# Static Program Analysis

## Lecture 20: Wrap-Up Interprocedural DFA & Pointer Analysis

### Winter Semester 2016/17


**Thomas Noll**
**Software Modeling and Verification Group**
**RWTH Aachen University**

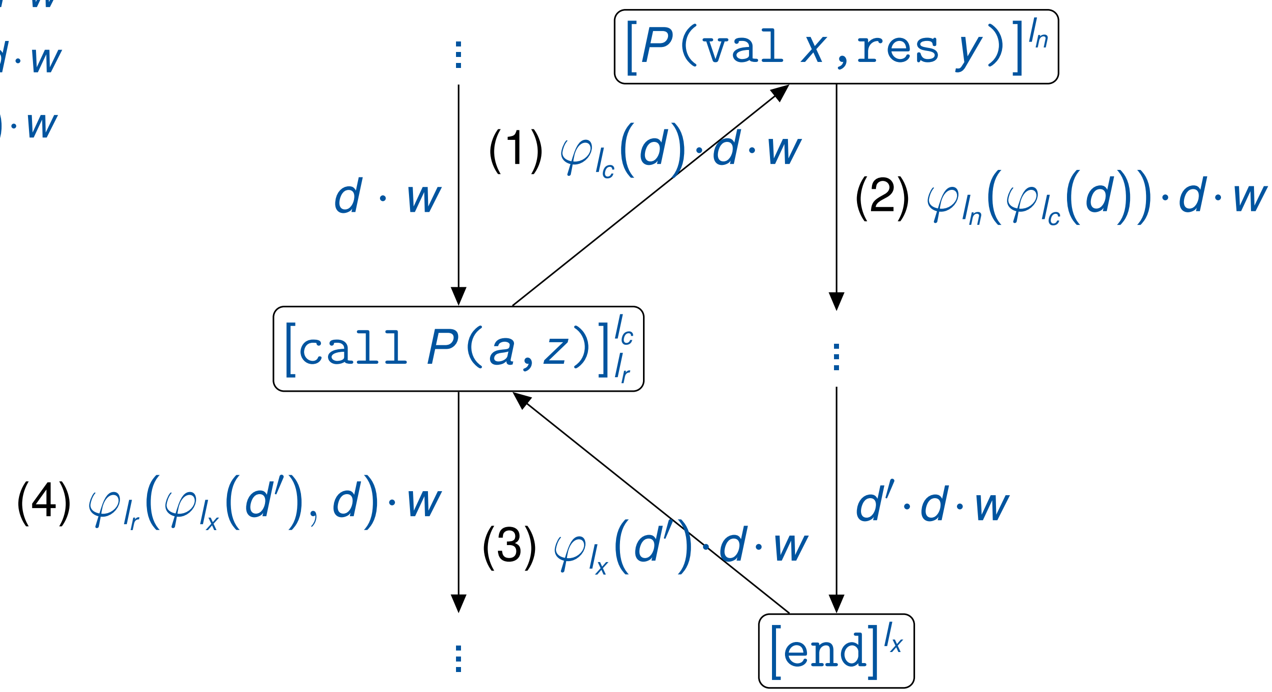

https://moves.rwth-aachen.de/teaching/ws-1617/spa/

# Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

## The Interprocedural Extension

Flow of information:

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. $\hat{\varphi}_{l_r}(d' \cdot d \cdot w) = \varphi_{l_r}(d', d) \cdot w$



### Types of Equations

For an interprocedural dataflow system $\hat{S} := (Lab, E, F, (\hat{D}, \hat{\sqsubseteq}), \hat{\iota}, \hat{\varphi})$, the intraprocedural equation system (cf. Definition 4.9)

$$\mathrm{AI}_l = \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\varphi_{l'}(\mathrm{AI}_{l'}) \mid (l', l) \in F\} & \text{otherwise} \end{cases}$$

is extended to a system with three kinds of equations (for every $l \in Lab$):

- for actual dataflow information: $\mathrm{AI}_l \in \hat{D}$
  - counterpart of intraprocedural AI
- for transfer functions of single nodes: $f_l : \hat{D} \to \hat{D}$
  - extension of intraprocedural transfer functions by special handling of procedure calls
- for transfer functions of complete procedures: $F_l : \hat{D} \to \hat{D}$
  - $F_l(w)$ yields information at $l$ if corresponding procedure is called with information $w$
  - thus complete procedure represented by $F_{l_x}$ ("procedure summary")

# Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

## Formal Definition of Equation System

Dataflow equations:

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

Node transfer functions (if $l$ not an exit label):

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

Procedure transfer functions (if $l$ occurs in some procedure):

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

As before: induces monotonic functional on lattice with ACC

$\Longrightarrow$ least fixpoint effectively computable

## Effectiveness of 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 $w \in D^*$,*

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

**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* $\hat{S} := \left(Lab, E, F, (\hat{D}, \hat{\sqsubseteq}), \hat{\iota}, \hat{\varphi}\right)$ *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 6.6)*

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

**Proof.**

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

## 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
  - information about calling states combined for all call sites
  - procedure body only analysed 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

## 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 (parallelisability)
  - 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]

## 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) | $Cmd$ | $c$ |

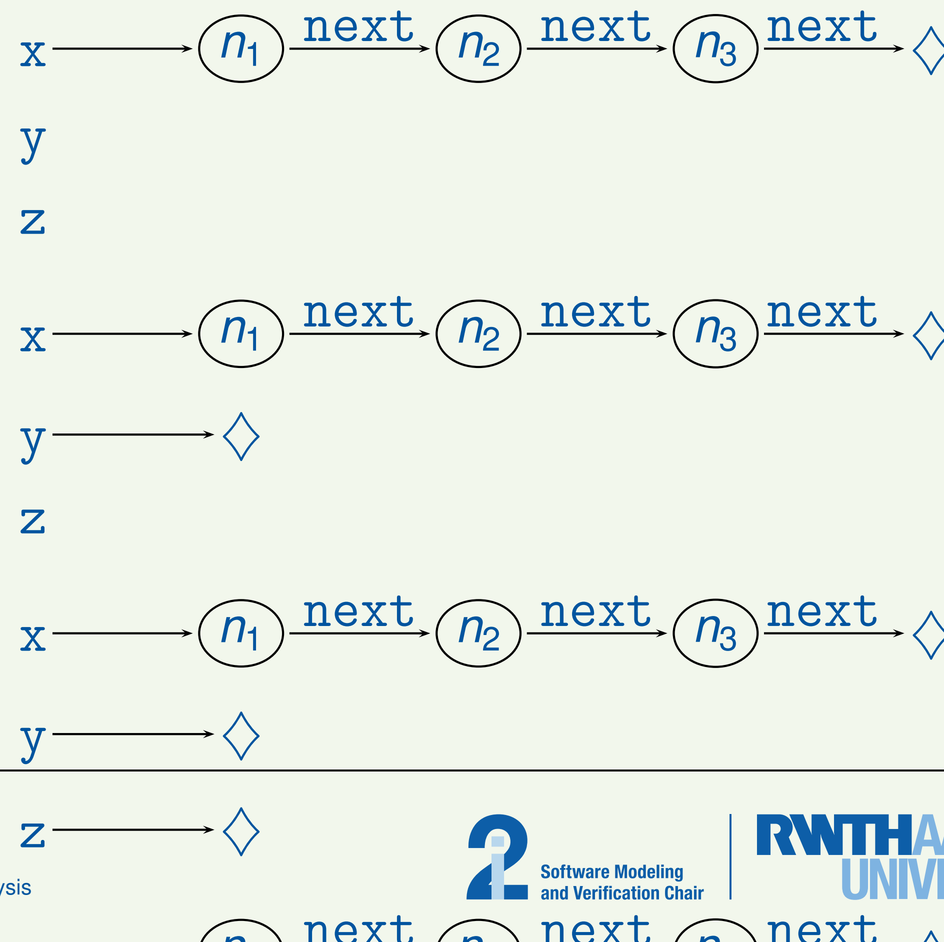
### Context-free grammar:

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

## An Example

**Example 20.3 (List reversal)**

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



## Heap Configurations

### Definition 20.4 (Heap configuration)

A (concrete) heap configuration is given by

$$H = (Nod, Sel, Var, \sigma, \longrightarrow)$$

where

- $Nod$ is a finite set of (concrete) nodes
- $Sel$ is a finite set of selector names
- $Var$ is a finite set of program variables
- $\sigma : Var \to \mathbb{Z} \cup Nod_\Diamond$ is a variable valuation (with $Nod_\Diamond := Nod \cup \{\Diamond\}$)
- $\longrightarrow: Nod \times Sel \to Nod_\Diamond$ is a (concrete) heap
  - notation: $n_1 \xrightarrow{sel} n_2$ for $((n_1, sel), n_2) \in \longrightarrow$

## Shape Graphs I

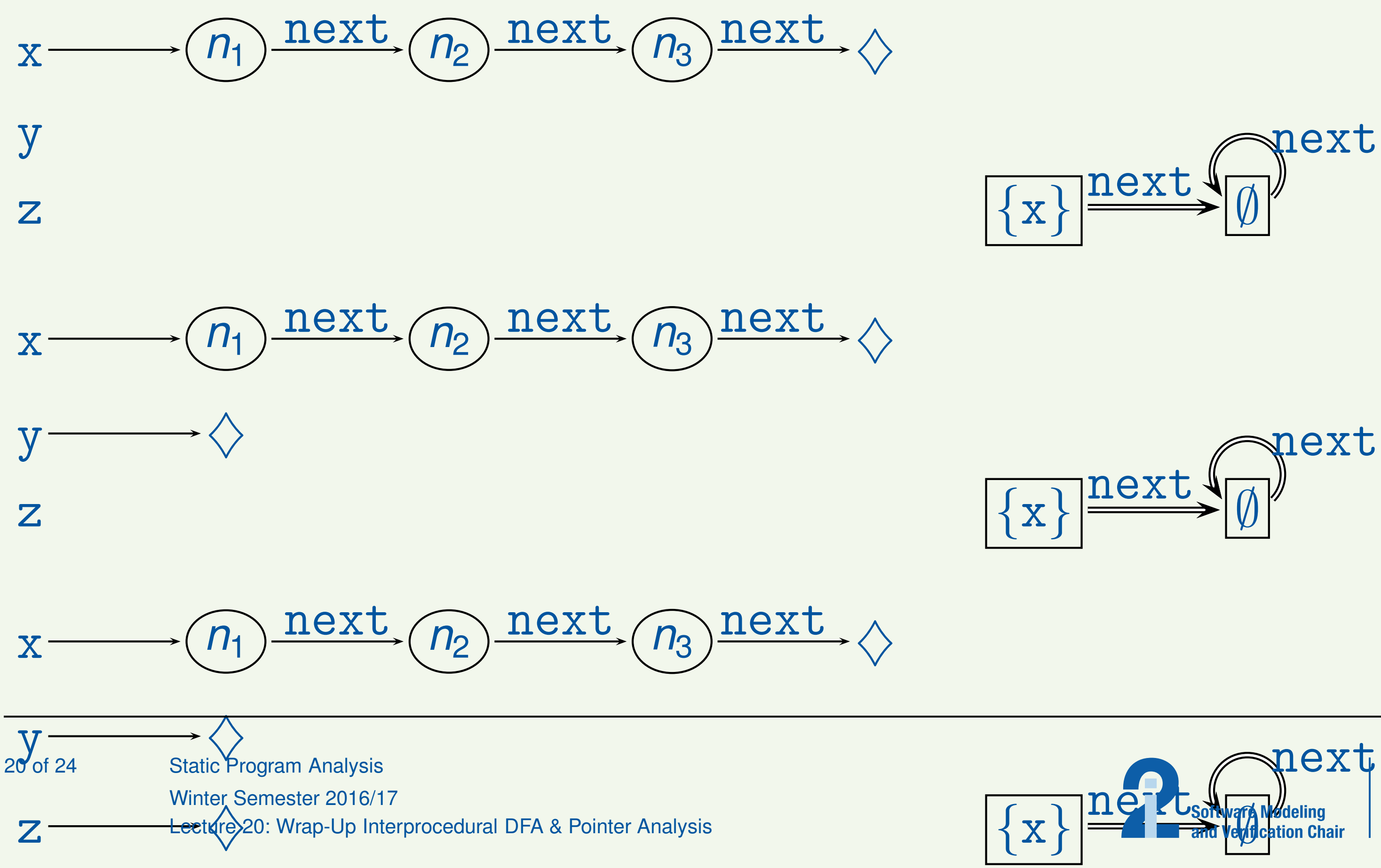
**Approach:** representation of (infinitely many) concrete heap states by (finitely many) abstract shape graphs

- 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 $X \stackrel{sel}{\Longrightarrow} Y$ (similarly for $X = \emptyset$ or $Y = \emptyset$)
- transfer functions transform (sets of) shape graphs

## Shape Graphs II

### Example 20.5 (List reversal; cf. Example 20.3)

**Concrete heap** **Shape graph**



## Shape Graphs III

### Definition 20.6 (Shape graph)

A shape graph
$$G = (Abs, \Longrightarrow)$$
consists of

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

with the following properties:

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

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

$SG$ denotes the set of all shape graphs.

## From Heap Configurations to Shape Graphs I

### Definition 20.7

Given a heap configuration $H = (Nod, Sel, Var, \sigma, \longrightarrow)$, the corresponding shape graph $G = (Abs, \Longrightarrow)$ is defined by

- $Abs := \{\sigma^{-1}(n) \mid n \in Nod\}$
  $\quad = \{\{x \in Var \mid \sigma(x) = n\} \mid n \in Nod\}$
- For all $X, Y \in Abs$ and $sel \in Sel$:

$$X \overset{sel}{\Longrightarrow} Y \quad \iff \quad \exists n_X, n_y \in Nod : \sigma^{-1}(n_X) = X, \sigma^{-1}(n_Y) = Y, n_X \overset{sel}{\longrightarrow} n_Y$$

**Remark:** yields Galois connection between sets of heap configurations and sets of shape graphs, both ordered by $\subseteq$

## From Heap Configurations to Shape Graphs II

**Remark:** the following example shows that determinacy can only be postulated if $X \neq \emptyset$:

- Concrete:

$$\begin{array}{l} y \longrightarrow \bullet \xleftarrow{sel} \bullet \\ z \longrightarrow \bullet \xleftarrow{sel} \bullet \end{array}$$

- Abstract:

$$\boxed{Y = \{y\}} \stackrel{sel}{\Longleftarrow} \boxed{X = \emptyset} \stackrel{sel}{\Longrightarrow} \boxed{Z = \{z\}}$$

## Shape Graphs and Concrete Heap Properties

### Example 20.8

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

- $\mathtt{x} \neq \mathtt{nil}$
  $\iff \exists X \in Abs : \mathtt{x} \in X$
- $\mathtt{x} = \mathtt{y} \neq \mathtt{nil}$ (aliasing)
  $\iff \exists Z \in Abs : \mathtt{x}, \mathtt{y} \in Z$
- $\mathtt{x.sel1} = \mathtt{y.sel2} \neq \mathtt{nil}$ (sharing)
  $\Longrightarrow \exists X, Y, Z \in Abs : \mathtt{x} \in X, \mathtt{y} \in Y, X \stackrel{\mathtt{sel1}}{\Longrightarrow} Z \stackrel{\mathtt{sel2}}{\Longleftarrow} Y$
  ("$\Longleftarrow$" only valid if $Z \neq \emptyset$)