Static Program Analysis

Lecture 21: Shape Analysis & Final Remarks

Winter Semester 2016/17

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

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

Outline of Lecture 21

Recap: Pointer Analysis

Shape Analysis

Further Topic in Program Analysis

Final Remarks

Recap: Pointer Analysis

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]

Recap: Pointer Analysis

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:

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

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

Definition (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: $X \overset{sel}{\Longrightarrow} Y$ 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 $X \overset{sel}{\Longrightarrow} Y$ and $X \overset{sel}{\Longrightarrow} Z \implies Y = Z$

(target location is unique if source node is unique)

SG denotes the set of all shape graphs.

From Heap Configurations to Shape Graphs

Definition

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 $sel \in Sel$:

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

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

Shape Graphs and Concrete Heap Properties

Example

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 $Z \neq \emptyset$)

Outline of Lecture 21

Recap: Pointer Analysis

Shape Analysis

Further Topic in Program Analysis

Final Remarks

The Goal

Shape Analysis

The goal of Shape Analysis is to determine, for each program point, a set of shape graphs that together represent all concrete heap configurations which can occur during program execution at that point.

The Goal

Shape Analysis

The goal of Shape Analysis is to determine, for each program point, a set of shape graphs that together represent all concrete heap configurations which can occur during program execution at that point.

- Forward analysis

The Goal

Shape Analysis

The goal of Shape Analysis is to determine, for each program point, a set of shape graphs that together represent all concrete heap configurations which can occur during program execution at that point.

- Forward analysis
- Domain: $(D, \sqsubseteq) := (2^{SG}, \subseteq)$ $\quad$ (Var, Sel finite $\Longrightarrow$ SG finite $\Longrightarrow$ 2^{SG} finite $\Longrightarrow$ ACC)

The Goal

Shape Analysis

The goal of Shape Analysis is to determine, for each program point, a set of shape graphs that together represent all concrete heap configurations which can occur during program execution at that point.

- Forward analysis
- Domain: $(D, \sqsubseteq) := (2^{SG}, \subseteq)$ $\quad$ (*Var*, *Sel* finite $\implies$ *SG* finite $\implies$ 2^{SG} finite $\implies$ ACC)
- Extremal value: $\iota :=$ {shape graphs for possible initial values of *Var*}

The Goal

Shape Analysis

The goal of Shape Analysis is to determine, for each program point, a set of shape graphs that together represent all concrete heap configurations which can occur during program execution at that point.

- Forward analysis
- Domain: $(D, \sqsubseteq) := (2^{SG}, \subseteq)$ $\quad$ (Var, Sel finite $\implies$ SG finite $\implies$ 2^{SG} finite $\implies$ ACC)
- Extremal value: $\iota :=$ {shape graphs for possible initial values of Var}

Example 21.1 (List reversal; cf. Example 20.5)

- Variables: $Var = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$
- Assumption: $\mathtt{x}$ points to any (finite, non-cyclic) list, $\mathtt{y} = \mathtt{z} = \mathtt{nil}$

$$\Rightarrow \iota = \left\{ \underbrace{(\emptyset, \emptyset)}_{\text{empty}}, \underbrace{\boxed{\{\mathtt{x}\}}}_{\text{1 elem.}}, \underbrace{\boxed{\{\mathtt{x}\}} \overset{\mathtt{next}}{\Longrightarrow} \boxed{\emptyset}}_{\text{2 elem.}}, \underbrace{\boxed{\{\mathtt{x}\}} \overset{\mathtt{next}}{\Longrightarrow} \overset{\overset{\mathtt{next}}{\curvearrowright}}{\boxed{\emptyset}}}_{\geq \text{3 elem.}} \right\}$$

Software Modeling and Verification Chair

The Transfer Functions

Transform each single shape graph into a set of shape graphs: for each $l \in Lab$,

$$\varphi_l : 2^{SG} \to 2^{SG} : \{G_1, \ldots, G_n\}) \mapsto \bigcup_{i=1}^{n} \varphi_l(G_i)$$

Definition 21.2 (Transfer functions for shape analysis)

$\varphi_l(G) \subseteq SG$ is determined by B^l (where $G = (Abs, \Longrightarrow)$):

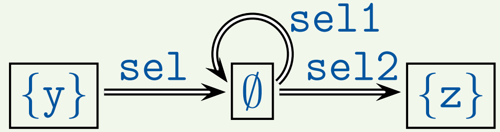
- $[\mathtt{skip}]^l$: $\varphi_l(G) := \{G\}$
- $[b]^l$: $\varphi_l(G) := \{G\}$
- $[p := a]^l$: case-by-case analysis w.r.t. p and a
  - [Nielson/Nielson/Hankin 2005, Sct. 2.6.3]: 12 cases on 11 p.
  - may involve (high degree of) non-determinism
  - see example on following slide
- $[\mathtt{malloc}\ x]^l$: $\varphi_l(G) := \{(Abs' \cup \{\{x\}\}, \Longrightarrow')\}$ with
  - $Abs' := \{X \setminus \{x\} \mid X \in Abs\}$
  - $\forall X, Y \in Abs, sel \in Sel$:

    $X \setminus \{x\} \overset{sel}{\Longrightarrow}{}' Y \setminus \{x\}$ iff $X \overset{sel}{\Longrightarrow} Y$
- $[\mathtt{malloc}\ x.sel]^l$: equivalent to $[\mathtt{malloc}\ t]^{l_1}; [x.sel := t]^{l_2}; [t := \mathtt{nil}]^{l_3}$ (with fresh $t \in Var$ and $l_1, l_2, l_3 \in Lab$)
- Fixpoint solution yields $SG_l \subseteq SG$ for each $l \in Lab$

Shape Analysis

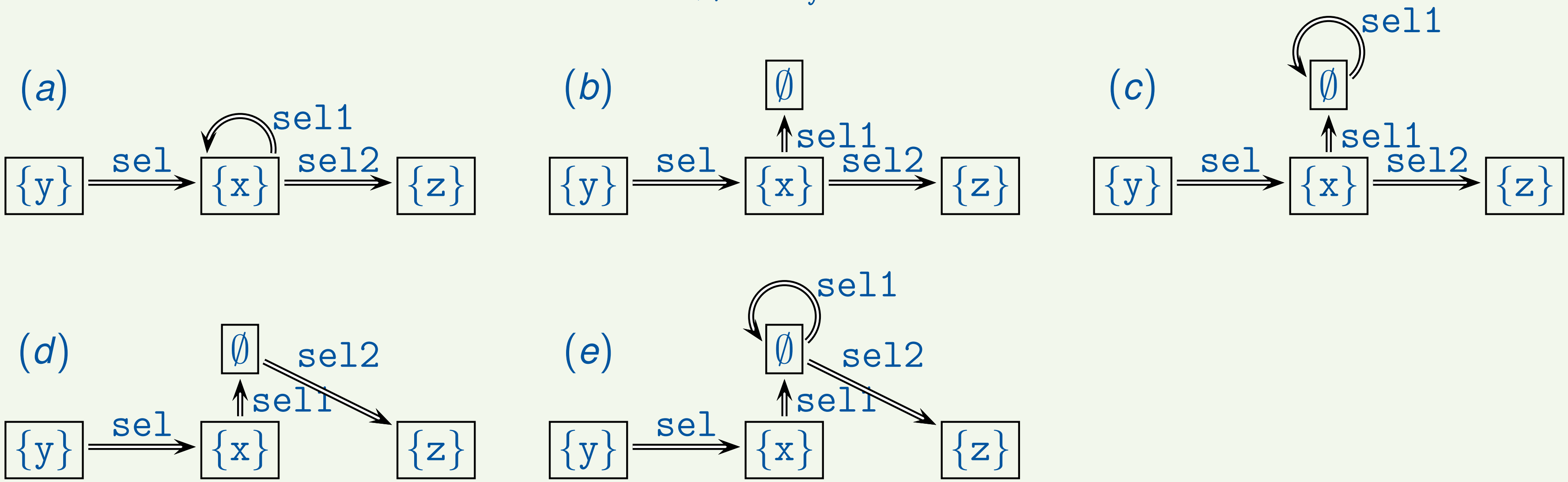
An Example

Example 21.3 (Transfer function for pointer assignment)



(justification: on the board)

$\downarrow \varphi_{\texttt{x := y.sel}}$



Soundness of Abstraction

Theorem 21.4 (Safety of approximation)

Let H be a heap configuration with corresponding shape graph G (according to Definition 20.7), and let $l \in Lab$. If B^l maps H to heap configuration H', then there exists a shape graph $G' \in \varphi_l(G)$ that corresponds to H'.

Proof.

omitted □

Application to List Reversal

Example 21.5 (List reversal; cf. Example 20.5)

Shape analysis of list reversal program yields final result

$$\left\{ \underbrace{(\emptyset, \emptyset)}_{\text{empty}} \;,\; \underbrace{\boxed{\{y\}}}_{\text{1 elem.}} \;,\; \underbrace{\boxed{\{y\}} \overset{\texttt{next}}{\Longrightarrow} \boxed{\emptyset}}_{\text{2 elem.}} \;,\; \underbrace{\boxed{\{y\}} \overset{\texttt{next}}{\Longrightarrow} \overset{\overset{\texttt{next}}{\curvearrowright}}{\boxed{\emptyset}}}_{\geq \text{ 3 elem.}} \right\}$$

Application to List Reversal

Example 21.5 (List reversal; cf. Example 20.5)

Shape analysis of list reversal program yields final result

$$\left\{ \underbrace{(\emptyset, \emptyset)}_{\text{empty}} \ , \ \underbrace{\boxed{\{y\}}}_{\text{1 elem.}} \ , \ \underbrace{\boxed{\{y\}} \xRightarrow{\texttt{next}} \boxed{\emptyset}}_{\text{2 elem.}} \ , \ \underbrace{\boxed{\{y\}} \xRightarrow{\texttt{next}} \overset{\texttt{next}}{\overset{\curvearrowright}{\boxed{\emptyset}}}}_{\geq 3 \text{ elem.}} \right\}$$

Interpretation:

+ Result again a finite list
– but potentially cyclic (may be a “lasso”, but not a ring)
– also “reversal” property not guaranteed
  – result could be in wrong order or have more/less entries

Outline of Lecture 21

Recap: Pointer Analysis

Shape Analysis

Further Topic in Program Analysis

Final Remarks

Dedicated Algorithms for Pointer Analysis

- `nil` Pointer Analysis: checks whether dereferencing operations possibly involve `nil` pointers
  - with shape analysis: $x = \texttt{nil}$ possible for $x \in Var$ at $l \in Lab$ if there exists $G = (Abs, \Longrightarrow) \in SG_l$ such that $x \notin \bigcup_{X \in Abs} X$
- Points-To Analysis: yields function pt that for each $x \in Var$ returns set $pt(x) \subseteq Nod$ of possible pointer targets
  - x and y may be aliases if $pt(x) \cap pt(y) \neq \emptyset$
  - with shape analysis: there exists $G = (Abs, \Longrightarrow) \in SG_l$ and $Z \in Abs$ such that $x, y \in Z$
- Usually faster and sometimes more precise than shape analysis, but less general (only "shallow" properties)
- Fastest algorithms are flow-insensitive (points-to edges only added but never removed)

Graph Grammar Approaches to Pointer Analysis

- E.g., J. Heinen, C. Jansen, J.-P. Katoen, T. Noll: *Verifying Pointer Programs using Graph Grammars*. Science of Computer Programming 97, 157–162, 2015
- Idea: specify data structures by graph production rules
- Concretisation by forward application
- Abstraction by backward application
- All pointer operations remain concrete

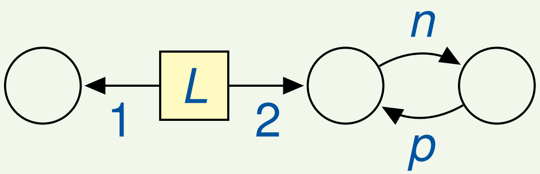
$\Rightarrow$ Avoids involved definition of transfer functions

Graph Grammar Approaches to Pointer Analysis

- E.g., J. Heinen, C. Jansen, J.-P. Katoen, T. Noll: *Verifying Pointer Programs using Graph Grammars*. Science of Computer Programming 97, 157–162, 2015
- Idea: specify data structures by graph production rules
- Concretisation by forward application
- Abstraction by backward application
- All pointer operations remain concrete

$\Rightarrow$ Avoids involved definition of transfer functions

Example 21.6 (Doubly-linked lists)



Further Topic in Program Analysis

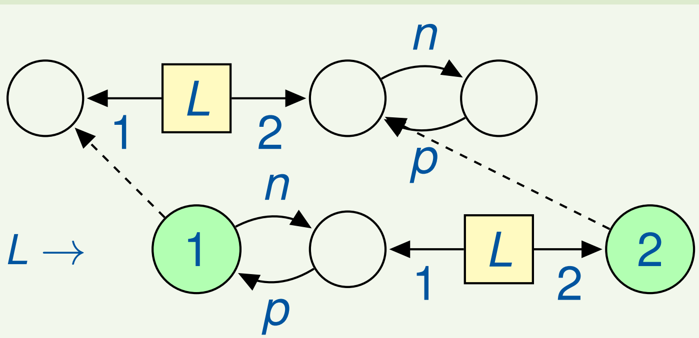
Graph Grammar Approaches to Pointer Analysis

- E.g., J. Heinen, C. Jansen, J.-P. Katoen, T. Noll: *Verifying Pointer Programs using Graph Grammars*. Science of Computer Programming 97, 157–162, 2015
- Idea: specify data structures by graph production rules
- Concretisation by forward application
- Abstraction by backward application
- All pointer operations remain concrete

$\Rightarrow$ Avoids involved definition of transfer functions

Example 21.6 (Doubly-linked lists)

$L \rightarrow$

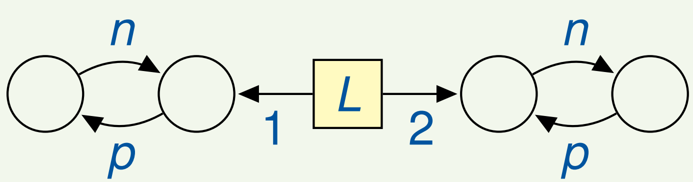


Graph Grammar Approaches to Pointer Analysis

- E.g., J. Heinen, C. Jansen, J.-P. Katoen, T. Noll: *Verifying Pointer Programs using Graph Grammars*. Science of Computer Programming 97, 157–162, 2015
- Idea: specify data structures by graph production rules
- Concretisation by forward application
- Abstraction by backward application
- All pointer operations remain concrete

⇒ Avoids involved definition of transfer functions

Example 21.6 (Doubly-linked lists)

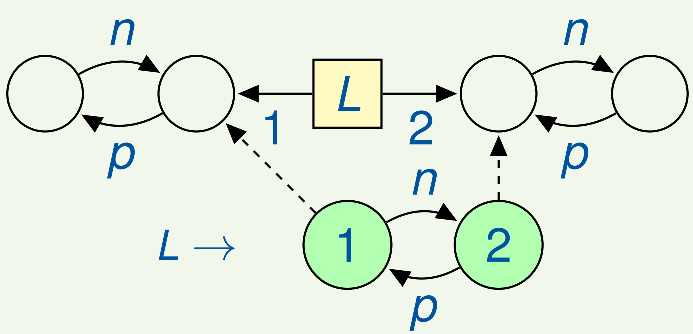


Graph Grammar Approaches to Pointer Analysis

- E.g., J. Heinen, C. Jansen, J.-P. Katoen, T. Noll: *Verifying Pointer Programs using Graph Grammars*. Science of Computer Programming 97, 157–162, 2015
- Idea: specify data structures by graph production rules
- Concretisation by forward application
- Abstraction by backward application
- All pointer operations remain concrete

$\Rightarrow$ Avoids involved definition of transfer functions

Example 21.6 (Doubly-linked lists)



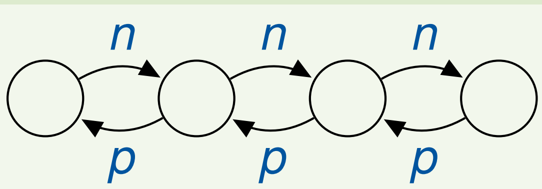
Further Topic in Program Analysis

Graph Grammar Approaches to Pointer Analysis

- E.g., J. Heinen, C. Jansen, J.-P. Katoen, T. Noll: *Verifying Pointer Programs using Graph Grammars*. Science of Computer Programming 97, 157–162, 2015
- Idea: specify data structures by graph production rules
- Concretisation by forward application
- Abstraction by backward application
- All pointer operations remain concrete

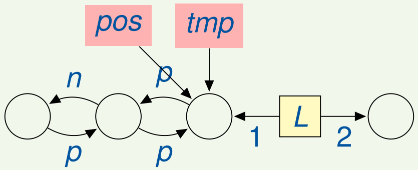
⇒ Avoids involved definition of transfer functions

Example 21.6 (Doubly-linked lists)

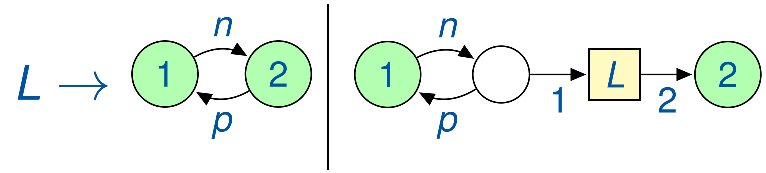


Abstract Execution Using Graph Grammars

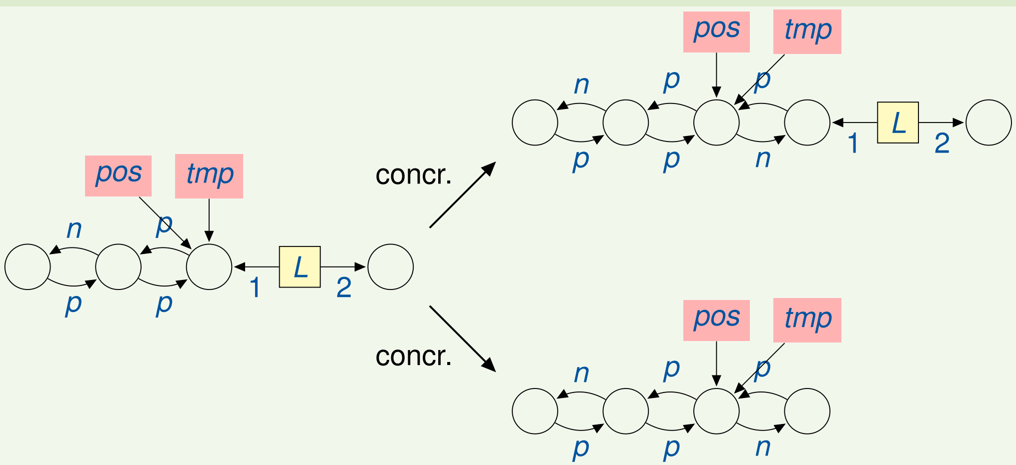
Example 21.7 (`tmp := pos.next;`)



Abstract Execution Using Graph Grammars

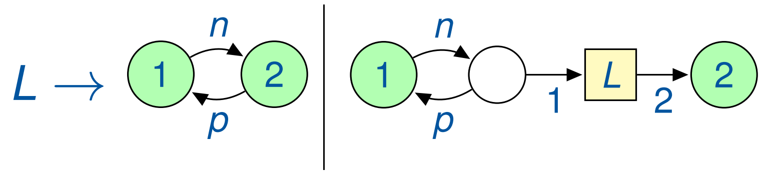


Example 21.7 (`tmp := pos.next;`)

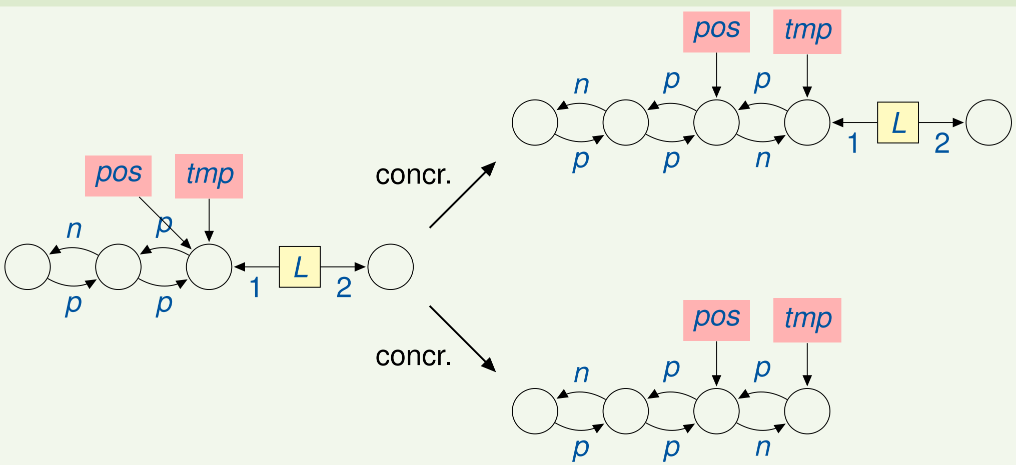


Further Topic in Program Analysis

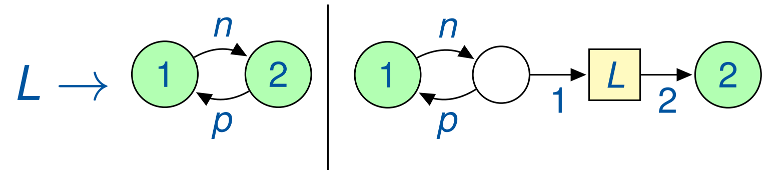
Abstract Execution Using Graph Grammars



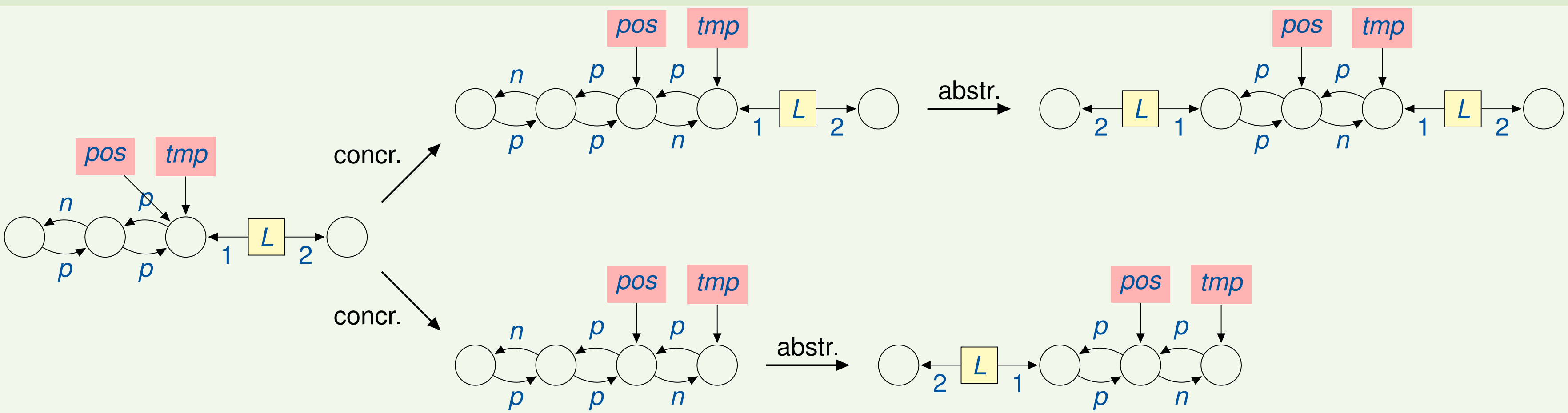
Example 21.7 (`tmp := pos.next;`)



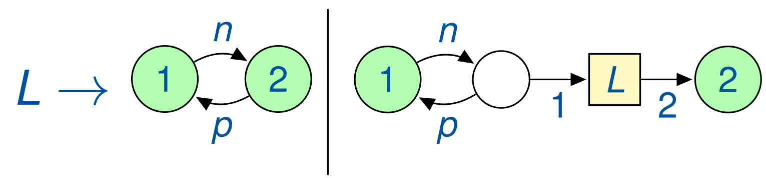
Abstract Execution Using Graph Grammars



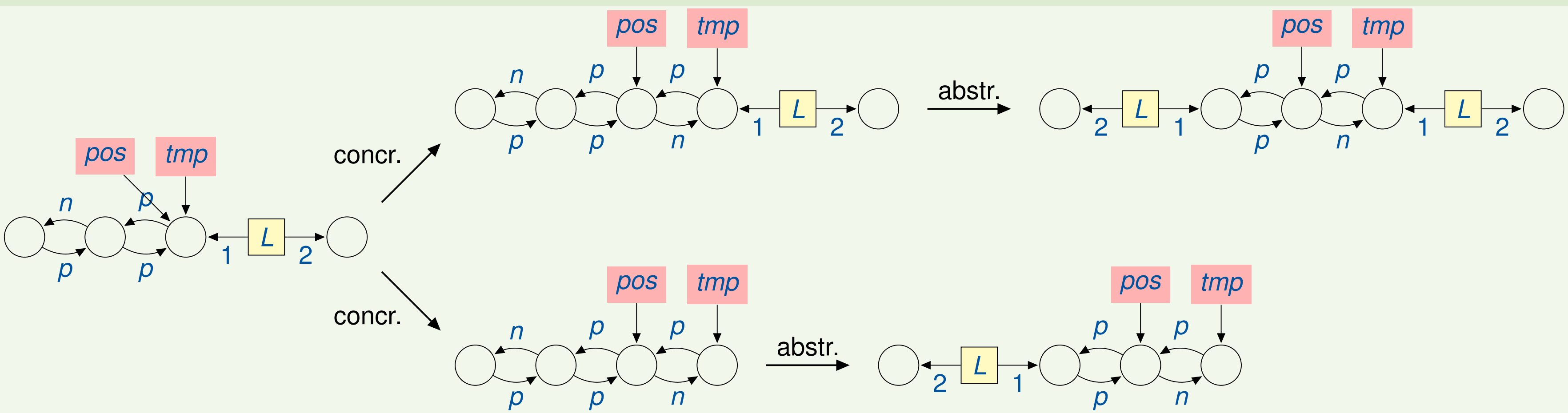
Example 21.7 (`tmp := pos.next;`)



Abstract Execution Using Graph Grammars



Example 21.7 (`tmp := pos.next;`)



Principle

Concretise whenever necessary; abstract whenever possible.

Correctness of Dataflow Analyses

- **So far:** semantics and dataflow analysis of programs considered independently (formal soundness proofs only for abstract interpretation; cf. Lecture 12/13)

Further Topic in Program Analysis

Correctness of Dataflow Analyses

- **So far:** semantics and dataflow analysis of programs considered independently (formal soundness proofs only for abstract interpretation; cf. Lecture 12/13)
- Of course both are (and should be) related!

Correctness of Dataflow Analyses

- **So far:** semantics and dataflow analysis of programs considered independently (formal soundness proofs only for abstract interpretation; cf. Lecture 12/13)
- Of course both are (and should be) related!
- To this aim: compare results of concrete semantics (Definition 11.9) with outcome of analysis

Correctness of Dataflow Analyses

- **So far:** semantics and dataflow analysis of programs considered independently (formal soundness proofs only for abstract interpretation; cf. Lecture 12/13)
- Of course both are (and should be) related!
- To this aim: compare results of concrete semantics (Definition 11.9) with outcome of analysis
- See [Nielson/Nielson/Hankin 2005, Sct. 2.2] for details

Example 21.8 (Correctness of Constant Propagation)

Let $c \in Cmd$, $l \in Lab_c$, $x \in Var$, and $z \in \mathbb{Z}$ such that $\mathrm{CP}_l(x) = z$.
Then for all $\sigma_0, \sigma \in \Sigma$ such that $\langle \mathrm{init}(c), \sigma_0 \rangle \to^* \langle l, \sigma \rangle$, $\sigma(x) = z$.

Outline of Lecture 21

Recap: Pointer Analysis

Shape Analysis

Further Topic in Program Analysis

Final Remarks

Final Remarks

Written Exam

- Dates:
  - Tue 21 Feb, 15:00–17:00, AH 2/3
  - Thu 23 Mar, 10:00–12:00, AH 2
- Q&A session on Wed 08 Feb (12:00, AH 3)
  - please submit questions beforehand to Christina Jansen or Christoph Matheja

Forthcoming Course in SS 2017

Compiler Construction [Noll; V3 Ü2]

1. Lexical analysis of programs (Scanner)
2. Syntactic analysis of programs (Parser)
3. Semantic analysis of programs
4. Code generation
5. Tools for compiler construction