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/
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, Sect. 2.6]
Recap: Pointer Analysis

Extending the Syntax

Syntactic categories:

<table>
<thead>
<tr>
<th>Category</th>
<th>Domain</th>
<th>Meta variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic expressions</td>
<td>$AExp$</td>
<td>$a$</td>
</tr>
<tr>
<td>Boolean expressions</td>
<td>$BExp$</td>
<td>$b$</td>
</tr>
<tr>
<td>Selector names</td>
<td>$Sel$</td>
<td>$sel$</td>
</tr>
<tr>
<td>Pointer expressions</td>
<td>$PExp$</td>
<td>$p$</td>
</tr>
<tr>
<td>Commands (statements)</td>
<td>$Cmd$</td>
<td>$c$</td>
</tr>
</tbody>
</table>

Context-free grammar:

\[
a ::= z \mid x \mid a_1 + a_2 \mid \ldots \mid p \mid \text{nil} \in AExp
\]
\[
b ::= t \mid a_1 = a_2 \mid b_1 \land b_2 \mid \ldots \mid \text{is-nil}(p) \in BExp
\]
\[
p ::= x \mid x . sel
\]
\[
c ::= [\text{skip}]' \mid [p := a]' \mid c_1 ; c_2 \mid \text{if } [b]' \text{ then } c_1 \text{ else } c_2 \text{ end} \mid \text{while } [b]' \text{ do } c \text{ end} \mid [\text{malloc } p]' \in Cmd
\]
Recap: Pointer Analysis

Shape Graphs I

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

- abstract nodes $X = \text{sets of variables}$
- 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 $X \xrightarrow{\text{sel}} Y$ (similarly for $X = \emptyset$ or $Y = \emptyset$)
- transfer functions transform (sets of) shape graphs
Recap: Pointer Analysis

Shape Graphs II

Definition (Shape graph)

A shape graph 

\[ G = (\text{Abs}, \rightarrow) \]

consists of

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

with the following properties:

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

Determinacy: \( X \neq \emptyset \) and \( X \xleftarrow{\text{sel}} Y \) and \( X \xleftarrow{\text{sel}} Z \Rightarrow Y = Z \)
(target location is unique if source node is unique)

\( \text{SG} \) denotes the set of all shape graphs.
Recap: Pointer Analysis

From Heap Configurations to Shape Graphs

Definition

Given a heap configuration \( H = (\text{Nod}, \text{Sel}, \text{Var}, \sigma, \rightarrow) \), the corresponding shape graph \( G = (\text{Abs}, \Rightarrow) \) is defined by

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

\[X \overset{\text{sel}}{\Rightarrow} Y \iff \exists n_X, n_Y \in \text{Nod} : \sigma^{-1}(n_X) = X, \sigma^{-1}(n_Y) = Y, n_X \overset{\text{sel}}{\rightarrow} n_Y\]

Remark: yields Galois connection between sets of heap configurations and sets of shape graphs, both ordered by \( \subseteq \)
Recap: Pointer Analysis

Shape Graphs and Concrete Heap Properties

Example

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

- $x \neq \text{nil}$
  \[ \iff \exists X \in Abs : x \in X \]

- $x = y \neq \text{nil}$ (aliasing)
  \[ \iff \exists Z \in Abs : x, y \in Z \]

- $x.sel1 = y.sel2 \neq \text{nil}$ (sharing)
  \[ \Rightarrow \exists X, Y, Z \in Abs : x \in X, y \in Y, X \Rightarrow Z \Leftrightarrow Y \]
  (“$\Leftrightarrow$” only valid if $Z \neq \emptyset$)
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**: \((D, \subseteq) := (2^{SG}, \subseteq)\)  
  \((Var, Sel)\) finite \(\implies SG\) finite \(\implies 2^{SG}\) finite \(\implies ACC\)
- **Extremal value**: \(\iota := \{\text{shape graphs for possible initial values of } Var\}\)

**Example 21.1 (List reversal; cf. Example 20.5)**

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

\[\implies \iota = \left\{ \begin{array}{l}
(\emptyset, \emptyset), \quad \text{empty} \\
\{x\}, \quad 1 \text{ elem.} \\
\{x\} \xrightarrow{\text{next}} \emptyset, \quad 2 \text{ elem.} \\
\{x\} \xrightarrow{\text{next}} \emptyset, \quad \geq 3 \text{ elem.}
\end{array} \right\} \]
Shape Analysis

The Transfer Functions

Transform each single shape graph into a set of shape graphs: for each \( l \in \text{Lab} \),

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

Definition 21.2 (Transfer functions for shape analysis)

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

- \([\text{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 \\
  - \text{[Nielson/Nielson/Hankin 2005, Sct. 2.6.3]: 12 cases on 11 p.} \\
  - \text{may involve (high degree of) non-determinism} \\
  - \text{see example on following slide}
- \([\text{malloc } x]^l:\varphi_l(G) := \{(Abs' \cup \{\{x\}\}, \Rightarrow')\} \text{ with} \\
  - \text{Abs'} := \{X \setminus \{x\} \mid X \in \text{Abs}\} \\
  - \forall X, Y \in \text{Abs}, sel \in \text{Sel} : \\
  X \setminus \{x\} \xrightarrow{\text{sel}} Y \setminus \{x\} \text{ iff } X \xrightarrow{\text{sel}} Y
- \([\text{malloc } x.\text{sel}]^l:\) equivalent to \( [\text{malloc } t]_{l1};[x.\text{sel} := t]_{l2};[t := \text{nil}]_{l3} \) (with fresh \( t \in \text{Var} \) and \( l_1, l_2, l_3 \in \text{Lab} \))
- Fixpoint solution yields \( SG_l \subseteq SG \) for each \( l \in \text{Lab} \)
Shape Analysis

An Example

Example 21.3 (Transfer function for pointer assignment)

\[
\begin{align*}
\{y\} & \xrightarrow{\text{sel}} \emptyset & \xrightarrow{\text{sel1, sel2}} \{z\} \\
\downarrow \varphi_x := y.\text{sel} & & (\text{justification: on the board})
\end{align*}
\]

(a) \[
\begin{align*}
\{y\} & \xrightarrow{\text{sel}} \{x\} & \xrightarrow{\text{sel1, sel2}} \{z\}
\end{align*}
\]

(b) \[
\begin{align*}
\{y\} & \xrightarrow{\text{sel}} \{x\} & \xrightarrow{\text{sel1, sel2}} \{z\}
\end{align*}
\]

(c) \[
\begin{align*}
\{y\} & \xrightarrow{\text{sel}} \{x\} & \xrightarrow{\text{sel1, sel2}} \{z\}
\end{align*}
\]

(d) \[
\begin{align*}
\{y\} & \xrightarrow{\text{sel}} \emptyset & \xrightarrow{\text{sel1, sel2}} \{z\}
\end{align*}
\]

(e) \[
\begin{align*}
\{y\} & \xrightarrow{\text{sel}} \emptyset & \xrightarrow{\text{sel1, sel2}} \{z\}
\end{align*}
\]
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 \( l \in \text{Lab} \). 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
Shape Analysis

Application to List Reversal

Example 21.5 (List reversal; cf. Example 20.5)

Shape analysis of list reversal program yields final result

\[
\begin{align*}
&\emptyset, \\
&\{y\}, \\
&\{y\} \xrightarrow{\text{next}} \emptyset, \\
&\{y\} \xrightarrow{\text{next}} \emptyset
\end{align*}
\]

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
Further Topic in Program Analysis

Dedicated Algorithms for Pointer Analysis

- **nil Pointer Analysis**: checks whether dereferencing operations possibly involve `nil` pointers
  - with shape analysis: $x = \text{nil}$ possible for $x \in \text{Var}$ at $I \in \text{Lab}$ if there exists $G = (\text{Abs}, \Rightarrow) \in SG_i$ such that $x \notin \bigcup_{X \in \text{Abs}} X$
- **Points-To Analysis**: yields function $pt$ that for each $x \in \text{Var}$ returns set $pt(x) \subseteq \text{Nod}$ of possible pointer targets
  - $x$ and $y$ may be aliases if $pt(x) \cap pt(y) \neq \emptyset$
  - with shape analysis: there exists $G = (\text{Abs}, \Rightarrow) \in SG_i$ and $Z \in \text{Abs}$ 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)
Further Topic in Program Analysis

Graph Grammar Approaches to Pointer Analysis

- 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)

\[ L \rightarrow n \rightarrow L \rightarrow n \rightarrow L \rightarrow n \]

\[ L \rightarrow L1 \rightarrow 1 \rightarrow p \rightarrow 2 \rightarrow 2 \rightarrow p \rightarrow 2 \rightarrow L \]
Further Topic in Program Analysis

Abstract Execution Using Graph Grammars

Example 21.7 ($\text{tmp := pos.next;}$)

Principle

Concretise whenever necessary; abstract whenever possible.
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!
- 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 $c \in \text{Cmd}$, $l \in \text{Lab}_c$, $x \in \text{Var}$, and $z \in \mathbb{Z}$ such that $\text{CP}_l(x) = z$.
Then for all $\sigma_0, \sigma \in \Sigma$ such that $\langle \text{init}(c), \sigma_0 \rangle \rightarrow^* \langle l, \sigma \rangle$, $\sigma(x) = z$. 
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
Final Remarks

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