Static Program Analysis

Lecture 20: Wrap-Up Interprocedural DFA & Pointer Analysis

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH AACHEN UNIVERSITY

noll@cs.rwth-aachen.de

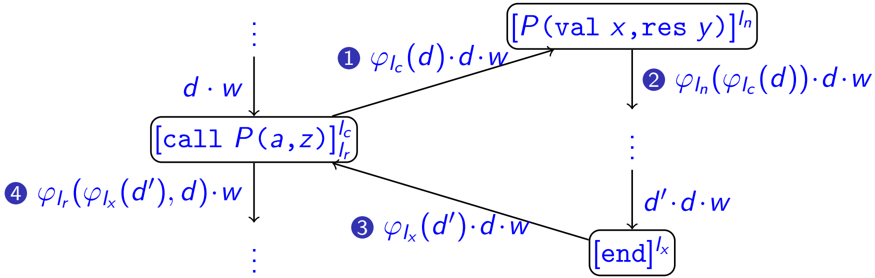
http://moves.rwth-aachen.de/teaching/ws-1415/spa/

Winter Semester 2014/15

Outline

Visualization of

1. $\hat{\varphi}_{l_c}(d \cdot w) = \varphi_{l_c}(d) \cdot d \cdot w$
2. $\hat{\varphi}_{l_n}(d' \cdot d \cdot w) = \varphi_{l_n}(d') \cdot d \cdot w$
3. $\hat{\varphi}_{l_x}(d' \cdot d \cdot w) = \varphi_{l_x}(d') \cdot d \cdot w$
4. $\hat{\varphi}_{l_r}(d' \cdot d \cdot w) = \varphi_{l_r}(d', d) \cdot w$



Formal Definition of Equation System

Dataflow equations:

$$\mathsf{AI}_l = \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\hat{\varphi}_{l_c}(\mathsf{AI}_{l_c}) \mid (l_c, l_n, l_x, l_r) \in \mathsf{iflow}\} & \text{if } l = l_n \text{ for some } (l_c, l_n, l_x, l_r) \in \mathsf{iflow} \\ \bigsqcup\{f_{l'}(\mathsf{AI}_{l'}) \mid (l', l) \in F\} & \text{otherwise} \end{cases}$$

(if l not a return label)

Node transfer functions:

$$f_l(w) = \begin{cases} \hat{\varphi}_{l_r}(\hat{\varphi}_{l_x}(F_{l_x}(\hat{\varphi}_{l_c}(w)))) & \text{if } l = l_c \text{ for some } (l_c, l_n, l_x, l_r) \in \mathsf{iflow} \\ \hat{\varphi}_l(w) & \text{otherwise} \end{cases}$$

(if l not an exit or return label)

Procedure transfer functions:

$$F_l(w) = \begin{cases} w & \text{if } l = l_n \text{ for some } (l_c, l_n, l_x, l_r) \in \mathsf{iflow} \\ \bigsqcup\{f_{l'}(F_{l'}(w)) \mid (l', l) \in F\} & \text{otherwise} \end{cases}$$

(if l occurs in some procedure)

As before: induces monotonic functional on lattice with ACC

$\Longrightarrow$ least fixpoint effectively computable

Outline

The Fixpoint Iteration

For the fixpoint iteration it is important that the auxiliary functions only operate (at most) on the two topmost elements of the stack:

Lemma 20.1

For every $l \in Lab$, $d \in D$, *and* $w \in D^*$,

$$f_l(d' \cdot d \cdot w) = f_l(d' \cdot d) \cdot w \text{ and } F_l(d' \cdot d \cdot w) = F_l(d' \cdot d)w$$

Proof.

see J. Knoop, B. Steffen: *The Interprocedural Coincidence Theorem*, Proc. CC '92, LNCS 641, Springer, 1992, 125–140 □

It therefore suffices to consider stacks with at most two entries, and so the fixpoint iteration ranges over "finitary objects".

Soundness and Completeness

The following results carry over from the intraprocedural case:

Theorem 20.2

Let $\hat{S} := (Lab, E, F, (\hat{D}, \hat{\sqsubseteq}), \hat{\iota}, \hat{\varphi})$ *be an interprocedural dataflow system.*

1. *(cf. Theorem 6.3)*

$$\mathrm{mvp}(\hat{S}) \mathrel{\hat{\sqsubseteq}} \mathrm{fix}(\Phi_{\hat{S}})$$

2. *(cf. Theorem 7.3)*

$$\mathrm{mvp}(\hat{S}) = \mathrm{fix}(\Phi_{\hat{S}}) \textit{ if all } \hat{\varphi}_l \textit{ are distributive}$$

Proof.

see J. Knoop, B. Steffen: *The Interprocedural Coincidence Theorem*, Proc. CC '92, LNCS 641, Springer, 1992, 125–140 □

Outline

Context-Sensitive Interprocedural DFA

- **Observation:** MVP and fixpoint solution maintain proper relationship between procedure calls and returns
- **But:** do not distinguish between different procedure calls

$$\mathsf{AI}_l = \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\hat{\varphi}_{l_c}(\mathsf{AI}_{l_c}) \mid (l_c, l_n, l_x, l_r) \in \text{iflow}\} & \text{if } l = l_n \text{ for some } (l_c, l_n, l_x, l_r) \in \text{iflow} \\ \bigsqcup\{f_{l'}(\mathsf{AI}_{l'}) \mid (l', l) \in F\} & \text{otherwise} \end{cases}$$

  - information about calling states combined for all call sites
  - procedure body only analyzed once using combined information
  - resulting information used at all return points

$\Longrightarrow$ "context-insensitive"

- **Alternative:** context-sensitive analysis
  - separate information for different call sites
  - implementation by "procedure cloning" (one copy for each call site)
  - more precise
  - more costly

Outline

Pointer Analysis

- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps
- **Problem:**
  - Programs with pointers and dynamically allocated data structures are error prone
  - Identify subtle bugs at compile time
  - Automatically prove correctness
- **Interesting properties of heap-manipulating programs:**
  - No null pointer dereference
  - No memory leaks
  - Preservation of data structures
  - Partial/total correctness

The Shape Analysis Approach

- **Goal:** determine the possible shapes of a dynamically allocated data structure at given program point
- **Interesting information:**
  - data types (to avoid type errors, such as dereferencing `nil`)
  - aliasing (different pointer variables having same value)
  - sharing (different heap pointers referencing same location)
  - reachability of nodes (garbage collection)
  - disjointness of heap regions (parallelizability)
  - shapes (lists, trees, absence of cycles, ...)
- **Concrete questions:**
  - Does `x.next` point to a shared element?
  - Does a variable `p` point to an allocated element every time `p` is dereferenced?
  - Does a variable point to an acyclic list?
  - Does a variable point to a doubly-linked list?
  - Can a loop or procedure cause a memory leak?
- **Here:** basic outline; details in [Nielson/Nielson/Hankin 2005, Sct. 2.6]

Outline

Extending the Syntax

Syntactic categories:

Category	Domain	Meta variable
Arithmetic expressions	*AExp*	a
Boolean expressions	*BExp*	b
Selector names	*Sel*	*sel*
Pointer expressions	*PExp*	p
Commands (statements)	*Cmd*	c

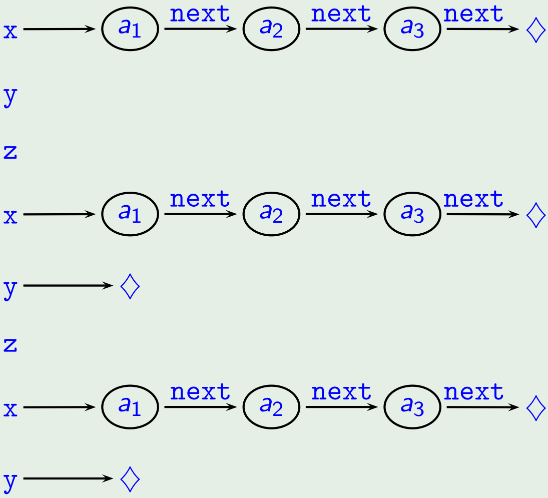
Context-free grammar:

$$a ::= z \mid x \mid a_1 + a_2 \mid \ldots \mid p \mid \texttt{nil} \in AExp$$
$$b ::= t \mid a_1 = a_2 \mid b_1 \wedge b_2 \mid \ldots \mid \texttt{is-nil}(p) \in BExp$$
$$p ::= x \mid x\,.\,sel$$
$$c ::= [\texttt{skip}]^l \mid [p \texttt{ := } a]^l \mid c_1 ; c_2 \mid \texttt{if } [b]^l \texttt{ then } c_1 \texttt{ else } c_2 \mid$$
$$\texttt{while } [b]^l \texttt{ do } c \mid [\texttt{malloc } p]^l \in Cmd$$

An Example

Example 20.3 (List reversal)

Program that reverses list pointed to by x and leaves result in y:



Outline

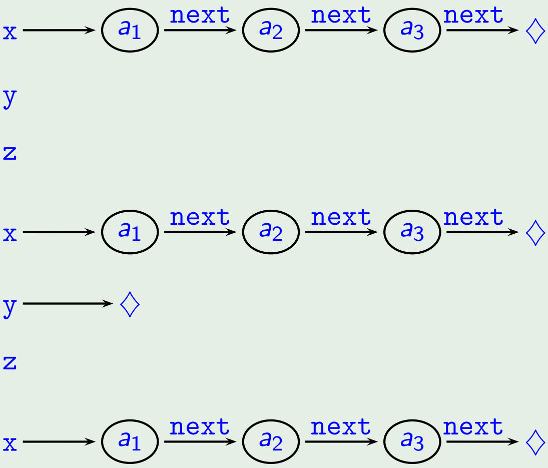
Shape Graphs I

Approach: representation of (infinitely many) concrete heap states by (finitely many) abstract shape graphs

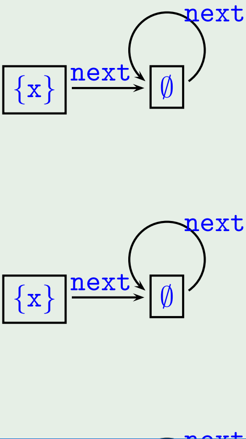
- abstract nodes X = sets of variables
- interpretation: $x \in X$ iff x points to concrete node represented by X
- $\emptyset$ represents all concrete nodes that are not directly addressed by pointer variables
- $x, y \in X$ (with $x \neq y$) indicate aliasing (as x and y point to the same concrete node)
- if $x.sel$ and y refer to the same heap address and if X, Y are abstract nodes with $x \in X$ and $y \in Y$, this yields abstract edge $X \xrightarrow{sel} Y$
- transfer functions transform (sets of) shape graphs

Example 20.4 (List reversal; cf. Example 20.3)

Concrete heap



Shape graph



Shape Graphs III

Definition 20.5 (Shape graph)

A shape graph $G = (S, H)$ consists of

- a set $S \subseteq 2^{Var}$ of abstract locations and
- an abstract heap $H \subseteq S \times Sel \times S$
  - notation: $X \xrightarrow{sel} Y$ for $(X, sel, Y) \in H$

with the following properties:

Disjointness: $X, Y \in S \implies X = Y$ or $X \cap Y = \emptyset$
(a variable can refer to at most one heap location)

Determinacy: $X \neq \emptyset$ and $X \xrightarrow{sel} Y$ and $X \xrightarrow{sel} Z \implies Y = Z$
(target location is unique if source node is unique)

SG denotes the set of all shape graphs.

Remark: the following example shows that determinacy requires $X \neq \emptyset$:

Concrete: $y \longrightarrow \bullet \xleftarrow{sel} \bullet$ Abstract: $\boxed{Y = \{y\}} \xleftarrow{sel} \boxed{X = \emptyset} \xrightarrow{sel} \boxed{Z = \{z\}}$

$z \longrightarrow \bullet \xleftarrow{sel} \bullet$

Shape Graphs and Concrete Heap Properties

Example 20.6

Let $G = (S, H)$ be a shape graph. Then the following concrete heap properties can be expressed as conditions on G:

- $\mathtt{x} \neq \mathtt{nil}$
  $\iff \exists X \in S : \mathtt{x} \in X$
- $\mathtt{x} = \mathtt{y} \neq \mathtt{nil}$ (aliasing)
  $\iff \exists Z \in S : \mathtt{x}, \mathtt{y} \in Z$
- $\mathtt{x.sel1} = \mathtt{y.sel2} \neq \mathtt{nil}$ (sharing)
  $\implies \exists X, Y, Z \in S : \mathtt{x} \in X, \mathtt{y} \in Y, X \stackrel{\mathtt{sel1}}{\longrightarrow} Z \stackrel{\mathtt{sel2}}{\longleftarrow} Y$
  ($\impliedby$ only valid if $Z \neq \emptyset$)