Static Program Analysis

Lecture 21: Shape Analysis & Final Remarks

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

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]

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:

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

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

Shape Graphs II

Definition (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: $y \longrightarrow \bullet \xleftarrow{sel} \bullet$
$z \longrightarrow \bullet \xleftarrow{sel} \bullet$

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

Shape Graphs and Concrete Heap Properties

Example

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

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

Outline

The Goal

Shape Analysis

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

- Forward analysis
- Domain: $(D, \sqsubseteq) := (2^{SG}, \subseteq)$
  - 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 represent all heap structures which can occur during program execution at that point.

- Forward analysis
- Domain: $(D, \sqsubseteq) := (2^{SG}, \subseteq)$
  - 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 represent all heap structures which can occur during program execution at that point.

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

Example 21.1 (List reversal; cf. Example 20.4)

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

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

The Transfer Functions

Transfer functions: $\varphi_l : 2^{SG} \to 2^{SG}$ (monotonic)

- Transform each single shape graph into a set of shape graphs:
  $\varphi_l(\{G_1, \ldots, G_n\}) = \bigcup_{i=1}^{n} \varphi_l(G_i)$

The Transfer Functions

Transfer functions: $\varphi_l : 2^{SG} \to 2^{SG}$ (monotonic)

- Transform each single shape graph into a set of shape graphs:
  $\varphi_l(\{G_1, \ldots, G_n\}) = \bigcup_{i=1}^{n} \varphi_l(G_i)$
- $\varphi_l(G)$ determined by B^l (where $G = (S, H)$):
  - $[\texttt{skip}]^l$: $\varphi_l(G) := \{G\}$
  - $[b]^l$: $\varphi_l(G) := \{G\}$
  - $[p \texttt{ := } a]^l$: case-by-case analysis w.r.t. p and a
    - [Nielson/Nielson/Hankin 2005, Sct. 2.6.3]: 12 cases
    - may involve (high degree of) non-determinism
    - see example on following slide
  - $[\texttt{malloc } x]^l$: $\varphi_l(G) := \{(S' \cup \{\{x\}\}, H')\}$ where
    - $S' := \{X \setminus \{x\} \mid X \in S\}$
    - $H' := H \cap S' \times Sel \times S'$
  - $[\texttt{malloc } x.sel]^l$: equivalent to $[\texttt{malloc } t]^{l_1}; [x.sel \texttt{ := } t]^{l_2}; [t \texttt{ := nil}]^{l_3};$
    (with fresh $t \in Var$ and $l_1, l_2, l_3 \in Lab$)

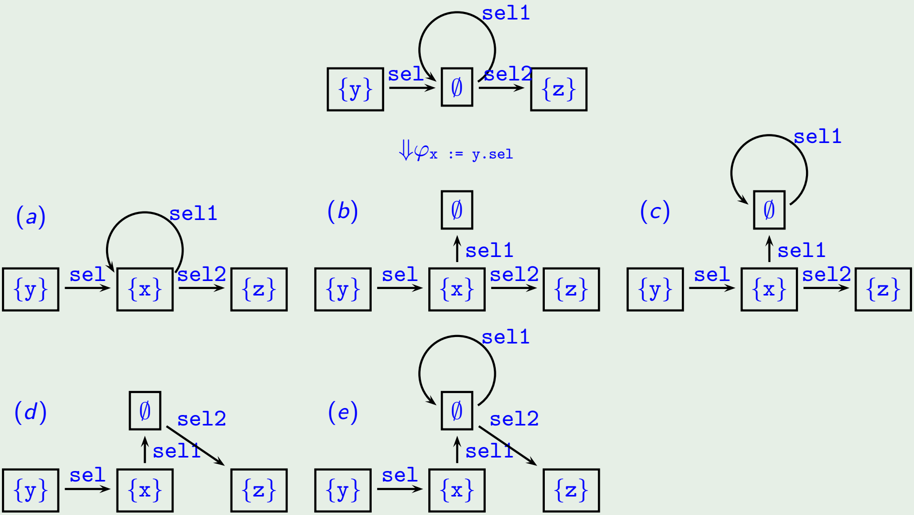
The Transfer Functions

Transfer functions: $\varphi_l : 2^{SG} \to 2^{SG}$ (monotonic)

- Transform each single shape graph into a set of shape graphs: $\varphi_l(\{G_1, \ldots, G_n\}) = \bigcup_{i=1}^{n} \varphi_l(G_i)$
- $\varphi_l(G)$ determined by B^l (where $G = (S, H)$):
  - $[\texttt{skip}]^l$: $\varphi_l(G) := \{G\}$
  - $[b]^l$: $\varphi_l(G) := \{G\}$
  - $[p \texttt{ := } a]^l$: case-by-case analysis w.r.t. p and a
    - [Nielson/Nielson/Hankin 2005, Sct. 2.6.3]: 12 cases
    - may involve (high degree of) non-determinism
    - see example on following slide
  - $[\texttt{malloc}\ x]^l$: $\varphi_l(G) := \{(S' \cup \{\{x\}\}, H')\}$ where
    - $S' := \{X \setminus \{x\} \mid X \in S\}$
    - $H' := H \cap S' \times Sel \times S'$
  - $[\texttt{malloc}\ x.sel]^l$: equivalent to $[\texttt{malloc}\ t]^{l_1}; [x.sel \texttt{ := } t]^{l_2}; [t \texttt{ := nil}]^{l_3};$ (with fresh $t \in Var$ and $l_1, l_2, l_3 \in Lab$)
- Crucial for soundness: safety of approximation
  If shape graph G approximates heap h and $h \xrightarrow{B^l} h'$,
  then there exists $G' \in \varphi_l(G)$ such that G' approximates h'

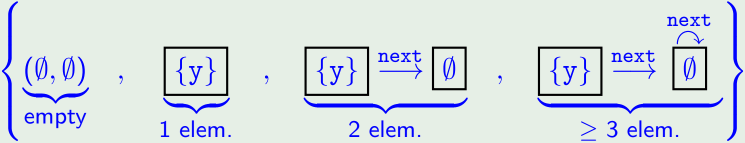
An Example

Example 21.2



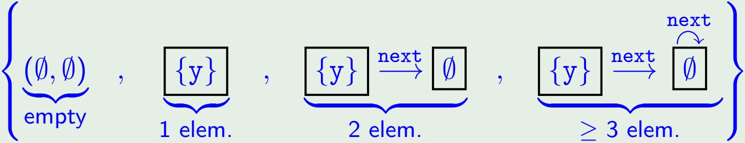
Example 21.3 (List reversal; cf. Example 20.4)

Shape analysis of list reversal program yields final result



Example 21.3 (List reversal; cf. Example 20.4)

Shape analysis of list reversal program yields final result



Interpretation:

\+ Result again a finite list

– but potentially cyclic (may be a "lasso", but not a ring)

– also "reversal" property not guaranteed

Outline

Dedicated Algorithms for Pointer Analysis

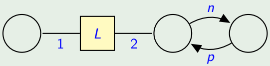
- `nil` Pointer Analysis: checks whether dereferencing operations possibly involve `nil` pointers
  - with shape analysis: possible for $x \in Var$ if there exists (reachable) $G = (S, H)$ such that $x \notin \bigcup_{X \in S} X$
- Points-To Analysis: yields function pt that for each $x \in Var$ returns set $pt(x)$ of possible pointer targets
  - x and y may be aliases if $pt(x) \cap pt(y) \neq \emptyset$
  - with shape analysis: there exists (reachable) $G = (S, H)$ and $Z \in S$ 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
- concretization by forward application
- abstraction by backward application
- all pointer operations remain concrete

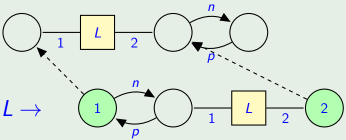
  $\Longrightarrow$ avoids complicated definition of transfer functions

Example 21.4 (Doubly-linked lists)



- 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
- concretization by forward application
- abstraction by backward application
- all pointer operations remain concrete
  $\Longrightarrow$ avoids complicated definition of transfer functions

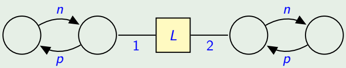
Example 21.4 (Doubly-linked lists)



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
- concretization by forward application
- abstraction by backward application
- all pointer operations remain concrete
  $\Longrightarrow$ avoids complicated definition of transfer functions

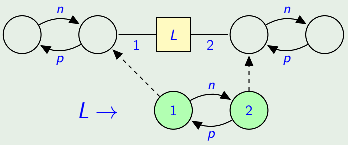
Example 21.4 (Doubly-linked lists)



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
- concretization by forward application
- abstraction by backward application
- all pointer operations remain concrete
  $\Longrightarrow$ avoids complicated definition of transfer functions

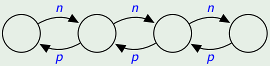
Example 21.4 (Doubly-linked lists)



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
- concretization by forward application
- abstraction by backward application
- all pointer operations remain concrete
  $\Longrightarrow$ avoids complicated definition of transfer functions

Example 21.4 (Doubly-linked lists)



Correctness of Dataflow Analyses

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

Correctness of Dataflow Analyses

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

  Let $c \in Cmd$ with $l_0 = \text{init}(c)$, and let $l \in Lab_c$, $x \in Var$, and $z \in \mathbb{Z}$ such that $\text{CP}_l(x) = z$. Then for all $\sigma_0, \sigma \in \Sigma$ such that $\langle l_0, \sigma_0 \rangle \to^* \langle l, \sigma \rangle$, $\sigma(x) = z$.

Correctness of Dataflow Analyses

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

  Let $c \in Cmd$ with $l_0 = \text{init}(c)$, and let $l \in Lab_c$, $x \in Var$, and $z \in \mathbb{Z}$ such that $\text{CP}_l(x) = z$. Then for all $\sigma_0, \sigma \in \Sigma$ such that $\langle l_0, \sigma_0 \rangle \to^* \langle l, \sigma \rangle$, $\sigma(x) = z$.
- see [Nielson/Nielson/Hankin 2005, Sct. 2.2]

Outline

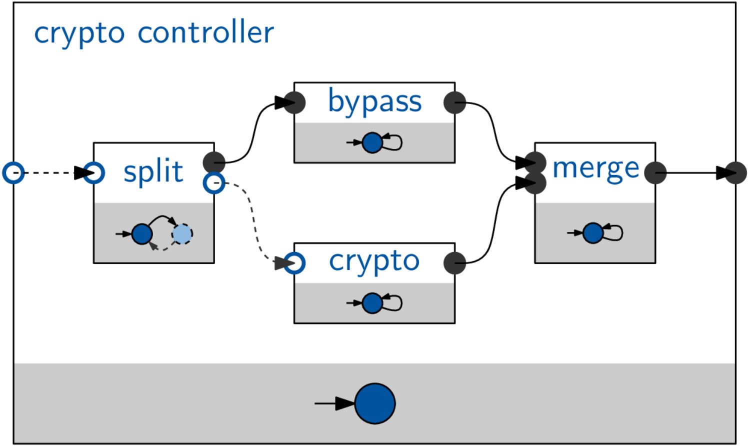
Oral Exams

- Schedule online
  - 12 + 24 March, 8 April
  - see http://moves.rwth-aachen.de/teaching/ws-1415/spa/
- Q&A session on Tuesday, 24 February, 14:00–15:30, AH 6
  - please submit questions beforehand to dehnert@cs.rwth-aachen.de or benjamin.kaminski@cs.rwth-aachen.de
  - contact me in case of unresolved/later questions

Thesis: Analysing Information Flows Using Slicing

- Computer security: system architectures that disallow sensitive information to be "leaked" to unauthorised entities
- Critical: covert channels that expose information
- Requires analysis of information flows within and between architectural components
- Standard approaches (non-interference, slicing) ignore encryption
- Goal: analysis of cryptographically-masked information flows using slicing techniques

Crypto controller



Forthcoming Courses in SS 2015

Introduction to Model Checking [Katoen; V3 Ü2]

1. Labelled transition systems
2. Classification of properties: safety, liveness, fairness
3. Temporal logics LTL and CTL
4. Model checking algorithms
5. Abstraction using (bi-)simulation

Semantics and Verification of Software [Noll; V3 Ü2]

1. The imperative model language WHILE
2. Operational, denotational and axiomatic semantics of WHILE
3. Equivalence of the semantics
4. Applications: compiler correctness, ...
5. Extensions: procedures, non-determinism, concurrency