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

Liqian Chen\textsuperscript{1,2} Renjian Li\textsuperscript{1} Xueguang Wu \textsuperscript{1} Ji Wang\textsuperscript{1}

\textsuperscript{1}National University of Defense Technology, Changsha, China
\textsuperscript{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
    → simplify the problem & precise enough
  - **shape abstraction + numerical abstraction**
    → 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)

- \( \text{Reach}(n, n') \triangleq \exists k \in \mathbb{N}. \forall 0 \leq i \leq k. n_i \in N. \)
  \[
  n_0 = n \wedge n_k = n' \wedge \forall 0 \leq j < k. \langle n_j, n_{j+1} \rangle \in E
  \]
- i.e., \( \text{Reach}(n, n') = \text{true} \) 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) \( \text{vec}_n \in \{0, 1\}^{|V|} \) that is a bit-vector of length \( |V| \), where
- \( \text{vec}_n[i] = 1 \) iff \( \text{Reach}(V_i, n) = \text{true} \)
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 $\Gamma$ denote the set of VRVs of all nodes. For arbitrary $\text{vec} \in \Gamma$, let $\mathcal{I}_{\text{vec}}$ denote the set of the 1-bits in $\text{vec}$:
  \[ \mathcal{I}_{\text{vec}} \triangleq \{ i \in \mathbb{N} \mid \text{vec}[i] = 1 \} \]

- If $i \in \mathcal{I}_{\text{vec}}$, then $V_i$ can reach (the corresponding nodes) of $\text{vec}$, denoted as $V_i \in \text{vec}$

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

Definition (Reachability relations between VRVs)

Given two VRVs \( \text{vec}_1, \text{vec}_2 \),

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

E.g., \( \text{vec}_{0100} \subset \text{vec}_{0111} ; \text{vec}_{0011} \subset \text{vec}_{0111} ; \text{vec}_{0100} \cap \text{vec}_{0011} = \emptyset \);

Diagram:

- \( p \) to \( n_1 \) to \( n_0 \)
- \( q \) to \( u \) to \( n_1 \)
- \( u \) to \( n_2 \) to \( n_3 \) to \( n_4 \) to \( n_5 \) to \( \text{NULL} \)
- \( \mathcal{V}_0 = p ; \mathcal{V}_1 = q ; \mathcal{V}_2 = u ; \mathcal{V}_3 = v \)
Properties of VRVs

E.g., Given a set of VRVs \(\{0011, 0100, 0111, 1111\}\)
- \(\text{vec}_{0011} \subseteq \text{vec}_{0111} \subseteq \text{vec}_{1111}\)
- \(p\) points to 0011
- \(p, q\) are alias
- \(p\) cannot reach the node pointed to by \(u\)

The set \(\Gamma\) of VRVs of a singly-linked list satisfies \(|\Gamma| \leq 2|V|\)
VRVs with counters

Definition

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

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

Lists:

- shape: VRVs $\leftarrow$ nodes; VRV reachability relations $\leftarrow$ edges
- numerical: counters $\leftarrow$ quantitative information of the nodes

Motivation

A combined abstract domain for lists

Analysis of list-manipulating programs

shape abstraction for lists

numerical abstraction for lists
Counter variables: auxiliary (non-negative) integer variables

- for each \( \text{vec} \in \text{VRVs} \), introduce a counter variable \( t^{\text{vec}} \in \mathbb{N} \)
  - to record the number of the list nodes whose VRV is \( \text{vec} \)
- a special auxiliary variable \( t^{0\ldots00} \in \mathbb{N} \)
  - to specify memory leak when \( t^{0\ldots00} > 0 \)
- variable ordering: \( t^{0\ldots00} < t^{0\ldots01} < t^{0\ldots10} < \ldots < t^{1\ldots11} \)
- a bijection between \( \text{vec} \) and \( t^{\text{vec}} \)
- \( \{ \langle \text{vec}, t^{\text{vec}} \rangle \mid t^{\text{vec}} > 0 \} \) 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 $t^{vec}$

- 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

\[ p, q \in P\text{Var} \]

\[
\begin{align*}
\text{AsgnStmt} & := p := \text{null} \mid p := q \mid p := q \rightarrow \text{next} \mid p \rightarrow \text{next} := \text{null} \mid \\
& \quad p \rightarrow \text{next} := q \mid p := \text{malloc}() \mid \text{free}(p) \\
\text{Cond} & := p == q \mid p == \text{null} \mid \neg \text{Cond} \mid \text{Cond}_1 \lor \text{Cond}_2 \mid \\
& \quad \text{Cond}_1 \land \text{Cond}_2 \mid \text{true} \mid \text{false} \mid \text{brandom} \\
\text{BranchStmt} & := \text{if } \text{Cond} \text{ then } \{\text{Stmt};\}^* \text{ else } \{\text{Stmt};\}^* \text{ fi} \\
\text{WhileStmt} & := \text{while } \text{Cond} \text{ do } \{\text{Stmt};\}^* \text{ od} \\
\text{Stmt} & := \text{AsgnStmt} \mid \text{BranchStmt} \mid \text{WhileStmt} \\
\text{Program} & := \{\text{Stmt};\}^*
\end{align*}
\]

Domain operations: on top of shape and numerical abstraction

- inclusion test \( \sqsubseteq \)
- join \( \sqcup \)
- widening \( \nabla \)
- transfer functions \( \tau \)
  - condition test
  - assignment
  - ...

Liqian Chen
Static Analysis of Lists via Bit-Vectors and Numerical Abstractions
Let $\mathbf{vec}' = \mathbf{vec}/\{p\} \leftarrow q$. For each $\mathbf{vec} \in \Gamma$ s.t. $\mathbf{vec}' \neq \mathbf{vec}$, we build numerical statements:

$$\text{if } (t^{\mathbf{vec}} \geq 1) \{ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}}; t^{\mathbf{vec}} := 0; \}$$
void copy_and_delete(List* xList) { /* ∗: ∀vec. NoOccurrenceOf vec implies t^vec = 0 */
    /* assume \length(xList)==9; */
  1: List* yList, pList, qList; /* pList ≺ qList ≺ xList ≺ yList */
      /* t^{0100} = 9; t^{0100} ∈ [9, 9]; ∗ */
  2: 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]; ∗ */
  3: while (yList != null) {
      /* t^{0100} + t^{1100} = 9, t^{0011} + t^{1100} = 9; t^{0100} ∈ [1, 9], t^{1100} ∈ [1, 9], t^{0011} ∈ [0, 8]; ∗ */
        yList = yList → next; qList = malloc();
  4: qList → next = pList; pList = qList;}
      /* t^{0011} = 9, t^{0100} = 9; t^{0011} ∈ [9, 9], t^{0100} ∈ [9, 9]; ∗ */
  5: yList = xList;
      /* t^{0011} − t^{1100} = 0; t^{0011} ∈ [0, 9], t^{1100} ∈ [0, 9]; ∗ */
  6: while (yList != null) {
      /* t^{0011} − t^{1100} = 0; t^{0011} ∈ [1, 9], t^{1100} ∈ [1, 9]; ∗ */
        yList = yList → next; qList = qList → next;
  7: free(xList); free(pList); xList = yList; pList = qList;
  8: } /* ∀vec. t^{vec} = 0 */
}
Example analysis

```c
void copy_and_delete(List* xList) { /* \(\forall t^{vec}.\text{NoOccurrenceOf} \ vec \ implies \ t^{vec} = 0\) */
    /* assume \(\text{length}(xList)==9\); */
1: List* yList, pList, qList; /* \(pList < qList < xList < yList\) */
2: yList = xList; qList = pList = null;
3: while (yList != null){
4:     yList = yList \rightarrow next; qList = malloc();
5:     qList \rightarrow next = pList; pList = qList;
6: yList = xList;
7: while (yList != null){
    /* \(t^{0011} - t^{1100} = 0; t^{0011} \in [1, 9], t^{1100} \in [1, 9]\); \(\heartsuit\) */
8:     yList = yList \rightarrow next; qList = qList \rightarrow next;
9: free(xList); free(pList); xList = yList; pList = qList;
10: }
}
```

- \(pList, qList\) are alias; \(xList, yList\) are alias
- the length of \(pList\) equals to that of \(xList\)
- no null pointer dereference
Example analysis

```c
void copy_and_delete(List* xList) {
    /* ∃ vec. NoOccurrenceOf vec implies t^vec = 0 */
    /* assume ∀ t^vec. length(xList) == 9 */
    1: List* yList, pList, qList; /* pList ≺ qList ≺ xList ≺ yList */
    2: yList = xList; qList = pList = null;
    3: while (yList != null)
    4:     yList = yList → next; qList = malloc();
    5:     qList → next = pList; pList = qList;
    6: yList = xList;
    7: while (yList != null)
    8:     yList = yList → next; qList = qList → next;
    9: free(xList); free(pList); xList = yList; pList = qList;
10: } /* ∀ vec. t^vec = 0; ∃ */
}
```

- all heap cells are freed
Example analysis

```c
void copy_and_delete(List* xList) { /* ⊸ : ∀vec. NoOccurrenceOf vec implies \( t^vec = 0 \) */
    /* assume \( \text{length}(xList) = 9 \) */

1: List* yList, pList, qList; /* pList ≺ qList ≺ xList ≺ yList */
2: yList = xList; qList = pList = null;
3: while (yList != null) {
4:     yList = yList → next; qList = malloc();
5:     qList → next = pList; pList = qList;
6:     yList = xList;
7:     while (yList != null) {
8:         yList = yList → next; qList = qList → next;
9:         free(xList); free(pList); xList = yList; pList = qList;
10:    }
}

• Global invariants: \( t^{0\ldots0} \equiv 0 \) ⊸ 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 inferring advanced properties such as sortedness, no duplicated elements