}) = \mathtt{0x1}$.

4. Numerical Abstraction over Counters

For each $\mathbf{vec} \in \Gamma$, we introduce an auxiliary counter variable $t^{\mathbf{vec}} \in \mathbb{N}$ to denote the value of the corresponding *num* component (i.e., the number of the list nodes whose VRV is **vec**) of VRVCs. We maintain a bijection between **vec** and $t^{\mathbf{vec}}$. For each $t^{\mathbf{vec}}$, we use $VEC(t^{\mathbf{vec}})$ to obtain its corresponding bit vector **vec**. Furthermore, we introduce a special auxiliary variable $t^{0\ldots00} \in \mathbb{N}$ to specify memory leak (i.e., $t^{0\ldots00} > 0$). We use a lexicographic ordering on counter variables: $t^{0\ldots00} \prec t^{0\ldots01} \prec t^{0\ldots10} \prec \cdots \prec t^{1\ldots11}$. And $\{\langle \mathbf{vec}, t^{\mathbf{vec}}\rangle \mid t^{\mathbf{vec}} > 0\}$ represents the shape of a list, if it is consistent.

Since counter variables $t^{\mathbf{vec}} \in \mathbb{N}$ are numerical variables, we could leverage numerical abstraction techniques over $t^{\mathbf{vec}}$. In this paper, we present an abstract domain, namely the $\mathcal{CD}$ domain, to perform numeric abstraction over counter variables, which combines the interval abstract domain [8] and the affine equality

abstract domain [9]. If the program has k pointer variables, we need introduce 2^k auxiliary counter variables. We choose intervals and affine equalities to construct the $\mathcal{CD}$ domain, because they are cheap in both time and memory, and bounds as well as equality relations are important for list-manipulating programs. However, it is also worth noting that according to Theorem 3, most auxiliary counter variables equal to 0 and only linear number of counter variables with respect to $|V|$ need to be tracked. Hence, for the sake of memory, we omit constraints $t^{\mathbf{vec}} = 0$ in the constraint representation of the abstract domain. In other words, for those counter variables $t^{\mathbf{vec}}$ that are not involved in the constraint system, we have implicit information that $t^{\mathbf{vec}} = 0$.

4.1. Representation

We use intervals to track the range information of each counter variable $t^{\mathbf{vec}} \in \mathbb{N}$, and use affine equalities to track the relational information among those counter variables. Hence, each domain element P in the $\mathcal{CD}$ domain is described as an affine system $Ax = b$ in reduced row echelon form together with bounds for counter variables $x \in [c, d]$, where $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n, c \in \mathbb{N}^n, d \in \{\mathbb{N}, +\infty\}^n, 0 \leq c \leq d$. It represents the set $\gamma(P) = \{x \in \mathbb{N}^n \mid Ax = b, c \leq x \leq d\}$ where each point x is a possible environment (or state), i.e., an assignment of nonnegative integer values to counter variables. For the sake of convenience, we use $EQS(P)$ to denote the affine equality part, and $ITV(P)$ to denote the interval part from the domain representation.

Given each $x \in \gamma(P)$, we can derive a set of VRVs from x: $VRV(x) = \{VEC(x_j) \mid x_j \geq 1\}$, as well as a set of VRVs with counters $VRVC(x) = \{\langle VEC(x_j), x_j\rangle \mid x_j \geq 1\}$. Obviously, if $VRV(x)$ is consistent (i.e., $wf(VRV(x))$), then $VRVC(x)$ describes a shape of singly-linked lists. And we use $\bar{\gamma}(P)$ to denote the obtained consistent set of VRVCs:

$$\bar{\gamma}(P) = \{VRVC(x) \mid x \in \gamma(P) \wedge wf(VRV(x))\}$$

$\bar{\gamma}(P)$ can be constructed from $\gamma(P)$ by considering the consistency among VRVs.

Example 3. Consider the program fragment

```
traverse(q) {
① p := q → next;
② while (p ≠ null) do {
③       p := p → next; } od
}
```

Assume q points to a list with length 9 before calling $traverse(q)$ and the variable

ordering is $p \prec q$ (which means that the bit vectors are in the form of $\flat^q\flat^p$). Then in the first iteration, at program point ③, we obtain the $\mathcal{CD}$ domain element $P = \{t^{10} = 1, t^{11} = 8; t^{10} \in [1,1], t^{11} \in [8,8]\}$. Then we know that $\bar{\gamma}(P)$ contains only one possible list shape: $\bar{\gamma}(P) = \{\{\langle 10, 1\rangle, \langle 11, 8\rangle\}\}$.

4.2. Domain Operations over Counters

In the $\mathcal{CD}$ domain, most domain operations (such as meet, join, inclusion, etc.) over counter variables can be directly constructed by combining the corresponding domain operations of the interval abstract domain and those of the affine equality abstract domain. E.g., the join operation for a control-flow join in the $\mathcal{CD}$ domain is defined as

$$P \sqcup P' \stackrel{\text{def}}{=} (EQS(P) \sqcup_{EQS} EQS(P')) \wedge (ITV(P) \sqcup_i ITV(P'))$$

where $\sqcup_{EQS}$ denotes the join operation in the affine equality domain which can be implemented via computing the affine hull of $EQS(P)$ and $EQS(P')$ and $\sqcup_i$ denotes the interval union. The inclusion test operation in the $\mathcal{CD}$ domain is defined as

$$P \sqsubseteq P' \iff (EQS(P) \sqcap EQS(P') = EQS(P)) \wedge (ITV(P) \sqsubseteq_i ITV(P'))$$

Bound tightening. In the $\mathcal{CD}$ domain, the bounds of each variable can be obtained from the ITV part of the domain element. The bounds may be changed during domain operations of the affine equalities. E.g., when an affine equality is added, the bounds of variables need to be updated. In this paper, we use bound propagation technique to tighten the bounds.

In fact, each affine equality from the EQS part of the $\mathcal{CD}$ element can be used to tighten the bounds for those variables occurring in the equality. E.g., given an equality $\Sigma_i a_i x_i = b$, if $a_i > 0$, a new candidate lower bound for x_i comes from: $\underline{x}'_i = \lceil (b - \Sigma_{j \neq i} a_j \dot{x}_j)/a_i \rceil$ where $\dot{x}_j = a_j > 0 \,?\, \overline{x}_j : \underline{x}_j$, and a new candidate upper bound for x_i comes from: $\overline{x}'_i = \lfloor (b - \Sigma_{j \neq i} a_j \ddot{x}_j)/a_i \rfloor$ where $\ddot{x}_j = a_j > 0 \,?\, \underline{x}_j : \overline{x}_j$. If the new candidate bounds are tighter, then x_i's bounds are updated. This process can be repeated for each variable in that equality and for each equality in the EQS system. Since in the $\mathcal{CD}$ domain all counter variables are nonnegative integers, the convergence of the iterations of bound tightening is always attained in a finite number of steps. In practice, we employ interval Gauss-Seidel method [11] to make use of affine equalities to tighten variable bounds.

Widening. The height of the lattice of affine equalities is finite, but intervals do not satisfy the ascending chain condition. Thus, to cope with loops, a widening

operator is needed to ensure the convergence of fixpoint computations over the $\mathcal{CD}$ domain. Given two $\mathcal{CD}$ elements P, P' satisfying $P \sqsubseteq P'$,

$$P \nabla P' \stackrel{\text{def}}{=} EQS(P') \sqcap (ITV(P) \nabla_i ITV(P')).$$

Since $t^{\mathbf{vec}} \geq 0$ always holds, we refine the widening over intervals as:

$$[a,b]\nabla_i[c,d] \stackrel{\text{def}}{=} \begin{cases} [a, b \geq d ? b : +\infty] & \text{if } a \leq c, \\ [1, b \geq d ? b : +\infty] & \text{else if } a > c \geq 1, \\ [0, b \geq d ? b : +\infty] & \text{otherwise.} \end{cases}$$

Here we take 1 as a special threshold for the interval widening, since pointer variable p may use $p \rightarrow next$ to access the successive node, which may cause null pointer dereference.

Furthermore, we use $EQS(P')$ to tighten the bounds of variables obtained by $ITV(P)\nabla_i ITV(P')$ after widening. To avoid the well-known convergence problem of interaction between reduction and widening [12], we perform bound tightening over the widening results only for finite times overall.

In Algorithm 1, we show our fixpoint iteration algorithm for loops used in the $\mathcal{CD}$ domain, which is based on the classic fixpoint iteration algorithm in abstract interpretation [3]. The algorithm takes three input parameters: 1) P_0: the initial abstract $\mathcal{CD}$ element before executing the loop; 2) **while** b **do** {stmts} **od**: a while loop with a boolean loop condition b and loop body {stmts} which represents a sequence of statements such as assignments and conditional branches; 3) K: a positive integer threshold of the times of applying bound tightening over the widening results. In each iteration of Algorithm 1, P denotes the $\mathcal{CD}$ element at the loop head before the i-th execution of the loop body, and P' denotes the resulting $\mathcal{CD}$ element at the loop head after the i-th execution of the loop body. In each iteration, line 6 corresponds to the test transfer function handling the loop condition. Line 7 corresponds to the post-image transfer function handling the loop body, which computes the resulting abstract element after the loop body given the abstract element before. In line 8, we join together the $\mathcal{CD}$ element P at the loop head before the i-th execution of the loop body with the resulting element Q' after the i-th execution, and then widen it against P. From line 9 to 13, we apply bound tightening (denoted by the *BoundTightening*() function in Algorithm 1) over the widening result if not exceeding the threshold of the times of applying bound tightening over the widening results.

The fixpoint iteration algorithm shown in Algorithm 1 is guaranteed to terminate in finite time. The lattice of affine equalities is of finite height, and thus the

Algorithm 1 The fixpoint iteration algorithm for loops in the $\mathcal{CD}$ domain

Input: P_0: initial abstract element before executing the loop;
while b **do** {stmts} **od**: the loop program;
K: threshold of the times of applying bound tightening over widening results.
Output: P': the stable abstract element at the loop head.

1: $i \leftarrow 0$
2: $j \leftarrow 0$
3: $P' \leftarrow P_0$
4: **repeat**
5: $P \leftarrow P'$
6: $Q \leftarrow P \sqcap b$
7: $Q' \leftarrow post[\text{stmts}]Q$
8: $P' \leftarrow P \nabla (P \sqcup Q')$
9: **if** $(j < K)$ **then**
10: $R \leftarrow BoundTightening(P')$
11: **if** $(R \neq P')$ **then**
12: $P' \leftarrow R$
13: $j \leftarrow j + 1$
14: $i \leftarrow i + 1$
15: **until** $P' \sqsubseteq P$
16: **return** P'

affine equality part of the $\mathcal{CD}$ domain will become stable with no need of widening. And in the worst case, after K times of applying bound tightening over the widening results, there is no more reduction of using affine equalities to tighten interval bounds. Then the interval part of the $\mathcal{CD}$ domain will become stable due to usage of interval widening.

Example 4. Consider again the program fragment shown in Example 3. Assume q points to a list with length 9 before calling *traverse*(q) and the variable ordering is $p \prec q$ (which means that the bit vectors are in the form of $\mathfrak{b}^q\mathfrak{b}^p$). At program point ②, we get before the first loop iteration the $\mathcal{CD}$ element $P_0 = \{t^{10} = 1, t^{11} = 8; t^{10} \in [1,1], t^{11} \in [8,8]\}$ and after the first loop iteration $Q_1 = \{t^{10} = 2, t^{11} = 7; t^{10} \in [2,2], t^{11} \in [7,7]\}$. Since the loop head is also a control flow join, we compute the join of P_0 and Q_1

$$P_0 \sqcup Q_1 = \{t^{10} + t^{11} = 9; t^{10} \in [1,2], t^{11} \in [7,8]\}$$

After that, we apply the widening operation at loop head ②, and get

$$P_1 = P_0 \nabla (P_0 \sqcup Q_1) = \{t^{10} + t^{11} = 9; t^{10} \in [1, +\infty], t^{11} \in [1, 8]\}$$
$$= \{t^{10} + t^{11} = 9; t^{10} \in [1, 8], t^{11} \in [1, 8]\}$$

After executing the loop body for a second time, we will get $Q_2 = \{t^{10} + t^{11} = 9; t^{10} \in [2, 9], t^{11} \in [0, 7]\}$. Then we compute the join of P_1 and Q_2

$$P_1 \sqcup Q_2 = \{t^{10} + t^{11} = 9; t^{10} \in [1, 9], t^{11} \in [0, 8]\}$$

After that, we apply again the widening operation at loop head ②, and get

$$P_2 = P_1 \nabla (P_1 \sqcup Q_2) = \{t^{10} + t^{11} = 9; t^{10} \in [1, +\infty], t^{11} \in [0, 8]\}$$
$$= \{t^{10} + t^{11} = 9; t^{10} \in [1, 9], t^{11} \in [0, 8]\}$$

After executing the loop body for a third time, we will found that $P_3 = P_2$, which means that the fixpoint iteration becomes stable and the found invariant is $\{t^{10} + t^{11} = 9; t^{10} \in [1, 9], t^{11} \in [0, 8]\}$.

5. Analysis of Programs Manipulating Non-Circular Lists

In this section, we show how to combine the shape abstraction (in Sect. 3) and the numerical abstraction (in Sect. 4) to analyze programs manipulating non-circular lists written in the syntax described in Sect. 2. For the domain representation, we keep in memory the constraints over $t^{\mathbf{vec}}$s. However, the condition test and assignment statements in the syntax of list-manipulating programs (as shown in Fig. 1) manipulate pointer variables rather than $t^{\mathbf{vec}}$s. Hence, we need to transform the abstract semantics of test and assignment transfer functions on pointer variables into abstract semantics of transfer functions on $t^{\mathbf{vec}}$s. In other words, we transform the abstract semantics of list-manipulating programs over pointer variables into that of numerical programs over counter variables.

5.1. Transfer Function over Non-Circular Lists

Test transfer function over non-circular lists. In this paper, we consider only four basic kinds of test conditions over pointer variables: $p == null$, $p == q$, $p \neq null$, $p \neq q$. Other complex conditions can be obtained by introducing auxiliary pointer variables and composing basic conditions via logical operators. Let P be the input $C\mathcal{D}$ element before and P' be the resulting $C\mathcal{D}$ element after applying the transfer function.

1. $[\![p == null]\!]^\sharp$: When $p == null$ holds, it means that pointer variable p does not point to any lists and thus cannot reach any list nodes, i.e.,

$$\forall t^{\mathbf{vec}}.VEC(t^{\mathbf{vec}})[\mathcal{I}_p] = 1 \rightarrow t^{\mathbf{vec}} = 0$$

   where $\mathcal{I}_p$ denotes the index of the bit corresponding to pointer variable p. In the $\mathcal{CD}$ domain, we add constraints $t^{\mathbf{vec}} = 0$ to P for those $t^{\mathbf{vec}}$ satisfying $VEC(t^{\mathbf{vec}})[\mathcal{I}_p] = 1$. Then we check the emptiness of P', and tighten variable bounds.
2. $[\![p == q]\!]^\sharp$: When $p == q$ holds, it means that pointer variables p, q are aliases. Hence,

$$\forall t^{\mathbf{vec}}.VEC(t^{\mathbf{vec}})[\mathcal{I}_p] \oplus VEC(t^{\mathbf{vec}})[\mathcal{I}_q] = 1 \rightarrow t^{\mathbf{vec}} = 0$$

   where $\oplus$ denotes the bitwise XOR operation. In the $\mathcal{CD}$ domain, we add constraints $t^{\mathbf{vec}} = 0$ to P for those $t^{\mathbf{vec}}$ satisfying $VEC(t^{\mathbf{vec}})[\mathcal{I}_p] \oplus VEC(t^{\mathbf{vec}})[\mathcal{I}_q] = 1$. Then we check the emptiness of P' and tighten variable bounds.
3. $[\![p \neq null]\!]^\sharp$: When $p \neq null$ holds, it means that pointer variable p does point to some list node. Hence,

$$\sum_{\mathbf{vec}[\mathcal{I}_p]=1} t^{\mathbf{vec}} \geq 1$$

   In the $\mathcal{CD}$ domain, if $t^{\mathbf{vec}} = 0$ holds for all $\mathbf{vec}$ satisfying $\mathbf{vec}[\mathcal{I}_p] = 1$, we put $P' = \bot$. Otherwise, we use the constraint $\sum_{\mathbf{vec}[\mathcal{I}_p]=1} t^{\mathbf{vec}} \geq 1$ to tighten variable bounds.
4. $[\![p \neq q]\!]^\sharp$: When $p \neq q$ holds, it means that there exists at least one list node that p and q do not point to at the same time. Hence,

$$\sum_{\mathbf{vec}[\mathcal{I}_p] \oplus \mathbf{vec}[\mathcal{I}_q]=1} t^{\mathbf{vec}} \geq 1$$

   In the $\mathcal{CD}$ domain, if $t^{\mathbf{vec}} = 0$ holds for all $\mathbf{vec}$ satisfying $\mathbf{vec}[\mathcal{I}_p] \oplus \mathbf{vec}[\mathcal{I}_q] = 1$,we put $P' = \bot$. Otherwise, we use $\sum_{\mathbf{vec}[\mathcal{I}_p] \oplus \mathbf{vec}[\mathcal{I}_q]=1} t^{\mathbf{vec}} \geq 1$ to tighten the variable bounds.

Assignment transfer function over non-circular lists. We consider the assignment transfer functions in the form of $P' = [\![astmt]\!]^\sharp(P)$, where *astmt* denotes an assignment statement of shapes. Let $\mathbf{vec}/_{\mathcal{I}\leftarrow 0}$ denote the bitwise substitution of those bits in $\mathcal{I}$ with value 0, $\mathbf{vec}/_{\mathcal{I}\leftarrow q}$ denote the bitwise substitution of those bits

$[\![p := null]\!]^\sharp$	Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\{p\}\leftarrow 0}$. For each $\mathbf{vec} \in \Gamma$ such that $\mathbf{vec}' \neq \mathbf{vec}$, we build numerical statements: **if** $(t^{\mathbf{vec}} \geq 1)$ **then** $\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$ **fi**
$[\![p := \text{malloc}()]\!]^\sharp$	① First, we apply $[\![p := null]\!]^\sharp$. ② Let $\mathbf{vec}' \overset{\text{def}}{=} 0/_{\{p\}\leftarrow 1}$. We build numerical statements: $\{\ t^{\mathbf{vec}'} := 1;\ \}$.
$[\![\text{free}(p)]\!]^\sharp$	Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{I_{\mathbf{vec}_p^0}\leftarrow 0}$. For each $\mathbf{vec} \in \Gamma$ such that $(\mathbf{vec}\ \&\ \mathbf{vec}_p^0) = \mathbf{vec}_p^0$, • if $\mathbf{vec} = \mathbf{vec}_p^0$, we build numerical statements: **if** $(t^{\mathbf{vec}_p^0} \geq 1)$ **then** $\{\ t^{0\ldots0} := t^{0\ldots0} + t^{\mathbf{vec}_p^0} - 1;\ t^{\mathbf{vec}_p^0} := 0;\ \}$ **fi** • otherwise, we build numerical statements: **if** $(t^{\mathbf{vec}} \geq 1)$ **then** $\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$ **fi**
$[\![p := q]\!]^\sharp$	Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\{p\}\leftarrow q}$. For each $\mathbf{vec} \in \Gamma$ such that $\mathbf{vec}' \neq \mathbf{vec}$, we build numerical statements: **if** $(t^{\mathbf{vec}} \geq 1)$ **then** $\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$ **fi**
$[\![p := q \rightarrow next]\!]^\sharp$	① First, we apply $[\![p := null]\!]^\sharp$. ② Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\{p\}\leftarrow q}$. For each $\mathbf{vec} \in \Gamma$ such that $\mathbf{vec}_{\{pq\}} = 01$, • if $\mathbf{vec} = \mathbf{vec}_q^0$, we build numerical statements **if** $(t^{\mathbf{vec}} \geq 1)$ **then** $\{t^{\mathbf{vec}'} := t^{\mathbf{vec}} - 1; t^{\mathbf{vec}} := 1;\}$ **else** $\{P' := \top;\}$ **fi** • otherwise, we build numerical statements $\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$.
$[\![p \rightarrow next := null]\!]^\sharp$	Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{I_{\mathbf{vec}_p^0}\leftarrow 0}$. For each $\mathbf{vec}$ such that $(\mathbf{vec}\ \&\ \mathbf{vec}_p^0) = \mathbf{vec}_p^0$, • if $\mathbf{vec} = \mathbf{vec}_p^0$, we build numerical statements: **if**$(t^{\mathbf{vec}_p^0} \geq 1)$**then**$\{t^{0\ldots0} := t^{0\ldots0} + t^{\mathbf{vec}_p^0} - 1; t^{\mathbf{vec}_p^0} := 1;\}$**else**$\{P' := \top;\}$**fi** • otherwise, we build numerical statements: **if** $(t^{\mathbf{vec}} \geq 1)$ **then**$\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$ **fi**
$[\![p \rightarrow next := q]\!]^\sharp$	① First, we apply $[\![p \rightarrow next := null]\!]^\sharp$. ② Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{I_{\mathbf{vec}_p^0}\leftarrow q}$. For each $\mathbf{vec} \in \Gamma$ such that $\mathbf{vec}[q] = 1$ and $\mathbf{vec}' \neq \mathbf{vec}$, we build numerical statements: **if** $(t^{\mathbf{vec}} \geq 1)$ **then** $\{t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\}$ **fi**

Figure 3: Abstract assignment transfer functions over non-circular lists

in $\mathcal{I}$ with the value of the corresponding bit of variable q, and $\mathbf{vec}_{pq}$ denote the projection of **vec** on positions of p and q. The abstract semantics of the assignment transfer function over shapes is shown in Fig. 3. The main idea here is to transform an assignment over shapes into a series of numerical statements over counter variables, according to the changing of the shape.

Example 5. Consider the assignment transfer function $[\![p := u]\!]^\sharp$ over the list shown in Example 1. Recall that the bit vectors are in the form of $\flat^v\flat^u\flat^q\flat^p$. As depicted in Fig. 4, before applying $[\![p := u]\!]^\sharp$, we have $P = \{t^{0011} = 2, t^{0100} = 1, t^{0111} = 1, t^{1111} = 1; t^{0011} \in [2,2], t^{0100} \in [1,1], t^{0111} \in [1,1], t^{1111} \in [1,1]\}$. According to the semantics of $[\![p := u]\!]^\sharp$, we know that the bit vector 0011 changes to 0010, and thus we construct the following numeric assignments: **if**($t^{0011} \geq 1$)**then**{ $t^{0010} := t^{0010} + t^{0011}$; $t^{0011} := 0$; }**fi**. Similarly, for the change from 0100 to 0101, we build numerical assignments: **if**($t^{0100} \geq 1$)**then**{ $t^{0101} := t^{0101} + t^{0100}$; $t^{0100} := 0$; }**fi**. Finally, after having applied all the above assignment transfer functions over counters, we will get $P' = \{t^{0010} = 2, t^{0101} = 1, t^{0111} = 1, t^{1111} = 1; t^{0010} \in [2,2], t^{0101} \in [1,1], t^{0111} \in [1,1], t^{1111} \in [1,1]\}$.

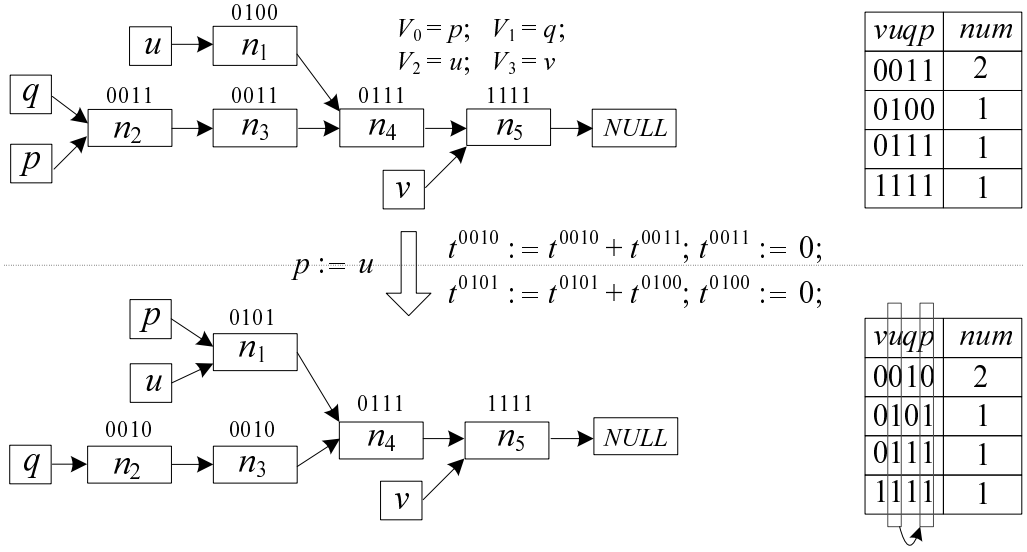

Figure 4: Example of an assignment transfer function over non-circular lists

5.2. Maintaining Points-to Sets over Non-Circular Lists

From the above, we see that the information of $\mathbf{vec}_p^0$ is important for transforming free() and list assignments that involve "*next*" field to numerical assignments.

$[\![p == null]\!]^{\sharp_0}$	① For all variables u, $\Gamma^0_u \leftarrow \Gamma^0_u/_{\{p\}\leftarrow 0}$; ② $\Gamma^0_p \leftarrow \emptyset$
$[\![p == q]\!]^{\sharp_0}$	$\Gamma^0_p, \Gamma^0_q \leftarrow \Gamma^0_p \cap \Gamma^0_q$
$[\![p \neq null]\!]^{\sharp_0}$	do nothing
$[\![p \neq q]\!]^{\sharp_0}$	If ($\lvert\Gamma^0_p\rvert = 1$) then $\Gamma^0_q \leftarrow \Gamma^0_q \setminus \Gamma^0_p$ else if ($\lvert\Gamma^0_q\rvert = 1$) then $\Gamma^0_p \leftarrow \Gamma^0_p \setminus \Gamma^0_q$
$[\![p := null]\!]^{\sharp_0}$	① For all variables u, $\Gamma^0_u \leftarrow \Gamma^0_u/_{\{p\}\leftarrow 0}$; ② $\Gamma^0_p \leftarrow \emptyset$
$[\![p := \text{malloc}()]\!]^{\sharp_0}$	① First, we apply $[\![p := null]\!]^{\sharp_0}$; ② $\Gamma^0_p \leftarrow \{\mathbf{0}/_{\{p\}\leftarrow 1}\}$
$[\![\text{free}(p)]\!]^{\sharp_0}$	① For each $\mathbf{vec}^0_p \in \Gamma^0_p$, $\mathbf{vec}^0_u \in \Gamma^0_u$ where $u \neq p$ do • if ($\mathbf{vec}^0_p == \mathbf{vec}^0_u$) $\Gamma^0_u \leftarrow \Gamma^0_u \setminus \{\mathbf{vec}^0_u\}$ • else if ($\mathbf{vec}^0_u[p] == 1$) $\Gamma^0_u \leftarrow \Gamma^0_u \setminus \{\mathbf{vec}^0_u\} \cup \{\mathbf{vec}^0_u/_{\mathcal{I}_{\mathbf{vec}^0_p}\leftarrow 0}\}$ ② $\Gamma^0_p \leftarrow \emptyset$
$[\![p := q]\!]^{\sharp_0}$	① First, we apply $[\![p := null]\!]^{\sharp_0}$; ② For all variables $u \neq p$, $\Gamma^0_u \leftarrow \Gamma^0_u/_{\{p\}\leftarrow\{q\}}$; ③ $\Gamma^0_p \leftarrow \Gamma^0_q$
$[\![p := q \rightarrow next]\!]^{\sharp_0}$	① First, we apply $[\![p := null]\!]^{\sharp_0}$; ② For each $\mathbf{vec}^0_q \in \Gamma^0_q$ • if $t^{\mathbf{vec}^0_q} = 1$ and there exists $\mathbf{vec}'$ which is the least vector satisfying $\mathbf{vec}^0_q \subseteq \mathbf{vec}' \wedge t^{\mathbf{vec}'} \geq 1$, then $\Gamma^0_p \leftarrow \Gamma^0_p \cup \{\mathbf{vec}'/_{\{p\}\leftarrow 1}\}$ • else if $t^{\mathbf{vec}^0_q} \geq 2$ then $\Gamma^0_p \leftarrow \Gamma^0_p \cup \{\mathbf{vec}^0_q/_{\{p\}\leftarrow 1}\}$; ③ For each $\mathbf{vec}^0_q \in \Gamma^0_q$, $\mathbf{vec}^0_u \in \Gamma^0_u$ satisfying $\mathbf{vec}^0_q \subset \mathbf{vec}^0_u \wedge u \neq p$ do $\Gamma^0_u \leftarrow \Gamma^0_u \setminus \{\mathbf{vec}^0_u\} \cup \{\mathbf{vec}^0_u/_{\{p\}\leftarrow 1}\}$
$[\![p \rightarrow next := null]\!]^{\sharp_0}$	For each $\mathbf{vec}^0_p \in \Gamma^0_p$, $\mathbf{vec}^0_u \in \Gamma^0_u$ satisfying $\mathbf{vec}^0_p \subset \mathbf{vec}^0_u \wedge u \neq p$ do $\Gamma^0_u \leftarrow \Gamma^0_u \setminus \{\mathbf{vec}^0_u\} \cup \{\mathbf{vec}^0_u/_{\mathcal{I}_{\mathbf{vec}^0_p}\leftarrow 0}\}$
$[\![p \rightarrow next := q]\!]^{\sharp_0}$	① First, we apply $[\![p \rightarrow next := null]\!]^{\sharp_0}$; ② For each $\mathbf{vec}^0_q \in \Gamma^0_q$, $\mathbf{vec}^0_u \in \Gamma^0_u$ satisfying $\mathbf{vec}^0_q \subseteq \mathbf{vec}^0_u$ do $\Gamma^0_u \leftarrow \Gamma^0_u \setminus \{\mathbf{vec}^0_u\} \cup \{\mathbf{vec}^0_u/_{\mathcal{I}_{\mathbf{vec}^0_p}\leftarrow 1}\}$

Figure 5: Transfer functions for maintaining points-to sets over non-circular lists

Recall that $\mathbf{vec}^0_p$ specifies the bit vector that the pointer variable p directly points to. In the concrete semantics, $\mathbf{vec}^0_p$ can be computed from the environment of auxiliary counter variables, i.e., the least bit vector $\mathbf{vec}$ such that $\mathbf{vec}[p] = 1 \wedge t^{\mathbf{vec}} > 0$. However, in the abstract semantics, due to precision loss, we may not have enough information to determine such a vector and may only know some $\mathbf{vec}$ satisfying $\mathbf{vec}[p] = 1 \wedge t^{\mathbf{vec}} \geq 0$. In this case, we have to consider the following 2 subcases:

- if $t^{\mathbf{vec}} > 0$, then $\mathbf{vec}^0_p = \mathbf{vec}$,
- else if $t^{\mathbf{vec}} = 0$, then $\mathbf{vec}^0_p$ is the least vector $\mathbf{vec}'$ satisfying $\mathbf{vec}'[p] = 1 \wedge t^{\mathbf{vec}'} > 0 \wedge \mathbf{vec} \subset \mathbf{vec}'$.

In other words, we may only know a possible set of $\mathbf{vec}^0_p$s from the abstract representation. Hence, during the analysis, we may get a set of possible $\mathbf{vec}^0_p$s, denoted as Γ^0_p, for each pointer variable p. We redefine the transfer functions based on Γ^0_p. The main idea here is to first apply the transfer function separately on each $\mathbf{vec}^0_p \in \Gamma^0_p$ and then perform join over the results computed from each $\mathbf{vec}^0_p$.

Furthermore, to improve the precision, we design another abstract domain to maintain Γ^0_p. For the control-flow join, we update Γ^0_p to the union of two resulting sets of Γ^0_p of the two branches. For condition tests and assignments, we update Γ^0_p according to the rules shown in Fig. 5.

6. Extension to Circular Lists

A circular singly-linked list contains a cycle in which the last node points to the first node via the *next* field. Note that the cycle of a circular singly-linked list is a directed cycle with respect to the "*next*" edge. However, if we simply use VRVs to describe the shape of a circular list (like we do in Sect. 3), the VRVs for all the list nodes on the same cycle are the same. Hence, we can not distinguish which node a pointer variable directly points to and which node is the first node on a cycle that a pointer variable reaches. In this section, we show how to solve this problem and extend our abstraction to fit for circular lists.

6.1. Shape Abstraction for Circular Lists

The main idea to extend shape abstraction based on bit-vectors to fit for circular lists is to cut the cycle in a circular list at some *next* edge. Then we use the VRVs to abstractly represent the resulting non-circular list after cut. Moreover, we add an auxiliary flag bit c for each VRV $\mathbf{vec}$ and denote the resulting VRV

Algorithm 2 OnSameCycle($\breve{\mathbf{vec}}$, $\breve{\mathbf{vec}}'$)

Input: $\breve{\mathbf{vec}}$: a circular VRV ;
$\breve{\mathbf{vec}}'$: another circular VRV ;
Output: b: whether $\breve{\mathbf{vec}}$ and $\breve{\mathbf{vec}}'$ are on the same cycle.

1: **if** ($\breve{\mathbf{vec}}[c] = 1 \wedge \breve{\mathbf{vec}}'[c] = 1 \wedge (\breve{\mathbf{vec}}' >> 1)\&(\breve{\mathbf{vec}} >> 1) \neq \mathtt{0x0}$)
2: **then** $b \leftarrow$ TRUE
3: **else** $b \leftarrow$ FALSE
4: **return** b

Algorithm 3 JoinCycl($\breve{\Gamma}$, $\breve{\mathbf{vec}}$)

Input: $\breve{\Gamma}$: a set of circular VRVs;
$\breve{\mathbf{vec}}$: a circular VRV in $\breve{\Gamma}$ s.t. $\breve{\mathbf{vec}}[c] = 0$;
Output: $\breve{join}$: the circular VRV of the first node on a cycle that $\breve{\mathbf{vec}}$ reaches.

1: $\breve{join} \leftarrow \mathtt{0x1}$
2: **for** each $\breve{\mathbf{vec}}' \in \breve{\Gamma}$ satisfying $\breve{\mathbf{vec}}'[c] = 1$ and $\breve{\mathbf{vec}} \subset \breve{\mathbf{vec}}'$ **do**
3: **if** ($\breve{join} = \mathtt{0x1} \vee \breve{\mathbf{vec}}' \subset \breve{join}$) **then** $\breve{join} \leftarrow \breve{\mathbf{vec}}'$
4: **return** $\breve{join}$

as a so-called *circular VRV* $\breve{\mathbf{vec}}$ where $\breve{\mathbf{vec}}[c] = 1$ if and only if the corresponding list segment is on a cycle. In this paper, we add the c bit as the lowest bit (i.e., $\mathcal{I}_c = 0$). The source node of the *next* edge that is cut is called the *tail* node of the cycle, while the target node of the *next* edge that is cut is called the *head* node of the cycle. It is easy to see that in the set of circular VRVs for those list nodes on the same cycle, the circular VRV of the *tail* node is the largest one while that of the *head* node is the smallest one. For the sake of conciseness, we introduce several algorithms to get commonly used shape information from the set of circular VRVs. We use OnSameCycle($\breve{\mathbf{vec}}$, $\breve{\mathbf{vec}}'$) to determine whether two circular VRVs $\breve{\mathbf{vec}}$ and $\breve{\mathbf{vec}}'$ are on the same cycle, shown in Algorithm 2. JoinCycl ($\breve{\Gamma}$, $\breve{\mathbf{vec}}$) computes the circular VRV of the node on a cycle that $\breve{\mathbf{vec}}$ reaches first, shown in Algorithm 3. Head ($\breve{\Gamma}$, $\breve{\mathbf{vec}}$) and Tail ($\breve{\Gamma}$, $\breve{\mathbf{vec}}$) respectively give the circular VRVs of the current *head* and *tail* node of the cycle that $\breve{\mathbf{vec}}$ lies on, shown in Algorithm 4 and Algorithm 5.

Example 6. Consider the circular singly-linked lists shown in Fig. 6(a) wherein the bit vectors are in the form of $\flat^r\flat^v\flat^u\flat^q\flat^p\,\flat^c$. For the circular list pointed to by p, if we cut the cycle at the *next* edge from list node n_5 to n_2, we get a non-

Algorithm 4 HEAD($\breve{\Gamma}$, $\breve{\mathbf{vec}}$)

Input: $\breve{\Gamma}$: a set of circular VRVs;
$\breve{\mathbf{vec}}$: a circular VRV in $\breve{\Gamma}$ s.t. $\breve{\mathbf{vec}}[c] = 1$;
Output: $\breve{head}$: the circular VRV of the head node of the cycle containing $\breve{\mathbf{vec}}$.

1: $\breve{head} \leftarrow \breve{\mathbf{vec}}$
2: **for** each $\breve{\mathbf{vec}}' \in \breve{\Gamma}$ satisfying $\breve{\mathbf{vec}}'[c] = 1$ **do**
3: **if** ($\breve{\mathbf{vec}}' \subset \breve{head} \wedge$ ONSAMECYCLE($\breve{\mathbf{vec}}', \breve{\mathbf{vec}}$)) **then**
4: $\breve{head} \leftarrow \breve{\mathbf{vec}}'$
5: **return** $\breve{head}$

Algorithm 5 TAIL($\breve{\Gamma}$, $\breve{\mathbf{vec}}$)

Input: $\breve{\Gamma}$: a set of circular VRVs;
$\breve{\mathbf{vec}}$: a circular VRV in $\breve{\Gamma}$ s.t. $\breve{\mathbf{vec}}[c] = 1$;
Output: $\breve{tail}$: the circular VRV of the tail node of the cycle containing $\breve{\mathbf{vec}}$.

1: $\breve{tail} \leftarrow \breve{\mathbf{vec}}$
2: **for** each $\breve{\mathbf{vec}}' \in \breve{\Gamma}$ satisfying $\breve{\mathbf{vec}}'[c] = 1$ **do**
3: **if** ($\breve{tail} \subset \breve{\mathbf{vec}}' \wedge$ ONSAMECYCLE($\breve{\mathbf{vec}}', \breve{\mathbf{vec}}$)) **then**
4: $\breve{tail} \leftarrow \breve{\mathbf{vec}}'$
5: **return** $\breve{tail}$

circular singly-linked list. Moreover, we use the flag bit to identify whether the node is on a cycle or not. n_1 is not on a cycle and thus its flag bit is 0, while $n_2, n_3, n_4, n_5, n_6, n_7, n_8$ are on cycles and thus their flag bits are 1. n_2, n_3, n_4, n_5 are on the same cycle of one list, while n_6, n_7, n_8 are on the same cycle of another list. Overall, for the two lists, we have a set of circular VRVs $\breve{\Gamma} = \{000100, 000111, 001111, 011111, 100001\}$. In Fig. 6(a), n_5, n_8 are the *tail* nodes while n_2, n_6 are the *head* nodes. n_4 is the first node on the cycle that pointer variable u can reach, and thus JOINCYCL($\breve{\Gamma}$, 000100) over shape in Fig. 6(a) returns 001111.

Intuitively, adding the flag bit c can be considered as introducing an auxiliary variable V_c which points to the destination node of the *next* edge that is cut. E.g., for the list pointed to by p, V_c points to n_2 in Fig. 6(a), n_4 in Fig. 6(c), n_5 in Fig. 6(e).

Theorem 4. *In singly-linked lists, a list node can be on at most one cycle.*

PROOF. Given in the appendix. □

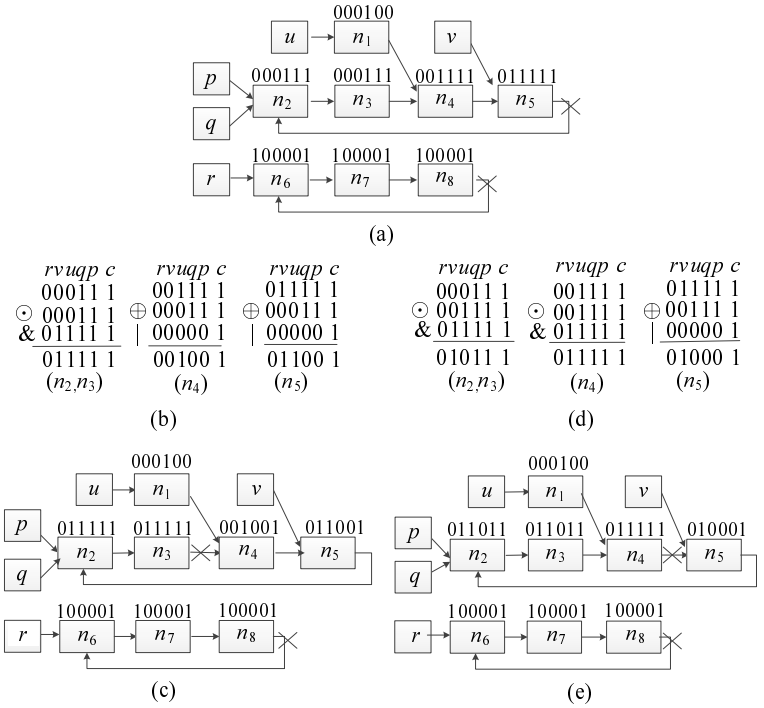

Figure 6: Example of circular lists

It is easy to see that there exists at most one cycle in a singly-linked list. Intuitively, a cycle as a whole could be considered as an abstract sink node of a singly-linked list. A pointer variable can reach at most one cycle. The circular VRVs $\mathbf{v\check{e}c}_{n_1}$, $\mathbf{v\check{e}c}_{n_2}$ of two list nodes from different cycles satisfy $(\mathbf{v\check{e}c}_{n_1} >> 1)\&(\mathbf{v\check{e}c}_{n_2} >> 1)=\texttt{0x0}$. Hence, although there may exist several circular lists at the same time in memory during program running (e.g., the two lists in Fig 6(a)), we need only one flag bit for all circular lists in memory.

Definition 6. A set of circular VRVs $\check{\Gamma}$ is *circularly consistent*, if for arbitrary two distinct circular VRVs $\mathbf{v\check{e}c}_{n_1}$ and $\mathbf{v\check{e}c}_{n_2}$ in $\check{\Gamma}$, $((\mathbf{v\check{e}c}_{n_1} >> 1)\&(\mathbf{v\check{e}c}_{n_2} >> 1) = 0) \vee \mathbf{v\check{e}c}_{n_1} \subset \mathbf{v\check{e}c}_{n_2} \vee \mathbf{v\check{e}c}_{n_2} \subset \mathbf{v\check{e}c}_{n_1}$ holds.

Theorem 5. *A set of circular VRVs of singly-linked lists is circularly consistent.*

Proof. Given in the appendix. □

Theorem 6. *A circularly consistent set of circular VRVs* $\check{\Gamma}$ *satisfies* $|\check{\Gamma}| \leq 2|V| + 2$.

Proof. Given in the appendix. □

Algorithm 6 ROTATE($\breve{\Gamma}$, $\breve{\mathbf{vec}}$)

Input: $\breve{\Gamma}$: a set of circular VRVs before rotation;
$\breve{\mathbf{vec}}$: a circular VRV of the new candidate tail node in $\breve{\Gamma}$ s.t.$\breve{\mathbf{vec}}[c] = 1$;
Output: M: a set of pairs that map a circular VRV to the new one after rotation.

1: $M \leftarrow \emptyset$
2: $\breve{tail} \leftarrow$ TAIL ($\breve{\Gamma}$, $\breve{\mathbf{vec}}$)
3: **for** each $\breve{\mathbf{vec}}' \in \breve{\Gamma}$ satisfying $\breve{\mathbf{vec}}'[c] = 1$ **do**
4: **if** ($\breve{\mathbf{vec}} \subset \breve{\mathbf{vec}}'$) **then**
5: $\breve{\mathbf{vec}}'' \leftarrow (\breve{\mathbf{vec}}' \oplus \breve{\mathbf{vec}}) \mid \mathtt{0x1}$
6: $M \leftarrow M \cup \{\langle \breve{\mathbf{vec}}', \breve{\mathbf{vec}}'' \rangle\}$
7: **else if** ($\breve{\mathbf{vec}}' \subseteq \breve{\mathbf{vec}}$) **then**
8: $\breve{\mathbf{vec}}'' \leftarrow (\breve{\mathbf{vec}}' \odot \breve{\mathbf{vec}}) \,\&\, \breve{tail}$
9: $M \leftarrow M \cup \{\langle \breve{\mathbf{vec}}', \breve{\mathbf{vec}}'' \rangle\}$
10: **return** M

Rotate operation. Given a circular list, we could choose different "*next*" edges to cut the cycle. And different cut points give different resulting sets of circular VRVs for the same circular list. To this end, we introduce a *rotate* operation to change the cut point for a circular list. We change the choice of the tail node of a cycle for implementing the rotate operation. More precisely, we change the cut point to the "*next*" edge of the last node of the list segment that the new candidate tail node lies on. The algorithm for the *rotate* operation, denoted as ROTATE($\breve{\Gamma}$, $\breve{\mathbf{vec}}$), is shown in Algorithm 6 where $\breve{\Gamma}$ is the current set of circular VRVs before rotation and $\breve{\mathbf{vec}}$ is the circular VRV of the new candidate tail node. E.g., in Fig. 6(a), if we call ROTATE($\breve{\Gamma}$, 000111), it means that we choose n_3 to become the new tail node.

Example 7. Rotate operation changes the choice of tail node of a cycle. In Fig. 6(a), n_5 is the tail node. If we would like to change the tail node to n_3, we call ROTATE($\breve{\Gamma}$, 000111). Fig. 6(b) shows the details of running ROTATE($\breve{\Gamma}$, 000111) while the resulting shape is shown in Fig. 6(c). Similarly, if we would like to change the tail node to n_4 in Fig 6(a), we call ROTATE($\breve{\Gamma}$, 001111). Fig. 6(d) shows the details of the algorithm of ROTATE($\breve{\Gamma}$, 001111) while the resulting shape is shown in Fig. 6(e). Since n_1 is not on the cycle, both ROTATE($\breve{\Gamma}$, 000111) and ROTATE($\breve{\Gamma}$, 001111) do not change its representation (i.e., 000100).

6.2. Numerical Abstraction for Circular Lists

Similarly as in Sect. 4, for each $\mathbf{v\check{e}c} \in \check{\Gamma}$, we introduce an auxiliary counter variable $t^{\mathbf{v\check{e}c}} \in \mathbb{N}$ to denote the value of the number of the list nodes whose circular VRV is $\mathbf{v\check{e}c}$. We maintain a bijection between $\mathbf{v\check{e}c}$ and $t^{\mathbf{v\check{e}c}}$. And $\{\langle \mathbf{v\check{e}c}, t^{\mathbf{v\check{e}c}} \rangle \mid t^{\mathbf{v\check{e}c}} > 0\}$ represents the shape of circular singly-linked lists, if it is circularly consistent. The shape in Fig. 6(a) can be represented by the set $\{\langle 000100, 1\rangle, \langle 000111, 2\rangle, \langle 001111, 1\rangle, \langle 011111, 1\rangle, \langle 100001, 3\rangle\}$, while the shape in Fig. 6(c) can be represented by the set $\{\langle 000100, 1\rangle, \langle 001001, 1\rangle, \langle 011001, 1\rangle, \langle 011111, 2\rangle, \langle 100001, 3\rangle\}$. For counter variables, we use the same numerical abstract domain (i.e., the $\mathcal{CD}$ domain) as in Sect. 4, and reuse those domain operations over counter variables.

6.3. Analysis of Programs Manipulating Circular Lists

In the following, we mainly focus on how to adapt abstract transfer functions over non-circular lists used in Sect. 5.1 to the case of circular lists. For test transfer function, we simply reuse the abstract semantics over non-circular lists in Sect. 5.1 for circular lists. However, considering assignment transfer functions, moreover, we need to consider the value of the flag bit c of circular VRVs, on top of the abstract semantics for non-circular lists in Sect. 5.1.

Abstract semantics for rotate operation. Before we define the abstract assignment transfer functions, we first give the abstract semantics for the rotate operation. Because the rotate operation is quite important for defining the abstract semantics for assignments such as $p := null$, free(p), $p := q \leftarrow next$, etc. Example 8 illustrates why we need the rotate operation for assignments. The Rotate() algorithm gives a mapping from original circular VRVs before rotation to the corresponding new circular VRVs after rotation. Since our analysis is performed on top of the domain over counter variables, we define the abstract semantics for rotate operation $[\![\text{Rotate}\ (\check{\Gamma}, \mathbf{v\check{e}c})]\!]^{\sharp}$ in terms of semantics over counter variables, as follows:

① Let M=Rotate($\check{\Gamma}$, $\mathbf{v\check{e}c}$) and $\check{\Gamma}_M = \bigcup_{\langle \mathbf{v\check{e}c}, \mathbf{v\check{e}c}' \rangle \in M} \{\mathbf{v\check{e}c}, \mathbf{v\check{e}c}'\}$. For each $\mathbf{v\check{e}c} \in \check{\Gamma}_M$, we introduce temporary variables $\dot{t}^{\mathbf{v\check{e}c}}$ initialized by zero: $\{\dot{t}^{\mathbf{v\check{e}c}} := 0;\}$

② For each $\langle \mathbf{v\check{e}c}, \mathbf{v\check{e}c}' \rangle \in M$, we build numerical statements:
if$(t^{\mathbf{v\check{e}c}} > 0)$ **then** $\{\dot{t}^{\mathbf{v\check{e}c}'} := t^{\mathbf{v\check{e}c}};\}$**fi**

③ For each $\mathbf{v\check{e}c} \in \check{\Gamma}_M$, we build numerical statements: $\{t^{\mathbf{v\check{e}c}} := \dot{t}^{\mathbf{v\check{e}c}};\}$

④ We project out and remove the dimensions of temporary variables $\dot{t}^{\mathbf{v\check{e}c}}$.

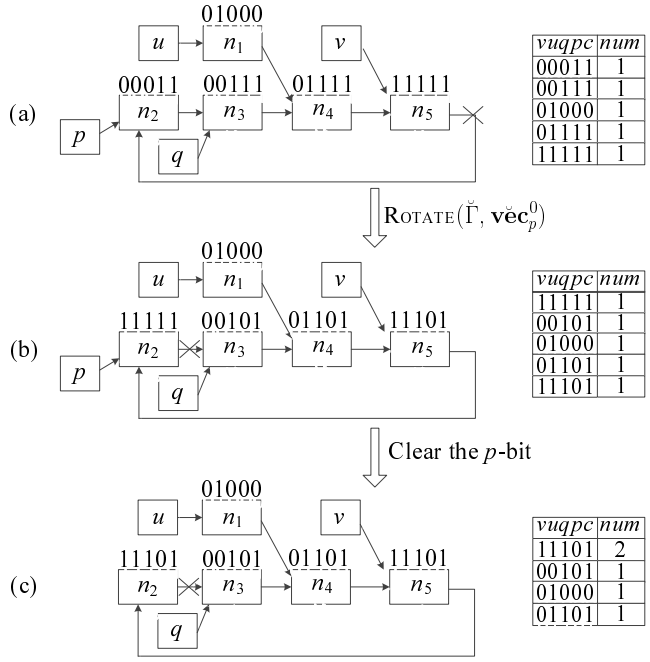

Figure 7: Example to show the need of rotate operation (for $[\![p := null]\!]^\sharp$)

Example 8. Consider the assignment transfer function $[\![p := null]\!]^\sharp$ over the circular list shown in Fig. 7. Note the bit vectors are in the form of $\flat^v\flat^u\flat^q\flat^p\,\flat^c$. If we do not perform the rotate operation and directly clear the p-bit in $\breve{\Gamma}$ over the shape in Fig. 7(a), then n_2 would be represented by 00001 which means that n_2 is on a cycle but no variable could reach it after cut, which is not sound according to the concrete semantics. And if we apply $[\![Rotate(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0)]\!]^\sharp$ where $\mathbf{v\breve{e}c}_p^0 = 00011$ over the shape in Fig. 7(a), we get the shape show in Fig. 7(b). Then, if we clear the p-bit in $\breve{\Gamma}$ over the shape in Fig. 7(b), we get the final result of $[\![p := null]\!]^\sharp$ shown in Fig. 7(c). For another example, let us consider $[\![u \rightarrow next := null]\!]^\sharp$ over the list shown in Fig. 6(c). If we do not perform rotate operation before clearing the u-bit, then n_4 will be leaked since it would be represented by 000001. However, if we first apply $[\![Rotate(\breve{\Gamma}, \textsc{JoinCycl}(\breve{\Gamma}, \mathbf{v\breve{e}c}_u^0))]\!]^\sharp$, we will get correct results.

Assignment transfer function over circular lists. Similarly as in Sect. 5.1, the main idea here is to transform an assignment over circular lists into a series of numerical statements over counter variables, taking into account to the changing of the shape. The abstract semantics of the assignment transfer function over

circular lists is shown in Fig. 8.

The key point here is that we first perform a rotate operation to change the cut point of a cycle such that the operated pointer variable points to or reaches first a list node that has the same circular VRV as the new tail node. E.g., when considering $[\![p := null]\!]^{\breve{\sharp}}$ over the circular list shown in Fig. 7(a), we first perform a rotate operation such that n_2 becomes a tail node, which results in the circular list shown in Fig. 7(b).

After having performed rotate operations to choose a new proper tail node when necessary, we could then follow the same idea as the corresponding semantics for non-circular lists given in Fig. 3. E.g., for the case of $[\![p := null]\!]^{\breve{\sharp}}$ over circular lists, we could simply perform $[\![p := null]\!]^{\sharp}$ (i.e., the ② step of $[\![p := null]\!]^{\breve{\sharp}}$), considering the cut circular lists as non-circular lists. However, for $[\![\text{free}(p)]\!]^{\breve{\sharp}}$ and $[\![p \rightarrow next := null]\!]^{\breve{\sharp}}$, we need to be more careful, since both operations will convert a circular list to a non-circular one when p points to a list node on a cycle, as considered in the ① step of $[\![\text{free}(p)]\!]^{\breve{\sharp}}$ and the ② step of $[\![p \rightarrow next := null]\!]^{\breve{\sharp}}$. On the other hand, $[\![p \rightarrow next := q]\!]^{\breve{\sharp}}$ forms a new cycle when q can reach the list node that p points to, as considered in the ③ step of $[\![p \rightarrow next := q]\!]^{\breve{\sharp}}$. E.g., when perform $[\![p \rightarrow next := u]\!]^{\breve{\sharp}}$ over the non-circular list shown in Fig. 9(b), we will get a circular list shown in Fig. 9(c). For the case of $[\![p := q \rightarrow next]\!]^{\breve{\sharp}}$, after rotating q to point to the tail node of a cycle when q lies on a cycle, if the length of the list segment that the tail node belongs to is 1, $q \rightarrow next$ will point to the head of the cycle, which is considered in the ② step of $[\![p := q \rightarrow next]\!]^{\breve{\sharp}}$.

Example 9. Fig. 9 shows examples of assignment transfer functions over circular lists, such as $[\![p \rightarrow next := null]\!]^{\breve{\sharp}}$ and $[\![p \rightarrow next := u]\!]^{\breve{\sharp}}$. Note the bit vectors are in the form of $\flat^v\flat^u\flat^q\flat^p\,\flat^c$. In Fig. 9(a), before applying $[\![p \rightarrow next := null]\!]^{\sharp}$, we have $P = \{t^{01000} = 1, t^{01001} = 1, t^{11001} = 1, t^{11101} = 1, t^{11111} = 1; t^{01000} \in [1,1], t^{01001} \in [1,1], t^{11001} \in [1,1], t^{11101} \in [1,1]\}, t^{11111} \in [1,1]\}$. According to the semantics of $[\![p \rightarrow next := null]\!]^{\breve{\sharp}}$, we build numerical statements:

- **if**$(t^{11111} \geq 1)$**then**$\{t^{00000} := t^{00000} + t^{11111} - 1; t^{11110} := 1; t^{11111} := 0\}$**fi** $\quad /\!*t^{11110} \leftarrow t^{11111}*\!/$
- **if**$(t^{01001} \geq 1)$**then**$\{t^{01000} := t^{01000} + t^{01001}; t^{01001} := 0\}$**fi** $\quad /\!*t^{01000} \leftarrow t^{01001}*\!/$
- **if**$(t^{11101} \geq 1)$**then**$\{t^{11100} := t^{11100} + t^{11101}; t^{11101} := 0\}$**fi** $\quad /\!*t^{11100} \leftarrow t^{11101}*\!/$
- **if**$(t^{11101} \geq 1)$**then**$\{t^{11000} := t^{11000} + t^{11001}; t^{11001} := 0\}$**fi** $\quad /\!*t^{11000} \leftarrow t^{11001}*\!/$

Finally, after having applied all the above assignment transfer functions over counters, we will get $P' = \{t^{01000} = 2, t^{11000} = 1, t^{11100} = 1, t^{11110} = 1; t^{01000} \in [2,2], t^{11000} \in [1,1], t^{11100} \in [1,1], t^{11110} \in [1,1]\}$. Furthermore we apply $[\![p \rightarrow$

$[\![p := null]\!]^{\breve{\sharp}}$	① If ($\mathbf{v\breve{e}c}_p^0[c] = 1$), we apply $[\![\textsc{Rotate}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0)]\!]^{\sharp}$. ② If ($\mathbf{v\breve{e}c}_p^0[c] = 0 \wedge \textsc{JoinCycl}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0) \neq \mathtt{0x1}$), we apply $[\![\textsc{Rotate}(\breve{\Gamma}, \textsc{JoinCycl}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0))]\!]^{\sharp}$. ③ We apply $[\![p := null]\!]^{\sharp}$.
$[\![p := \text{malloc}()]\!]^{\breve{\sharp}}$	① We apply $[\![p := \text{malloc}()]\!]^{\sharp}$ wherein $[\![p := null]\!]^{\breve{\sharp}}$ is used instead of $[\![p := null]\!]^{\sharp}$.
$[\![\text{free}(p)]\!]^{\breve{\sharp}}$	① If ($\mathbf{v\breve{e}c}_p^0[c] = 1$) • we apply $[\![\textsc{Rotate}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0)]\!]^{\sharp}$, and • for each $\mathbf{v\breve{e}c} \in \breve{\Gamma}$ s.t. $\textsc{OnSameCycle}(\mathbf{v\breve{e}c}, \mathbf{v\breve{e}c}_p^0) = \mathtt{TRUE}$, we build statements: $\mathbf{if}(t^{\mathbf{v\breve{e}c}} \geq 1)\mathbf{then}\{\ t^{\mathbf{v\breve{e}c}'} := t^{\mathbf{v\breve{e}c}'} + t^{\mathbf{v\breve{e}c}};\ t^{\mathbf{v\breve{e}c}} := 0;\ \}\mathbf{fi}$. wherein $\mathbf{v\breve{e}c}' = \mathbf{v\breve{e}c}/_{\mathcal{I}_{\{c\}}\leftarrow 0}$ ② If ($\mathbf{v\breve{e}c}_p^0[c] = 0 \wedge \textsc{JoinCycl}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0) \neq \mathtt{0x1}$) we apply $[\![\textsc{Rotate}(\breve{\Gamma}, \textsc{JoinCycl}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0))]\!]^{\sharp}$. ③ We apply $[\![\text{free}(p)]\!]^{\sharp}$.
$[\![p := q]\!]^{\breve{\sharp}}$	① First, we apply $[\![p = null]\!]^{\breve{\sharp}}$. ② We apply $[\![p := q]\!]^{\sharp}$.
$[\![p := q \rightarrow next]\!]^{\breve{\sharp}}$	① If ($\mathbf{v\breve{e}c}_q^0[c] = 0$) • we apply $[\![p := q \rightarrow next]\!]^{\sharp}$ wherein $[\![p := null]\!]^{\breve{\sharp}}$ is used instead of $[\![p := null]\!]^{\sharp}$. ② If ($\mathbf{v\breve{e}c}_q^0[c] = 1$) • we apply $[\![p := null]\!]^{\breve{\sharp}}$ and then $[\![\textsc{Rotate}(\breve{\Gamma}, \mathbf{v\breve{e}c}_q^0)]\!]^{\sharp}$, • if ($t^{\mathbf{v\breve{e}c}_q^0} \geq 2$) we apply the ② step of $[\![p := q \rightarrow next]\!]^{\sharp}$ • if ($t^{\mathbf{v\breve{e}c}_q^0} = 1$), for each $\mathbf{v\breve{e}c} \in \breve{\Gamma}$ s.t. $\textsc{OnSameCycle}(\mathbf{v\breve{e}c}, \mathbf{v\breve{e}c}_q^0)$=true, we build statements: $\mathbf{if}(t^{\mathbf{v\breve{e}c}} \geq 1)\ \mathbf{then}\ \{\ t^{\mathbf{v\breve{e}c}'} := t^{\mathbf{v\breve{e}c}'} + t^{\mathbf{v\breve{e}c}};\ t^{\mathbf{v\breve{e}c}} := 0;\ \}\mathbf{fi}$ where $\mathbf{v\breve{e}c}' \overset{\text{def}}{=} \mathbf{v\breve{e}c}/_{\{p\}\leftarrow 1}$
$[\![p \rightarrow next := null]\!]^{\breve{\sharp}}$	① If ($\mathbf{v\breve{e}c}_p^0[c] = 0$) • if ($\textsc{JoinCycl}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0) \neq \mathtt{0x1}$), we apply $[\![\textsc{Rotate}(\breve{\Gamma}, \textsc{JoinCycl}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0))]\!]^{\breve{\sharp}}$, and we apply $[\![p \rightarrow next := null]\!]^{\sharp}$. ② If ($\mathbf{v\breve{e}c}_p^0[c] = 1$) • we apply $[\![\textsc{Rotate}(\breve{\Gamma}, \mathbf{v\breve{e}c}_p^0)]\!]^{\sharp}$, and • for each $\mathbf{v\breve{e}c} \in \breve{\Gamma}$ s.t. $\textsc{OnSameCycle}(\mathbf{v\breve{e}c}, \mathbf{v\breve{e}c}_p^0)$=true, – if ($\mathbf{v\breve{e}c} = \mathbf{v\breve{e}c}_p^0$), we build statements: $\mathbf{if}(t^{\mathbf{v\breve{e}c}} \geq 1)\mathbf{then}\ \{t^{0\ldots0} := t^{0\ldots0} + t^{\mathbf{v\breve{e}c}} - 1; t^{\mathbf{v\breve{e}c}'} := 1; t^{\mathbf{v\breve{e}c}} := 0;\}\ \mathbf{else}\ \{P' := \top;\}\ \mathbf{fi}$ where $\mathbf{v\breve{e}c}' \overset{\text{def}}{=} \mathbf{v\breve{e}c}/_{\mathcal{I}_{\{c\}}\leftarrow 0}$ – else we build numerical statements: $\mathbf{if}(t^{\mathbf{v\breve{e}c}} \geq 1)\mathbf{then}\{\ t^{\mathbf{v\breve{e}c}'} := t^{\mathbf{v\breve{e}c}'} + t^{\mathbf{v\breve{e}c}};\ t^{\mathbf{v\breve{e}c}} := 0;\ \}\mathbf{fi}$ where $\mathbf{v\breve{e}c}' \overset{\text{def}}{=} \mathbf{v\breve{e}c}/_{\mathcal{I}_{\{c\}}\leftarrow 0}$
$[\![p \rightarrow next := q]\!]^{\breve{\sharp}}$	① First, we apply $[\![p \rightarrow next := null]\!]^{\breve{\sharp}}$. ② if ($\mathbf{v\breve{e}c}_p^0[q] = 0$), we apply $[\![p \rightarrow next := q]\!]^{\sharp}$ ③ if($\mathbf{v\breve{e}c}_p^0[q] = 1$) • for each $\mathbf{v\breve{e}c} \in \breve{\Gamma}$ such that $\mathbf{v\breve{e}c}[q] = 1$, we build statements: $\mathbf{if}(t^{\mathbf{v\breve{e}c}} \geq 1)\mathbf{then}\{\ t^{\mathbf{v\breve{e}c}'} := t^{\mathbf{v\breve{e}c}'} + t^{\mathbf{v\breve{e}c}};\ t^{\mathbf{v\breve{e}c}} := 0;\ \}\mathbf{fi}$ where $\mathbf{v\breve{e}c}' \overset{\text{def}}{=} \mathbf{v\breve{e}c}/_{\mathcal{I}_{\{c\}}\leftarrow 1}$

Figure 8: Abstract assignment transfer functions for circular lists

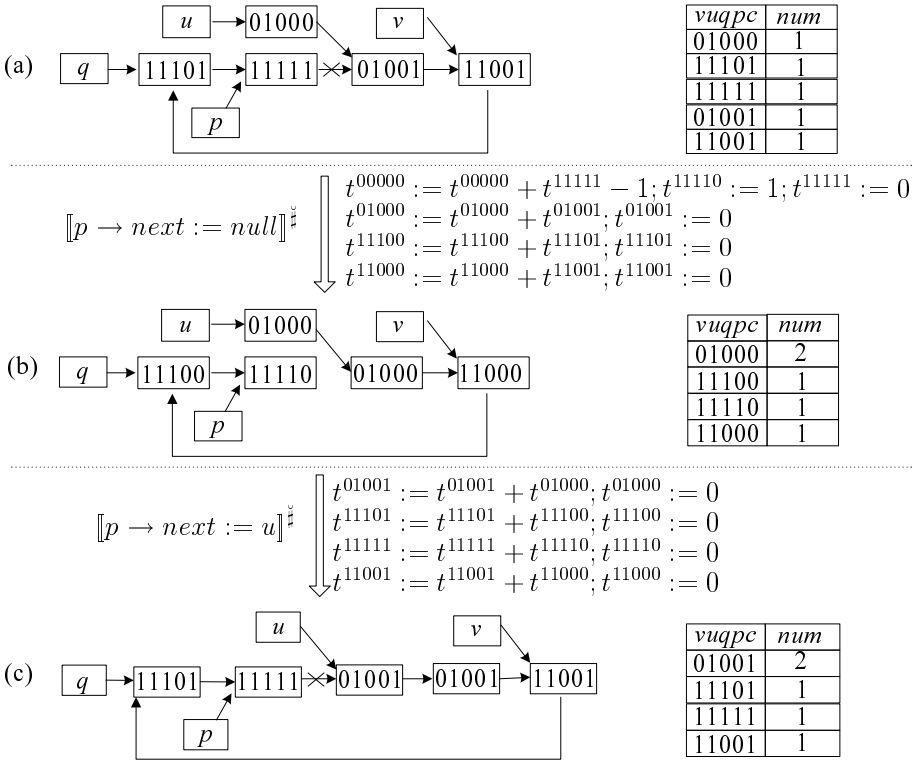

Figure 9: Examples of assignment transfer functions over circular lists

$next := u]\!]^{\breve{\sharp}}$. According to the semantics of $[\![p \to next := u]\!]^{\breve{\sharp}}$, we build numerical statements:

- **if**$(t^{01000} \geq 1)$**then**$\{t^{01001} := t^{01001} + t^{01000}; t^{01000} := 0\}$**fi** /*$t^{01001} \leftarrow t^{01000}$*/
- **if**$(t^{11100} \geq 1)$**then**$\{t^{11101} := t^{11101} + t^{11100}; t^{11100} := 0\}$**fi** /*$t^{11101} \leftarrow t^{11100}$*/
- **if**$(t^{11110} \geq 1)$**then**$\{t^{11111} := t^{11111} + t^{11110}; t^{11110} := 0\}$**fi** /*$t^{11111} \leftarrow t^{11110}$*/
- **if**$(t^{11000} \geq 1)$**then**$\{t^{11001} := t^{11001} + t^{11000}; t^{11000} := 0\}$**fi** /*$t^{11001} \leftarrow t^{11000}$*/

After having applied all the above assignment transfer functions over counters, we will get $P'' = \{t^{01001} = 2, t^{11001} = 1, t^{11101} = 1, t^{11111} = 1; t^{01001} \in [2,2], t^{11001} \in [1,1], t^{11101} \in [1,1], t^{11111} \in [1,1]\}$.

Maintaining points-to sets over circular lists. Similarly as in Sect. 5.2, for the case of circular lists, we also use another abstract domain to maintain a possible points-to set $\breve{\Gamma}^0_p$ for each pointer variable p, for the sake of precision. For most operations, we just follow similar ideas as those for non-circular lists shown in

Fig. 5. However, for several kinds of assignments that may change circular lists to non-circular ones or the other direction, we need to consider the change of the circle flag bit for elements in $\check{\Gamma}^0_p$. $[\![\text{free}(p)]\!]^{\check{\sharp}}$ and $[\![p \rightarrow next := null]\!]^{\check{\sharp}}$ may convert a circular list to a non-circular one. Hence, we may need to clear the cycle flag bit for those circular VRVs in the points-to sets of pointer variables that lie on the same cycle as p does. On the contrary, $[\![p \rightarrow next := q]\!]^{\check{\sharp}}$ may form a new cycle when q can reach the list node that p points to. Hence, we may need to set to 1 the cycle flag bit for those circular VRVs in the points-to sets of pointer variables u satisfying $\check{\mathbf{vec}}^0_u[q]$=1. Different with $[\![p \rightarrow next := q]\!]^{\sharp_0}$ for non-circular lists, we do not set the $I_{\check{\mathbf{vec}}^0_p}$ bits to 1 for those $\check{\mathbf{vec}}^0_u$ satisfying $\check{\mathbf{vec}}^0_u[q] = 1$, since in this case p points to the tail node of the newly formed cycle. E.g. when we perform $[\![p \rightarrow next := u]\!]^{\check{\sharp}}$ over the list in Fig 9(b), we have $\check{\mathbf{vec}}^0_p = 11110$, but we will not set $\mathcal{I}_{\check{\mathbf{vec}}^0_p}$ bits to 1 for the circular VRVs of list nodes pointed to by u, v, q. It is also worth noting that the rotate operation that is frequently used for circular lists will change the points-to set of those pointer variables that lie on the cycle which is rotated. Hence, we need to update the points-to sets for those variables after the rotate operation, according to the mapping given by the ROTATE() algorithm.

7. Experiments

We have developed a prototype tool for analyzing list manipulating programs based on the APRON [13] numerical abstract domain library and the INTERPROC [14] static analyzer. We implemented our $\mathcal{CD}$ domain inside APRON. Since INTERPROC uses the Spl input language which supports only numeric (integer or real) variables, inspired by CINV [4], we encode our programs on lists via Spl. Pointer variables of list type are coded by real variables while data variables (such as length-related parameters, loop counters) are encoded by integer variables. The constant NULL is encoded by value 0.0. And the operations on pointers are encoded using operations on real variables. E.g., $p := q \rightarrow next$ is encoded by $p = cast_{f,n}(q)$, and $p \rightarrow next := q$ is encoded by $p = cast_{d,n}(q)$. In the implementation of the $\mathcal{CD}$ domain, over the abstract environment for affine equalities, we maintain affine equality constraints among integer program variables (such as loop counter variables and length-related parameters) and auxiliary counter variables $t^{\mathbf{vec}}$. Hence, we could infer invariants like $t^{\mathbf{vec}} - n + i = 0$, where n is a length-related parameter and i is a loop counter variable.

To exemplify the ability of invariant synthesis of our $\mathcal{CD}$ domain, let us consider an example manipulating non-circular lists *copy_and_delete1* (which for sake of space allows statements like $p := p \rightarrow next$ and is a simplified non-parametric

void copy_and_delete1(List* $xList$) {
/* assume \length($xList$)==9; */
1: List* $yList$, $pList$, $qList$;
/ * $t^{0100} = 9; t^{0100} \in [9,9]; \heartsuit$ * /
2: $yList := xList$; $qList := pList := null$;
/ * $t^{0100} + t^{1100} = 9, t^{0011} + t^{1100} = 9; t^{0100} \in [0,9], t^{1100} \in [0,9], t^{0011} \in [0,9]; \heartsuit$ * /
3: **while** ($yList$!= $null$) **do** {
/ * $t^{0100} + t^{1100} = 9, t^{0011} + t^{1100} = 9; t^{0100} \in [0,8], t^{1100} \in [1,9], t^{0011} \in [0,8]; \heartsuit$ * /
4: $yList := yList \rightarrow next$; $qList$:= malloc();
5: $qList \rightarrow next := pList$; $pList := qList$;} **od**
/ * $t^{0011} = 9, t^{0100} = 9; t^{0011} \in [9,9], t^{0100} \in [9,9]; \heartsuit$ * /
6 : $yList := xList$;
/ * $t^{0011} - t^{1100} = 0; t^{0011} \in [0,9], t^{1100} \in [0,9]; \heartsuit$ * /
7 : **while** ($yList$!= $null$) **do** {
/ * $t^{0011} - t^{1100} = 0; t^{0011} \in [1,9], t^{1100} \in [1,9]; \heartsuit$ * /
8: $yList := yList \rightarrow next$; $qList := qList \rightarrow next$;
9: free($xList$); free($pList$);
10: $xList := yList$; $pList := qList$;
11: } **od**
/ * $\forall \mathbf{vec}.t^{\mathbf{vec}} = 0$ * /
}

Figure 10: Example program *copy_and_delete*1 and the generated invariants. The notation $\heartsuit$ means $t^{\mathbf{vec}} = 0$ for any $t^{\mathbf{vec}}$ that does not appear in the annotated invariant.

version of copy_and_delete2 in Fig. 11) together with the generated invariants by the $\mathcal{CD}$ domain, as shown in Fig. 10. The program first reversely copies one list to another and then deletes both lists simultaneously. Suppose the initial length of the input list $xList$ is 9 [1] and the variable ordering is $pList \prec qList \prec xList \prec yList$ (i.e., the bit vectors are in the form of $\flat^{yList}\flat^{xList}\flat^{qList}\flat^{pList}$). From the invariants after line 7, we can see: (1) Pointer variables $pList$, $qList$ are aliases while $xList$, $yList$ are aliases (since for all $t^{\mathbf{vec}} \neq 0, \mathbf{vec}[x] = 1$ holds if and only if $\mathbf{vec}[y] = 1$ holds); (2) The two lists respectively pointed to by $pList$ and $xList$ are of the same length (according to the invariant $t^{0011} - t^{1100} = 0$); (3) The bounds of counter variables are strictly positive, which indicates that the operations on lines 8-10 are free of null pointer dereference. Finally, the special auxiliary variable $t^{0\cdots0}$ equals to 0 at all the program points, which proves the absence of memory leak in the

[1]Note that our implementation allows that the initial length of a list can be a non-negative integer parameter such as $n \in \mathbb{N}$. However, for the sake of illustration, we use a constant number 9 as the initial length of the input list in this example.

program.

Our experiments were conducted on a selection of benchmark examples listed in Fig. 11, some of which are taken from [4]. These benchmark examples contain commonly used operations over lists, such as create, traverse, reverse, merge, copy, delete and dispatch. Programs with postfix name "circular" are the circular version of corresponding programs with non-circular lists. These benchmark programs involve relational properties among lengths of list segments. Our $\mathcal{CD}$ domain that is based on intervals and affine equalities, is able to find interesting affine equality relations and bounds of lengths of list segments. In Fig. 11, the column "Key Property Discovered" shows some key properties in the program discovered by our approach, and also gives some important affine equality relations among list segments found by our $\mathcal{CD}$ domain inside loops or after loops. In the last two programs, we detect memory related errors. For *del_without_head*, we detect a memory leak at the end of program, since in this program all list nodes are freed except the head node. For one_branch_free, we detect a null pointer dereference, since $p := q \rightarrow next$ appears after a branch statement wherein free(q) is called in one of the branches.

We have conducted experimental comparisons of our analysis based on the $\mathcal{CD}$ domain with CINV [4] and THOR [15] on those benchmark programs, as shown in Fig. 12 and Fig. 13 respectively. The column "#PVars" indicates the total number of pointer variables in the program. The columns "Time (s)" and "Memory (MB)" present the analysis times in seconds and the memory usage in MB respectively when the analyzer is run on a 2.5GHz PC with 2GB of RAM running Fedora 12.

Comparison with CINV. Fig. 12 shows the experimental comparison results of our analysis with CINV [4] which is a tool for analyzing programs manipulating singly linked lists developed by Bouajjani et al. CINV combines shape abstraction (via shape graph) and numerical abstractions, and is able to infer not only length properties of list segments but also properties over the numerical list contents (such as the sums or the multisets of list elements, sortedness). We compared our analysis with CINV on inferring length properties of list segments, since the benchmark programs used in this paper do not involve operations over list contents. Since CINV cannot handle circular lists, the time and memory cost of CINV over those programs manipulating circular lists are marked with "-".

From Fig. 12, we can see that our analysis based on the $\mathcal{CD}$ domain outperforms CINV both in execution time and memory usage. Concerning the memory usage, CINV utilizes shape graph to represent the shape of lists, while our anal-

Program	Key Property Discovered
create	Created list has length n (e.g., $t^{10} = n$)
traverse	Sum of traversed and remained list sizes is n during the loop (e.g., $t^{010} + t^{011} = n$)
reverse	Sum of reversed and remained list sizes is n during the loop (e.g., $t^{001} + t^{010} = n$)
length_equal	The two input lists have the same length n (e.g., $t^{00011} - t^{01100} = 0$)
merge	Merged list size is the sum of input list sizes (e.g., $t^{10001} = n_1 + n_2$)
copy_and_delete2	Copied list has the same length as input list before delete (e.g., $t^{00011} - t^{11000} = 0$)
dispatch	Sum of (two) dispatched and remained list sizes is n (e.g., $t^{000010} + t^{000100} + t^{001000} = n$)
counter	Sum of countered and remained list sizes is n during the loop (e.g., $t^{001} - counter = 0, t^{011} + counter = n$)
double_len	After deleting half the longer list, the lengths of the two lists are equal (e.g., $t^{00001} - t^{00010} = 0$)
create_circular	Created circular list has length n (e.g., $t^{0011} = n$)
reverse_circular	Sum of reversed and remained list sizes is n during the loop (e.g., $t^{00100} + t^{01000} + t^{11100} = n$)
copy_and_delete2 _circular	Copied list has the same length as input list before delete (e.g., $t^{0000101} - t^{0100001} = 0$)
counter_circular	Sum of countered and remained list sizes is n during the loop (e.g., $t^{0011} - counter = 0, t^{0111} + counter = n$)
del_without_head	A memory leak is detected (e.g., $t^{000} = 1$)
one_branch_free	A null pointer dereference is detected (e.g., $p := q \rightarrow next$ where $t^{\mathbf{vec}^0_q} \in [0, 1]$)

Figure 11: Key properties discovered for benchmark examples by the $\mathcal{CD}$ domain

Program		Time (s)		Memory (MB)	
Name	#PVars	CINV	$\mathcal{CD}$	CINV	$\mathcal{CD}$
create	2	0.872	0.022	20.2	6.5
traverse	3	0.920	0.027	21.0	6.7
reverse	3	0.933	0.044	23.8	6.7
length_equal	5	1.453	0.162	30.5	17.0
merge	5	0.953	0.127	25.9	13.5
copy_and_delete2	5	1.042	0.197	23.9	17.2
dispatch	6	1.24	0.52	57.8	39.6
counter	3	1.242	0.079	21.1	8.2
double_len	5	3.822	0.158	35.5	15.4
create_circular	3	-	0.041	-	8.4
reverse_circular	4	-	0.077	-	9.8
copy_and_delete2 _circular	6	-	0.543	-	61.9
counter_circular	3	-	0.040	-	8.3
del_without_head*	3	0.020	0.009	20.7	6.6
one_branch_free	2	0.005	0.005	19.3	6.5

Figure 12: Experimental comparison of our approach with CINV

ysis utilized bit vectors and thus consumes much less memory to describe the shape information. Concerning the execution time, CINV needs to perform shape analysis over shape graph to infer shape properties, while our analysis utilizes bit-wise operations on bit-vectors which is quite efficient. For the precision of the analysis, we have compared the discovered numerical invariants over lengths of list segments given by CINV and the $\mathcal{CD}$ domain. The generated invariants are equivalent for all benchmark programs manipulating non-circular lists except the *del_without_head* program marked with * which incurs memory leaks. CINV does not consider memory leaks, while our analysis can detect memory leaks or prove their absence since our $\mathcal{CD}$ domain utilizes a special counter variable $t^{0...0}$ to capture the relations of the amount of memory leaks with other counter variables and integer program variables.

Comparison with THOR. Fig. 13 shows the experimental comparison results of our analysis based on the $\mathcal{CD}$ domain with THOR [15] which is a tool able to translate a heap-manipulating program into a purely numerical program based on separation logic reasoning. THOR itself only performs sound shape analysis based on separation logic and has limited capability of numerical reasoning. It can

either prove the memory safety of the program or issue memory error alarms. If an alarm is issued by THOR, we need to feed the generated numerical program by THOR to a separate safety tool with more powerful numerical reasoning capabilities, to check whether the alarm is a false or true alarm. During our experiments, we chose BLAST to analyze the generated numerical programs, as in [15].

Program		Time (s)				Memory (MB)			
		THOR			$\mathcal{CD}$	THOR			$\mathcal{CD}$
Name	#PVars	THOR .native	BLAST	total		THOR .native	BLAST	total	
create	2	0.502	0.415	0.917	0.022	43.1	8.2	51.3	6.5
traverse	3	0.249	0.103	0.352	0.027	34.5	15.6	50.1	6.7
reverse	3	0.219	0.208	0.427	0.044	34.5	9.1	43.6	6.7
length_equal	5	1.782	60.624	62.406	0.162	38.2	53.6	91.8	17.0
merge	5	1.211	0.364	1.575	0.127	44.8	9.5	54.3	13.5
copy_and _delete2	5	0.196	93.864	94.06	0.197	34.2	132.6	166.8	17.2
dispatch	6	5.339	0.434	5.773	0.520	56.1	9.2	65.3	39.6
counter	3	0.902	0.245	1.147	0.079	40.2	9.1	49.3	8.2
create_circular	3	0.592	0.433	1.025	0.041	43.5	9.3	52.8	8.4
reverse_circular	4	1.589	6.679	8.268	0.077	45.8	12.8	58.6	9.8
copy_and_delete2 _circular	6	2.167	2.940	5.107	0.543	56.8	12.6	69.4	61.9
counter_circular	3	1.866	8.092	9.958	0.040	46.1	15.6	61.7	8.3
one_branch_free	2	0.273	0.138	0.411	0.005	38.2	1.1	39.3	6.5
del_without_head*	3	0.411	31.815	32.226	0.009	42.8	38.4	81.2	6.6
double_len*	5	1.878	1.015	2.893	0.158	47.8	26.4	74.2	15.4

Figure 13: Experimental comparison of our approach with THOR

In Fig. 13, the column "THOR.native" corresponds to the THOR tool itself. The benchmark programs are classified into three groups, according to the analysis results. For the programs in the first group (from *create* to *counter_circular*) that are in fact safe in memory, THOR itself issues unsafe memory alarms but which are found to be false alarms by BLAST afterwards. Our analysis based on the $\mathcal{CD}$ domain can also prove the memory safety of all those programs. For the program in the second group, i.e., *one_branch_free* which contains a null pointer dereference, THOR itself issues a null pointer dereference alarm which cannot be eliminated by BLAST afterwards, while our analysis also detects a null pointer dereference for this program. For the programs marked with * in the third group, THOR and our analysis provide different results. For *del_without_head*, THOR

does not detect the memory leak in the program while our analysis does. For the program *double_len* which is in fact safe in memory, THOR itself issues memory error alarms which are actually false alarms but cannot be eliminated by BLAST afterwards. *double_len* is a program with two input lists, i.e., *xList* with length n and *yList* with length $2n$. First, the program traverses the list nodes of *xList* in an outer loop and after each traverse step over a list node of *xList* an inner loop removes a tail node of *yList*. After this process, the lengths of the two lists should be equal. Then the program deletes both lists simultaneously in another loop. THOR together with BLAST cannot prove the memory safety of this problem while our analysis based on the $\mathcal{CD}$ domain can.

From Fig. 13, we can see that our analysis based on the $\mathcal{CD}$ domain outperforms the THOR approach (i.e., THOR together with BLAST) both in execution time and memory usage.

Experiments on different number of pointer variables. Given a program with $|V|$ number of pointer variables, the number of auxiliary counter variables (i.e., t^{vec}) is $2^{|V|}$. However, Theorem 3 and Theorem 6 show that to describe the list shape at some program point, at most $2|V| + 2$ counter variables are non-zero. At different program points in the same program, the shape of lists is changing and the set of non-zero counter variables are changing. Considering this, in our implementation, inside each abstract $\mathcal{CD}$ domain element we maintain a dynamic list of non-zero counter variables and maintain numerical constraints among only these non-zero counter variables.

Program		Time (s)			Memory (MB)		
Name	#PVars	CINV	THOR	$\mathcal{CD}$	CINV	THOR	$\mathcal{CD}$
traverse-1	3	0.920	0.249	0.027	21.0	34.5	6.7
traverse-2	6	1.046	1.615	0.059	23.9	45.836	6.9
traverse-3	9	1.119	>10min	0.081	27.8	-	7.1
traverse-4	12	1.263	>10min	0.120	34.2	-	7.4
traverse-5	15	1.397	>10min	0.144	43.0	-	7.6
traverse-6	18	1.501	>10min	0.183	53.4	-	8.0
traverse-7	21	1.719	>10min	0.225	66.4	-	8.5
traverse-8	24	1.828	>10min	0.264	81.8	-	9.2
traverse-9	27	2.092	>10min	0.330	100.3	-	9.8
traverse-10	30	2.368	>10min	0.383	122.4	-	10.5

Figure 14: Experiments on different number of pointer variables.

In order to evaluate the performance of our analysis on different number of

pointer variables, we have conducted an experiment over a series of programs that have the same functionality but with different number of pointer variables, as shown in Fig. 14. In Fig. 14, the program *traverse-n* is composed of *n* copies of the *traverse* program which contains a loop traversing a list and different copies use different set of pointer variables. The experimental results in Fig. 14 show that increasing of numbers of pointer variables does not necessarily cause significant performance degradation for our analysis. However, THOR (itself) cannot complete the transformation of list manipulating programs into numerical programs in 10 minutes when the number of pointer variables is greater than 9.

8. Related Work

Shape abstractions. Programs manipulating lists have gained much attention within the past decade [16, 17, 18]. And various abstractions have been used for analyzing shape properties of lists and other dynamically linked data structures that are more general, such as canonical abstraction [2, 17], boolean heaps [19], separation logic [20], etc. The TVLA system [21] is one of the first shape analysis engines for implementing shape abstractions via 3-valued logic formulae [2]. TVLA provides a set of predicates [17] for users to describe the shape of linked lists, including the *Reach* predicate that we use in our work. However, TVLA itself does not perform numerical reasoning and thus is not able to infer numerical relations among list segments. Recently, Ferrara et al. [22, 23] proposed a generic approach to combine TVLA with value analyses in the framework of abstract interpretation, but in which the value analyses are used to track the value contents contained in abstract heap nodes rather than length properties.

One work that is close to our shape abstraction is boolean heaps [19] by Podelski et al. It adopts also the concept "bit-vector" and utilizes sets of bit-vectors to encode boolean heaps. Their approach is general for modeling all kinds of relations in heaps by using proper heap predicates, while in this paper we focus on lists and utilize one basic predicate (*Reach*). In addition, our approach maintains automatically numerical relations among list segments. Recently, Gulwani et al. [24] propose an abstract domain that allows representation of must and may equalities among pointer expressions. Our work uses also equalities but to track the numerical properties over the number of list nodes.

In recent years, great progress has been made in automatic verification of heap-manipulating programs using separation logic [25, 26]. A number of automatic tools based on separation logic have been developed, including Smallfoot [27], SpaceInvader [28], SLAyer [29], Predator [30], Forrester [31]. Using separation

logic predicates can describe more general data structures (such as trees) besides linked lists. However, these approaches mainly focus on analyzing only shape properties, while our work considers not only shape properties but also length properties over lists in the framework of abstract interpretation.

Combining shape and numerical abstractions. Recently, much attention has been focused on combining shape and numerical abstractions [5, 6, 7, 18].

Bouajjani et al. [4, 18, 32] utilize counter automata as an abstract model for lists, and propose a framework for combining a heap abstraction with various abstractions over the sequences of data in a list. Their method maintains the exact data stored in a list segment as well as their sequences and thus can discover relational properties over list contents. Compared with the counter automata model, our heap abstraction based on bit-vectors is quite lightweight. Experimental comparison results in Fig. 12 show that our prototype is more efficient than their implementation CINV both in time and memory. Moreover, CINV cannot handle circular lists while our prototype can. In addition, our approach can also detect the amount of memory leak, which is not considered in CINV.

Chang and Rival [5] propose relational inductive shape analysis based on an abstract shape graph representation, in which heaps are described by user-supplied inductive predicates defined via separation logic. Their approach provides a modular framework to combine shape and numerical data abstract domains, and can handle more generic data structures such as red-back trees. Compared with their work, our work targets at only linked lists but provides a more lightweight encoding of shape abstraction for lists than their graph representation, thanks to the compact bit-vectors. Moreover, they use inductive predicates to specify the shape of the heap and do not allow changing the type of the shape, but our approach allows a program to change non-circular lists into circular lists and vice versa. In addition, as far as we know, in their implementation Xisa, they have not yet employed numerical abstract domains to perform numerical reasoning.

More recently, Qin et al. [7, 33, 34] propose a separation logic based approach to synthesize loop invariants involving both shape and numerical properties by utilizing a combined separation and numerical domain to enhance the HIP/SLEEK [35, 36] system which previously relied on users to provide annotations describing loop invariants. Due to the usage of separation logic, their approach enjoys the benefit from the frame rule and thus supports local reasoning, but relies on a separation logic prover for entailment checking over the heap domain. Their approach can handle more general data structures such as trees, but requires user-supplied inductive shape predicates. To ensure the termination of the fixpoint

iteration, their analysis algorithm needs a user-provided parameter to specify a finite upper bound of the number of shared logical variables [7]. Compared with their approach, our approach focuses on linked lists and is lightweight (thanks to the bit vector based shape abstraction), with no need of complex logic reasoning engine and user-supplied predicates. Moreover, we have adapted traditional numerical abstractions to fit for inferring length properties of list manipulating programs, taking into account special characteristics of the analyzed program. E.g., we have modified the traditional interval widening by taking into account the fact that counter variables are non-negative integer variables.

Gulwani et al. [6] propose a general combination framework for tracking relationships between sizes of memory partitions. It combines a set domain (for tracking memory partitions) with a set cardinality domain (for tracking relations between cardinalities of the partitions) via reduced products. Our work encodes the shape abstraction by bit-vectors that itself can be considered as numerical values, which makes it easy to build a combined domain based on numerical abstractions taking into account the semantics of shape abstractions, without resort to reduced product.

Reducing heap-manipulating to numerical programs. Magill et al. [15, 37] propose a method to automatically transform heap manipulating programs into numeric ones, based on their shape analysis over user-supplied separation logic predicates. The idea of separating shape abstraction and numerical abstraction in [15] has clear engineering advantages to make use of existing numeric reasoning tools. However, the transformation is unidirectional and thus may lose precision especially when the shape aspect and the numerical aspect interact in complicated ways. Our work takes into consideration both shape and numerical information at the same time during the analysis, and thus can be more precise. For example, our approach can detect the amount of memory leaks, while those information is lost during the transformation of their approach. In the experimental section, we have an example that our approach can prove the safety of the program while their implementation THOR [15] together with BLAST [38] could not. Moreover, experimental results in Sect. 7 show that our approach is more efficient than THOR (together with BLAST) both in time and memory.

Analysis and verification of circular lists. Little existing work considers analyzing and verifying properties over circular lists [17, 39, 40, 41]. Manevich et al. [17] use predicate abstraction and canonical abstraction for (potentially circular) singly-linked lists in the TVLA abstract interpreter [21], where a so-called instrumentation predicate $c_n(v)$ is introduced to describe that v resides on a cycle

of n fields, which is similar to idea of our circular flag bit. However, our approach further cuts a circular list into a non-circular one and tracks numerical relations among the sizes of list segments on the same cycle. Bouajjani et al. [40] propose an automata based approach to verify programs with singly-linked lists by regular model checking. The shape of (circular) lists is described by regular expressions and 0-k counter abstraction is used to describe single size of each list segment. However, our approach uses numerical abstract domains to capture the relations among the sizes of list segments. Bozga et al. [41] consider the complexity of checking safety and termination properties for flat programs with singly-linked lists. They show that verifying safety and termination for programs working on heaps with more than one cycle are undecidable, whereas decidability can be established when the input heap may have at most one loop.

9. Conclusion

We have presented an approach in the framework of abstract interpretation for analyzing list-manipulating programs. The main idea is to combine heap and numerical abstractions. The structural information of the shape of a list is encoded in a lightweight way via bit vectors, one for each list segment, while numerical relations among the number of list nodes in these segments are tracked by numerical abstract domains. We have instantiated our approach by establishing a combination domain of intervals and affine equalities to infer relations over the length of list segments. We have also show how our approach works for circular lists. A key benefit of our approach is the ability to leverage the power of the state-of-the-art numerical abstract domains to analyze intricate properties of list-manipulating programs.

Future work will consider extending our approach to infer properties over the content of lists, e.g., sortedness, no duplicated elements. Inferring non-trivial relational properties over list contents requires reasoning over inter-segment relations among different segments and intra-segment relations among elements in the same segment. Following the same idea of introducing counter variables, for each $\mathbf{vec} \in \Gamma$, we could introduce an auxiliary content variable $d^{\mathbf{vec}}$ to abstract the contents of the list nodes whose VRV is **vec**. If the list contents are of numerical data type, we could apply numerical abstractions to $d^{\mathbf{vec}}$, similarly to what we do over $t^{\mathbf{vec}}$. However, to be sound, we can only apply weak update semantics to handle assignments to $p \rightarrow data$. In order to obtain more precise information over $p \rightarrow data$, we need to extend our bit-vector shape abstraction to distinguish the first element from the rest elements in a list segment. Another direction of

the work is to deal with doubly linked lists that are well-founded [42]. We could maintain reachability properties for the *next* field and the *prev* field separately. In other words, we could maintain different VRV sets for *next* and *prev*. And we perform communication and propagation between the two fields when needed. Finally, it would be also interesting to consider more general data structures such as trees. However, our shape encoding via bit-vectors is initially designed specifically for linked lists. For other data structures such as trees, we need to design other kinds of shape abstraction, but the general idea of combining shape abstraction and numerical abstraction is still applicable.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments. This work is supported by the 973 Program under Grant No. 2014CB340703, the NSFC under Grant Nos. 61120106006, 61202120, 91318301, and the SRFDP under Grant No. 20124307120034.

References

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, H. Yang, Shape analysis for composite data structures, in: CAV, Vol. 4590 of LNCS, Springer, 2007.

[2] S. Sagiv, T. W. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, ACM Trans. Program. Lang. Syst. 24 (3) (2002) 217–298.

[3] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: POPL, ACM Press, 1977, pp. 238–252.

[4] A. Bouajjani, C. Dragoi, C. Enea, A. Rezine, M. Sighireanu, Invariant synthesis for programs manipulating lists with unbounded data, in: CAV, Vol. 6174 of LNCS, Springer, 2010, pp. 72–88.

[5] B. E. Chang, X. Rival, Relational inductive shape analysis, in: POPL, ACM, 2008, pp. 247–260.

[6] S. Gulwani, T. Lev-Ami, M. Sagiv, A combination framework for tracking partition sizes, in: POPL, ACM, 2009, pp. 239–251.

[7] S. Qin, G. He, C. Luo, W. Chin, Loop invariant synthesis in a combined domain, in: ICFEM, Vol. 6447 of LNCS, Springer, 2010, pp. 468–484.

[8] P. Cousot, R. Cousot, Static determination of dynamic properties of programs, in: Proc. of the 2nd International Symposium on Programming, Dunod, Paris, 1976, pp. 106–130.

[9] M. Karr, Affine relationships among variables of a program, Acta Inf. 6 (1976) 133–151.

[10] L. Chen, R. Li, X. Wu, J. Wang, Static analysis of list-manipulating programs via bit-vectors and numerical abstractions, in: SAC, ACM, 2013, pp. 1204–1210.

[11] A. Neumaier, Interval Methods for Systems of Equations, Cambridge University Press, 1990.

[12] A. Miné, The octagon abstract domain, Higher-Order and Symbolic Computation 19 (1) (2006) 31–100.

[13] B. Jeannet, A. Miné, Apron: A library of numerical abstract domains for static analysis, in: CAV, Vol. 5643 of LNCS, Springer, 2009, pp. 661–667.

[14] G. Lalire, M. Argoud, B. Jeannet, Interproc, http://pop-art.inrialpes.fr/people/bjeannet/ bjeannet-forge/interproc/.

[15] S. Magill, M. Tsai, P. Lee, Y. Tsay, Automatic numeric abstractions for heap-manipulating programs, in: POPL, ACM, 2010, pp. 211–222.

[16] N. Dor, M. Rodeh, M. Sagiv, Checking cleanness in linked lists, in: SAS, Vol. 1824 of LNCS, Springer, 2000, pp. 115–134.

[17] R. Manevich, E. Yahav, G. Ramalingam, M. Sagiv, Predicate abstraction and canonical abstraction for singly-linked lists, in: VMCAI, Vol. 3385 of LNCS, Springer, 2005, pp. 181–198.

[18] A. Bouajjani, C. Dragoi, C. Enea, M. Sighireanu, Abstract domains for automated reasoning about list-manipulating programs with infinite data, in: VMCAI, Vol. 7148 of LNCS, Springer, 2012, pp. 1–22.

[19] A. Podelski, T. Wies, Boolean heaps, in: SAS, Vol. 3672 of LNCS, Springer, 2005, pp. 268–283.

[20] D. Distefano, P.W.O'Hearn, H. Yang, A local shape analysis based on separation logic, in: TACAS, Vol. 3902 of LNCS, Springer, 2006, pp. 287–302.

[21] T. Lev-Ami, S. Sagiv, TVLA: A system for implementing static analyses, in: SAS, Vol. 1824 of Lecture Notes in Computer Science, Springer, 2000, pp. 280–301.

[22] P. Ferrara, R. Fuchs, U. Juhasz, TVAL+ : TVLA and value analyses together, in: SEFM, Vol. 7504 of Lecture Notes in Computer Science, Springer, 2012, pp. 63–77.

[23] P. Ferrara, Generic combination of heap and value analyses in abstract interpretation, in: VMCAI, Vol. 8318 of Lecture Notes in Computer Science, Springer, 2014, pp. 302–321.

[24] S. Gulwani, A. Tiwari, An abstract domain for analyzing heap-manipulating low-level software, in: CAV, Vol. 4590 of LNCS, Springer, 2007, pp. 379–392.

[25] S. S. Ishtiaq, P. W. O'Hearn, Bi as an assertion language for mutable data structures, in: POPL, ACM, 2001, pp. 14–26.

[26] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: LICS, IEEE Computer Society, 2002, pp. 55–74.

[27] J. Berdine, C. Calcagno, P. W. O'Hearn, Smallfoot: Modular automatic assertion checking with separation logic, in: FMCO, Vol. 4111 of Lecture Notes in Computer Science, Springer, 2005, pp. 115–137.

[28] D. Distefano, P. W. O'Hearn, H. Yang, A local shape analysis based on separation logic, in: TACAS, Vol. 3920 of Lecture Notes in Computer Science, Springer, 2006, pp. 287–302.

[29] A. Gotsman, J. Berdine, B. Cook, Interprocedural shape analysis with separated heap abstractions, in: SAS, Vol. 4134 of Lecture Notes in Computer Science, Springer, 2006, pp. 240–260.

[30] K. Dudka, P. Peringer, T. Vojnar, Predator: A practical tool for checking manipulation of dynamic data structures using separation logic, in: CAV, Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 372–378.

[31] P. Habermehl, L. Holík, A. Rogalewicz, J. Simácek, T. Vojnar, Forest automata for verification of heap manipulation, in: CAV, Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 424–440.

[32] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, T. Vojnar, Programs with lists are counter automata, Formal Methods in System Design 38 (2) (2011) 158–192.

[33] S. Qin, G. He, C. Luo, W.-N. Chin, X. Chen, Loop invariant synthesis in a combined abstract domain, J. Symb. Comput. 50 (2013) 386–408.

[34] S. Qin, G. He, W.-N. Chin, H. Yang, Invariants synthesis over a combined domain for automated program verification, in: Theories of Programming and Formal Methods, Vol. 8051 of LNCS, Springer, 2013, pp. 304–325.

[35] W.-N. Chin, C. David, H. H. Nguyen, S. Qin, Automated verification of shape, size and bag properties via user-defined predicates in separation logic, Sci. Comput. Program. 77 (9) (2012) 1006–1036.

[36] S. Qin, G. He, C. Luo, W.-N. Chin, H. Yang, Automatically refining partial specifications for heap-manipulating programs, Sci. Comput. Program. 82 (2014) 56–76.

[37] S. Magill, J. Berdine, E. M. Clarke, B. Cook, Arithmetic strengthening for shape analysis, in: SAS, Vol. 4634 of LNCS, Springer, 2007, pp. 419–436.

[38] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: POPL, ACM Press, 2002, pp. 58–70.

[39] H. H. Nguyen, C. David, S. Qin, W.-N. Chin, Automated verification of shape and size properties via separation logic, in: VMCAI, Vol. 4349 of LNCS, Springer, 2007, pp. 251–266.

[40] A. Bouajjani, P. Habermehl, P. Moro, T. Vojnar, Verifying programs with dynamic 1-selector-linked structures in regular model checking, in: TACAS, Vol. 3440 of Lecture Notes in Computer Science, Springer, 2005, pp. 13–29.

[41] M. Bozga, R. Iosif, On flat programs with lists, in: VMCAI, Vol. 4349 of Lecture Notes in Computer Science, Springer, 2007, pp. 122–136.

[42] S. K. Lahiri, S. Qadeer, Verifying properties of well-founded linked lists, in: POPL, ACM Press, 2006, pp. 115–126.

Appendix A.

Theorem 1. *Given two list nodes n_1, n_2 such that $\mathbf{vec}_{n_1} \neq \mathbf{vec}_{n_2}$ and $\mathbf{vec}_{n_1} \neq \mathbf{0}$, there exists one path from n_1 to n_2 if and only if $\mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2}$ holds.*

Proof. $\Rightarrow$):

Suppose that there exists one path from n_1 to n_2. We then have $\mathbf{vec}_{n_1} \subseteq \mathbf{vec}_{n_2}$ according to the Definition 2. Since $\mathbf{vec}_{n_1} \neq \mathbf{vec}_{n_2}$ by hypothesis, we then have $\mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2}$.

$\Leftarrow$):

Suppose $\mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2}$ holds. Then we have $(\mathbf{vec}_{n_1} \cap \mathbf{vec}_{n_2}) \neq \emptyset$. Let $i \in \mathcal{I}_{\mathbf{vec}_{n_1}} \cap \mathcal{I}_{\mathbf{vec}_{n_2}}$. There must exist a sequence of nodes $n_{k_0}, n_{k_1}, \cdots, n_{k_p} \in N$ satisfying $n_{k_0} = V_i, n_{k_p} = n_1$ and $\langle n_k, n_{k+1} \rangle \in E$ for each $k_0 \leq k < k_p$. Similarly, there must also exist a sequence of nodes $n_{t_0}, n_{t_1}, \cdots, n_{t_q} \in N$ satisfying $n_{t_0} = V_i, n_{t_q} = n_2$ and $\langle n_t, n_{t+1} \rangle \in E$ for each $t_0 \leq t < t_q$. Then we consider case by case as follows:

- Assume $p = q$. For singly linked lists, there is only one *next* node for each list node. Starting from the same node V_i and traverse the same number of *next* edges, we should always reach the same node. Hence, n_1 is identical with n_2, which conflicts with the hypothesis $\mathbf{vec}_{n_1} \neq \mathbf{vec}_{n_2}$.
- If $p < q$, then there must exist $t_p, \cdots, t_q$ such that $n_{t_p} = n_1$. Then we get one path from n_1 to n_2.
- If $p > q$, then there must exist $k_q, \cdots, k_p$ such that $n_{k_q} = n_2$. Then we get one path from n_2 to n_1. According to the discussion in the $\Rightarrow$ part, we then have $\mathbf{vec}_{n_2} \subset \mathbf{vec}_{n_1}$, which conflicts with the hypothesis $\mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2}$.

To summarize, there must exist one path from n_1 to n_2 if $\mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2}$. □

Theorem 2. *The set of VRVs of a singly-linked list is consistent.*

Proof. Consider two arbitrary VRVs $\mathbf{vec}_{n_1}, \mathbf{vec}_{n_2}$ from the set of VRVs of a singly-linked list such that $\mathbf{vec}_{n_1} \neq \mathbf{vec}_{n_2}$. There exists only two possibilities: $\mathbf{vec}_{n_1} \cap \mathbf{vec}_{n_2} = \emptyset$ or $\mathbf{vec}_{n_1} \cap \mathbf{vec}_{n_2} \neq \emptyset$. When $\mathbf{vec}_{n_1} \cap \mathbf{vec}_{n_2} \neq \emptyset$, following the same line of reasoning as in the $\Leftarrow$ part of the proof of Theorem 1, it is not hard to prove that $\mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2} \vee \mathbf{vec}_{n_2} \subset \mathbf{vec}_{n_1}$. □

Theorem 3. *A consistent set of VRVs Γ satisfies $|\Gamma| \leq 2|V|$.*

Proof. Let $\Gamma' = \Gamma \setminus \{\mathbf{0}\}$. And we will prove that $|\Gamma'| \leq 2|V| - 1$.

We first define the sets of predecessors and direct predecessors respectively as following: $pred^*(\mathbf{vec}) = \{\mathbf{vec}' \in \Gamma' \mid \mathbf{vec}' \subset \mathbf{vec}\}$ and $pred(\mathbf{vec}) = \{\mathbf{vec}' \in pred^*(\mathbf{vec}) \mid \nexists \mathbf{vec}'' \in pred^*(\mathbf{vec}).\mathbf{vec}' \subset \mathbf{vec}''\}$. And it's easy to see that we have the following properties

① For any distinct $\mathbf{vec}_1, \mathbf{vec}_2 \in pred(\mathbf{vec})$, we have that $\mathbf{vec}_1 \cap \mathbf{vec}_2 = \emptyset$.

② For any distinct $\mathbf{vec}_1, \mathbf{vec}_2 \in pred(\mathbf{vec})$ and $\mathbf{vec}'_1 \in pred^*(\mathbf{vec}_1), \mathbf{vec}'_2 \in pred^*(\mathbf{vec}_2)$, we have $\mathbf{vec}'_1 \cap \mathbf{vec}'_2 = \emptyset$.

Now,we will prove by induction that for each $\mathbf{vec} \in \Gamma'$, its predecessor set $pred^*(\mathbf{vec})$ satisfies $|pred^*(\mathbf{vec})| \leqslant 2(|\mathcal{I}_{\mathbf{vec}}| - 1)$.

1. The basis:

   - When $|\mathcal{I}_{\mathbf{vec}}| = 1$, $pred^*(\mathbf{vec})$ must be an empty set and thus satisfies $|pred^*(\mathbf{vec})| \leqslant 2(|\mathcal{I}_{\mathbf{vec}}| - 1)$.

   - When $|\mathcal{I}_{\mathbf{vec}}|=2$, suppose $\mathcal{I}_{\mathbf{vec}} = \{m, n\}$. For each $\mathbf{vec}' \in pred^*(\mathbf{vec})$, $\mathcal{I}_{\mathbf{vec}'}$ must be exactly $\{m\}$ or $\{n\}$. Hence $pred^*(\mathbf{vec})$ can contain at most two elements and thus $|pred^*(\mathbf{vec})| \leqslant 2(|\mathcal{I}_{\mathbf{vec}}| - 1)$ holds.

2. The inductive step: Assume $|pred^*(\mathbf{vec})| \leqslant 2(|\mathcal{I}_{\mathbf{vec}}| - 1)$ holds whenever $|\mathcal{I}_{\mathbf{vec}}| \leqslant M$. Then for the case of $|\mathcal{I}_{\mathbf{vec}}| = M + 1$, we have:

   - If $pred(\mathbf{vec}) = \emptyset$, $pred^*(\mathbf{vec})$ must also be an empty set. Then $|pred^*(\mathbf{vec})| \leqslant 2(|\mathcal{I}_{\mathbf{vec}}| - 1)$ holds obviously.

   - Suppose $pred(\mathbf{vec}) = \{\mathbf{vec}_1, ..., \mathbf{vec}_s\}$. According to properties ① ②, we know that for every $\mathbf{vec}' \in pred^*(\mathbf{vec})$, either $\mathbf{vec}' \in pred(\mathbf{vec})$ holds, or there exists one and only one $1 \leqslant i \leqslant s$ such that $\mathbf{vec}' \in pred^*(\mathbf{vec}_i)$. On the other hand, for each $1 \leqslant i \leqslant s$, $pred^*(\mathbf{vec}_i) \subseteq pred^*(\mathbf{vec})$ holds obviously. So we get $|pred^*(\mathbf{vec})| = \sum_{1 \leqslant i \leqslant s} |pred^*(\mathbf{vec}_i)| + |pred(\mathbf{vec})|$. For each $1 \leqslant i \leqslant s$, $|\mathcal{I}_{\mathbf{vec}_i}| \leqslant M$ holds because of $\mathcal{I}_{\mathbf{vec}_i} \subset \mathcal{I}_{\mathbf{vec}}$. According to the inductive assumption, we know that $|pred^*(\mathbf{vec}_i)| \leqslant 2(|\mathcal{I}_{\mathbf{vec}_i}| - 1)$. Hence $|pred^*(\mathbf{vec})| = \sum_{1 \leqslant i \leqslant s} |pred^*(\mathbf{vec}_i)| + s \leqslant \sum_{1 \leqslant i \leqslant s} 2(|\mathcal{I}_{\mathbf{vec}_i}| - 1) +$

$$s = 2 \sum_{1 \leqslant i \leqslant s} |\mathcal{I}_{\mathbf{vec}_i}| - s.$$

- When $s \geqslant 2$, for all $1 \leqslant i, j \leqslant s \wedge i \neq j$, we have $\mathcal{I}_{\mathbf{vec}_i} \subset \mathcal{I}_{\mathbf{vec}}$ and $\mathcal{I}_{\mathbf{vec}_j} \subset \mathcal{I}_{\mathbf{vec}}$. We also know that $\mathcal{I}_{\mathbf{vec}_i} \cap \mathcal{I}_{\mathbf{vec}_j} = \emptyset$ according to ①. So we have $\sum_{1 \leqslant i \leqslant s} |\mathcal{I}_{\mathbf{vec}_i}| \leqslant |\mathcal{I}_{\mathbf{vec}}|$. Finally, we get the result that $|pred^*(\mathbf{vec})| \leqslant 2|\mathcal{I}_{\mathbf{vec}}| - s \leqslant 2(|\mathcal{I}_{\mathbf{vec}}| - 1)$.
- When $s = 1$, $\mathbf{vec}$ has only one predecessor. Suppose $\mathbf{vec}_i$ is the unique predecessor. Then $|\mathcal{I}_{\mathbf{vec}_i}| < |\mathcal{I}_{\mathbf{vec}}|$ holds because $\mathcal{I}_{\mathbf{vec}_i} \subset \mathcal{I}_{\mathbf{vec}}$. Thus, $|pred^*(\mathbf{vec})| \leqslant 2(|\mathcal{I}_{\mathbf{vec}}| - 1)$ also holds for this case.

Therefore, $|pred^*(\mathbf{vec})| \leqslant 2(|\mathcal{I}_{\mathbf{vec}}| - 1)$ holds for every $\mathbf{vec} \in \Gamma'$.

We say that $\mathbf{vec} \in \Gamma'$ is a maximal VRV of Γ' if there doesn't exist one $\mathbf{vec}' \in \Gamma'$ satisfying $\mathbf{vec} \subset \mathbf{vec}'$. Supposing all the maximal VRVs in Γ' are $\mathbf{vec}_1, ..., \mathbf{vec}_t$. It's obviously that $\mathcal{I}_{\mathbf{vec}_i} \cap \mathcal{I}_{\mathbf{vec}_j} = \emptyset$ holds for all $1 \leqslant i, j \leqslant t$ and $i \neq j$. For each $x \in \mathcal{I}_{\mathbf{vec}_i}$, it must satisfy $0 \leqslant x < |V|$. So $\sum_{1 \leqslant i \leqslant t} |\mathcal{I}_{\mathbf{vec}_i}| \leqslant |V|$ holds. According to ②, for each $\mathbf{vec}' \in \Gamma'$, either $\mathbf{vec}' \in \{\mathbf{vec}_1, ..., \mathbf{vec}_t\}$ holds, or there exists one and only one $1 \leqslant i \leqslant t$ satisfying $\mathbf{vec}' \in pred^*(\mathbf{vec}_i)$. So we know that $|\Gamma'| = \sum_{1 \leqslant i \leqslant t} |pred^*(\mathbf{vec}_i)| + t \leqslant \sum_{1 \leqslant i \leqslant t} 2(|\mathcal{I}_{\mathbf{vec}_i}| - 1) + t = 2 \sum_{1 \leqslant i \leqslant t} |\mathcal{I}_{\mathbf{vec}_i}| - t \leqslant 2|V| - t \leqslant 2|V| - 1$. □

Theorem 4. *In singly-linked lists, a list node could be on at most one cycle.*

Proof. Recall that a list node in a singly-linked list has one *next* field and has at most one successive node. If a list node lies on two cycles, then some list node in the same singly-linked list should have two successive nodes, which is impossible. □

Theorem 5. *A set of circular VRVs of singly-linked lists is circularly consistent.*

Proof. Let $\breve{\Gamma}$ be a set of circular VRVs of singly-linked lists. Let us consider two arbitrary $\mathbf{v\breve{e}c}_{n_1}, \mathbf{v\breve{e}c}_{n_2} \in \breve{\Gamma}$. Let $\mathbf{vec}_{n_1} \overset{\text{def}}{=} \mathbf{v\breve{e}c}_{n_1} >> 1, \mathbf{vec}_{n_2} \overset{\text{def}}{=} \mathbf{v\breve{e}c}_{n_2} >> 1$. In other words, we get $\mathbf{vec}_{n_1}, \mathbf{vec}_{n_2}$ by removing the circular flag bit from $\mathbf{v\breve{e}c}_{n_1}, \mathbf{v\breve{e}c}_{n_2}$.

1) If $\mathbf{v\breve{e}c}_{n_1}, \mathbf{v\breve{e}c}_{n_2}$ respectively belong to two different singly-linked lists that are not connected, then it is easy to see that $\mathbf{vec}_{n_1} \& \mathbf{vec}_{n_2} = \mathtt{0x0}$.

2) If $\check{\mathbf{vec}}_{n_1}, \check{\mathbf{vec}}_{n_2}$ belong to a connected graph, we consider that $\check{\mathbf{vec}}_{n_1}, \check{\mathbf{vec}}_{n_2}$ are defined by cutting the cycle and a new variable V_c is introduced to point to the destination node of the *next* edge that is cut. After cutting, the connected graph turns to a set of non-circular singly-linked lists that is consistent over variable set $V \cup \{V_c\}$. Hence, according to Definition 4 and Theorem 2, it holds that $\check{\mathbf{vec}}_{n_1} \cap \check{\mathbf{vec}}_{n_2} = \emptyset \vee \check{\mathbf{vec}}_{n_1} \subset \check{\mathbf{vec}}_{n_2} \vee \check{\mathbf{vec}}_{n_2} \subset \check{\mathbf{vec}}_{n_1}$. When both $\check{\mathbf{vec}}_{n_1}, \check{\mathbf{vec}}_{n_2}$ belong to a connected graph, $\check{\mathbf{vec}}_{n_1} \cap \check{\mathbf{vec}}_{n_2} = \emptyset$ implies that $\mathbf{vec}_{n_1} \& \mathbf{vec}_{n_2} = \mathtt{0x0}$. Hence, $\mathbf{vec}_{n_1} \cap \mathbf{vec}_{n_2} = \mathtt{0x0} \vee \check{\mathbf{vec}}_{n_1} \subset \check{\mathbf{vec}}_{n_2} \vee \check{\mathbf{vec}}_{n_2} \subset \check{\mathbf{vec}}_{n_1}$ holds. □

Theorem 6. *A circularly consistent set of circular VRVs* $\check{\Gamma}$ *satisfies* $|\check{\Gamma}| \leq 2|V| + 2$.

Proof. In memory, there may exist several unconnected linked lists at the same time. The shape graph of those lists could be divided into several maximal connected subgraphs. And one pointer variable can reach only one of the maximal connected subgraphs. Hence, we now consider a circularly consistent set $\check{\Gamma}_1$ of circular VRVs of one arbitrary maximal connected subgraph of the shape graph, which could be reached by a subset of pointer variables V'. Let $\check{\Gamma}'_1 = \check{\Gamma}_1 \setminus \{\mathbf{0}\}$. And we will prove that $|\check{\Gamma}'_1| \leq 2|V'| + 1$.

As we have explained, the set of circular VRVs $\check{\Gamma}'_1$ can be considered as the representation of non-circular VRVs for non-circular singly-linked list over the variable set $V' \cup \{V_c\}$. Hence, according to the proof of Theorem 3, we have $|\check{\Gamma}'_1| \leq 2(|V'| + 1) - 1$, i.e., $|\check{\Gamma}'_1| \leq 2|V'| + 1$.

And we use $\mathbf{0}$ to denote the set of list nodes that cannot be reached by any variable. Hence, overall, a circularly consistent set of circular VRVs $\check{\Gamma}$ satisfies $|\check{\Gamma}| \leq 2|V| + 2$. □