Static Analysis of List-Manipulating Programs via Bit-Vectors and Numerical Abstractions

Liqian Chen[1,2] Renjian Li[1] Xueguang Wu [1] Ji Wang[1]

[1]National University of Defense Technology, Changsha, China
[2]National Lab. for Parallel and Distributed Processing, Changsha, China

21/03/2013 – ACM SAC'13 (Track SVT)

Overview

- Motivation
- A combined abstract domain for lists
- Analysis of list-manipulating programs
- Conclusion

Motivation

Motivation

Linked list: a basic dynamic data structure

- commonly used in OS kernels, network protocols, ...
- **errors**: memory leaks, dangling references, double free, null pointer dereference, ...

Analysis of list manipulating programs

- **problem**: high complexity
- **solution**: abstraction to make the problem tractable
  - abstraction according to the characteristics of lists
    $\longrightarrow$ simplify the problem & precise enough
  - **shape abstraction + numerical abstraction**
    $\longrightarrow$ numerical related properties over lists

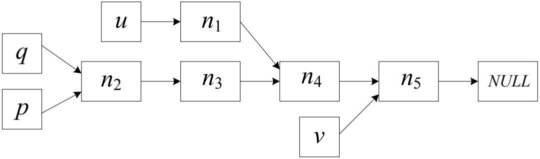
Motivation

Idea: combining shape and numerical abstractions
under the framework of **abstract interpretation**

- a combined abstract domain for singly-linked lists
  - shape: bit vectors
  - numerical: polyhedra, octagons, intervals, ...
- analysis of list-manipulating programs based on this domain

A combined abstract domain for lists

Concrete heap state



Shape graph: $\langle N, V, E \rangle$

- $N = \{u, v, p, q, n_1, n_2, n_3, n_4, n_5\}$
- $V = \{u, v, p, q\}$
- $E = \{\langle u, n_1 \rangle, \langle p, n_2 \rangle, \langle q, n_2 \rangle, \langle v, n_5 \rangle, \langle n_1, n_4 \rangle, \langle n_2, n_3 \rangle, \langle n_3, n_4 \rangle, \langle n_4, n_5 \rangle, \langle n_5, NULL \rangle\}$

Limitations of shape graphs:

- high memory costs (explicit storage)
- lists with symbolic length

Shape abstraction for lists

Definition (*Reach* predicate)

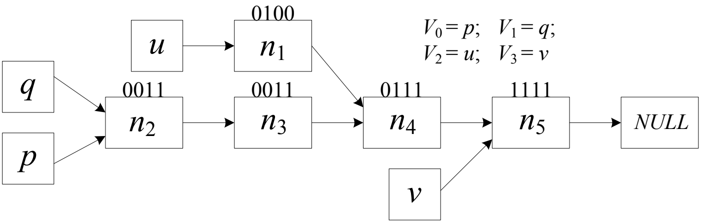
- $Reach(n, n') \triangleq \exists k \in \mathbb{N}. \forall 0 \leqslant i \leqslant k. n_i \in N.$
  $$n_0 = n \wedge n_k = n' \wedge \forall 0 \leqslant j < k. \langle n_j, n_{j+1} \rangle \in E$$
- i.e., $Reach(n, n') = true$ iff there exists a path from n to n'

Definition (Variable Reachability Vector)

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

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

Shape abstraction for lists



Variable Reachability Vector: describe reachability properties of all variables to nodes

- each VRV can be considered as an abstract node

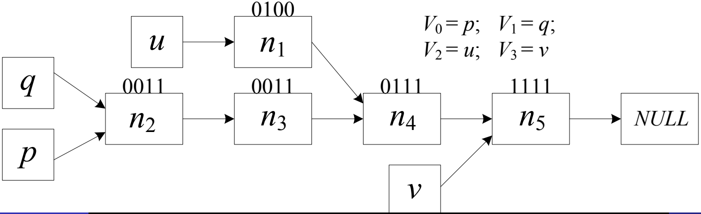
Shape abstraction for lists

Reachability information from VRVs

Definition

- Let Γ denote the set of VRVs of all nodes. For arbitrary $\mathbf{vec} \in \Gamma$, let $\mathcal{I}_{\mathbf{vec}}$ denote the set of the 1-bits in $\mathbf{vec}$:
$$\mathcal{I}_{\mathbf{vec}} \triangleq \{i \in \mathbb{N} \mid \mathbf{vec}[i] = 1\}$$
- If $i \in \mathcal{I}_{\mathbf{vec}}$, then V_i can reach (the corresponding nodes) of $\mathbf{vec}$, denoted as $V_i \in \mathbf{vec}$

E.g., $\mathcal{I}_{0100} = \{2\}; \mathcal{I}_{0011} = \{0, 1\};$



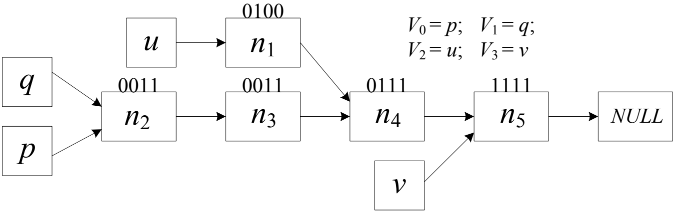
Shape abstraction for lists

Definition (Reachability relations between VRVs)

Given two VRVs $\mathbf{vec}_1, \mathbf{vec}_2$,

- if $\mathcal{I}_{\mathbf{vec}_1} \subseteq \mathcal{I}_{\mathbf{vec}_2}$, then $\mathbf{vec}_1$ can reach $\mathbf{vec}_2$ ($\mathbf{vec}_1 \subseteq \mathbf{vec}_2$)
- if $\mathcal{I}_{\mathbf{vec}_1} \subset \mathcal{I}_{\mathbf{vec}_2}$, then $\mathbf{vec}_1$ can strictly reach $\mathbf{vec}_2$ ($\mathbf{vec}_1 \subset \mathbf{vec}_2$)
- if $\mathcal{I}_{\mathbf{vec}_1} \cap \mathcal{I}_{\mathbf{vec}_2} = \emptyset$, then $\mathbf{vec}_1$ and $\mathbf{vec}_2$ can not reach each other ($\mathbf{vec}_1 \cap \mathbf{vec}_2 = \emptyset$)

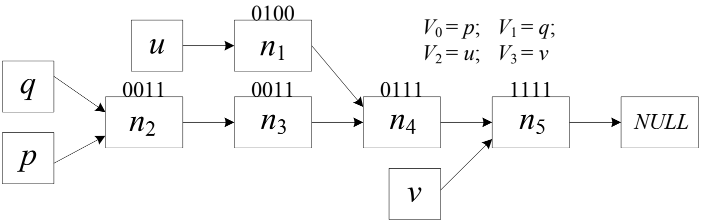
E.g., $\mathbf{vec}_{0100} \subset \mathbf{vec}_{0111}$; $\mathbf{vec}_{0011} \subset \mathbf{vec}_{0111}$; $\mathbf{vec}_{0100} \cap \mathbf{vec}_{0011} = \emptyset$;



Properties of VRVs

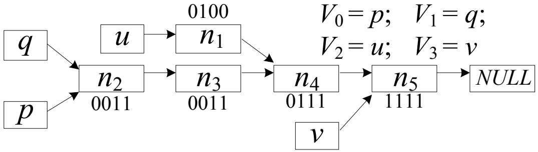
E.g., Given a set of VRVs $\{0011, 0100, 0111, 1111\}$

- $\mathbf{vec}_{0011} \subset \mathbf{vec}_{0111} \subset \mathbf{vec}_{1111}$
- p points to 0011
- p, q are alias
- p cannot reach the node pointed to by u



The set Γ of VRVs of a singly-linked list satisfies $|\Gamma| \leq 2|V|$

VRVs with counters



pquv	*num*
0011	2
0100	1
0111	1
1111	1

Definition

The set of VRVs with counters *VRVCs* $\mathbf{\Gamma}^+ \subseteq \mathbf{\Gamma} \times \mathbb{N}$ is defined as a set of 2-tuples $\langle \mathbf{vec}, num \rangle$ where

- $\mathbf{vec} \in \mathbf{\Gamma}$
- $num \in \mathbb{N}$: the number of the list nodes whose VRV is **vec**

Lists:

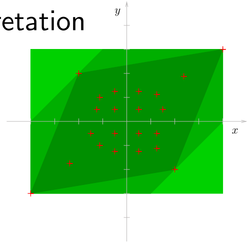
- **shape**: VRVs ← nodes; VRV reachability relations ← edges
- **numerical**: counters ← quantitative information of the nodes

Numerical abstraction for lists

Counter variables: auxiliary (non-negative) integer variables

- for each **vec** $\in$ *VRVs*, introduce a counter variable $t^{\mathbf{vec}} \in \mathbb{N}$
  - to record the number of the list nodes whose VRV is **vec**
- a special auxiliary variable $t^{0\ldots00} \in \mathbb{N}$
  - to specify memory leak when $t^{0\ldots00} > 0$
- variable ordering: $t^{0\ldots00} \prec t^{0\ldots01} \prec t^{0\ldots10} \prec \cdots \prec t^{1\ldots11}$
- a bijection between **vec** and $t^{\mathbf{vec}}$
- $\{\langle \mathbf{vec}, t^{\mathbf{vec}} \rangle \mid t^{\mathbf{vec}} > 0\}$ represents a list, if it is consistent

Numerical abstraction for lists



Numerical abstract domains in abstract interpretation

- infer relations among numerical variables
- examples
  - intervals ($a \leq x \leq b$)
  - octagons ($\pm x \pm y \leq c$)
  - polyhedra ($\Sigma_k a_k x_k \leq b$)

Chosen numerical abstract domains for counter variables $t^{\mathbf{vec}}$

- intervals ($a \leq x \leq b$)
- affine equalities ($\Sigma_k a_k x_k = b$)

Analysis of list-manipulating programs

Analysis of list-manipulating programs

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

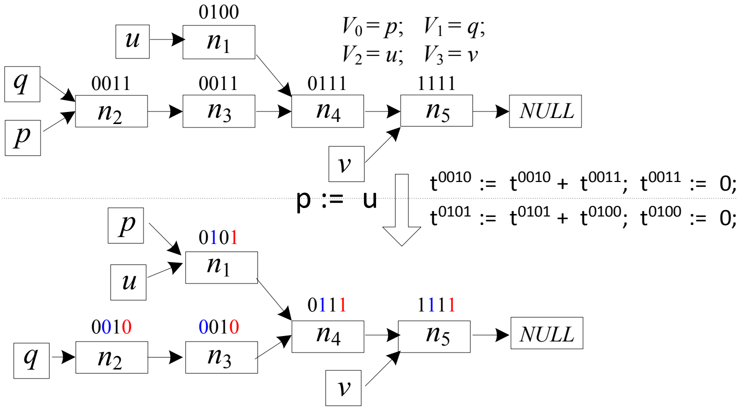
Domain operations: on top of shape and numerical abstraction

- inclusion test $\sqsubseteq$
- join $\sqcup$
- widening ∇
- transfer functions τ
    - condition test
    - assignment
- ...

$[\![p := q]\!]^\sharp$

Let $\mathbf{vec}' \stackrel{\text{def}}{=} \mathbf{vec}/_{\{p\}\leftarrow q}$. For each $\mathbf{vec} \in \Gamma$ s.t. $\mathbf{vec}' \neq \mathbf{vec}$, we build numerical statements:

$$if\,(t^{\mathbf{vec}} \geq 1)\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$$



vuqp	*num*
0011	2
0100	1
0111	1
1111	1

vuqp	*num*
0010	2
0101	1
0111	1
1111	1

Example analysis

```
void copy_and_delete(List* xList) { /* ♡ : ∀t^vec.NoOccurenceOf vec implies t^vec = 0 */
  /* assume \length(xList)==9; */
  List* yList, pList, qList;          /* pList ≺ qList ≺ xList ≺ yList */
  / * t^0100 = 9; t^0100 ∈ [9,9]; ♡ * /
  yList = xList;   qList = pList = null;
  / * t^0100 + t^1100 = 9, t^0011 + t^1100 = 9; t^0100 ∈ [1,9], t^1100 ∈ [0,9], t^0011 ∈ [0,9]; ♡ * /
  while (yList != null){
  / * t^0100 + t^1100 = 9, t^0011 + t^1100 = 9; t^0100 ∈ [1,8], t^1100 ∈ [1,9], t^0011 ∈ [0,8]; ♡ * /
      yList = yList → next; qList = malloc();
      qList → next = pList; pList = qList;}
  / * t^0011 = 9, t^0100 = 9; t^0011 ∈ [9,9], t^0100 ∈ [9,9]; ♡ * /
  yList = xList;
  / * t^0011 − t^1100 = 0; t^0011 ∈ [0,9], t^1100 ∈ [0,9]; ♡ * /
  while (yList != null){
  / * t^0011 − t^1100 = 0; t^0011 ∈ [1,9], t^1100 ∈ [1,9]; ♡ * /
      yList = yList → next; qList = qList → next;
      free(xList); free(pList); xList = yList; pList = qList;
  } / * ∀vec.t^vec = 0 * /
}
```

Example analysis

```
void copy_and_delete(List* xList) { /* ♡ : ∀t^vec.NoOccurenceOf vec implies t^vec = 0 */
    /* assume \length(xList)==9; */
    List* yList, pList, qList;          /* pList ≺ qList ≺ xList ≺ yList */
    yList = xList;    qList = pList = null;
    while (yList != null){
        yList = yList → next; qList = malloc();
        qList → next = pList; pList = qList;}
    yList = xList;
    while (yList != null){
        /* {t^0011 − t^1100 = 0; t^0011 ∈ [1,9], t^1100 ∈ [1,9]}; ♡ */
        yList = yList → next; qList = qList → next;
        free(xList); free(pList); xList = yList; pList = qList;
    }
}
```

- *pList*, *qList* are alias; *xList*, *yList* are alias
- the length of *pList* equals to that of *xList*
- no null pointer dereference

Example analysis

```
void copy_and_delete(List* xList) { /* ♡ : ∀t^vec.NoOccurenceOf vec implies t^vec = 0 */
    /* assume \length(xList)==9; */
    List* yList, pList, qList;          /* pList ≺ qList ≺ xList ≺ yList */
    yList = xList;    qList = pList = null;
    while (yList != null){
        yList = yList → next; qList = malloc();
        qList → next = pList; pList = qList;
    yList = xList;
    while (yList != null){
        yList = yList → next; qList = qList → next;
        free(xList); free(pList); xList = yList; pList = qList;
    } /* ∀vec.t^vec = 0; ♡ */
}
```

- all heap cells are freed

Example analysis

```
void copy_and_delete(List* xList) { /* ♡ : ∀t^vec.NoOccurenceOf vec implies t^vec = 0 */
    /* assume \length(xList)==9; */
    List* yList, pList, qList;          /* pList ≺ qList ≺ xList ≺ yList */
    yList = xList;    qList = pList = null;
    while (yList != null){
        yList = yList → next; qList = malloc();
        qList → next = pList; pList = qList;
    yList = xList;
    while (yList != null){
        yList = yList → next; qList = qList → next;
        free(xList); free(pList); xList = yList; pList = qList;
    }
}
```

- Global invariants: $t^{0\cdots0} \equiv 0 \rightsquigarrow$ no memory leak

Conclusion

Summary: analysis of lists via abstract interpretation

- **main idea**: combining shape and numerical abstractions
- a combined abstract domain for lists
  - the **structural** information of the shape: bit vectors
    - each bit-vector represents a list segment
  - the **number** of nodes in a segment: numerical abstract domains
    - a counter variable to record the number of nodes in a list segment

Future work

- reasoning over the content of lists (e.g., lists of integers)
  - enable infering advanced properties such as sortedness, no duplicated elements