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

Outline of Lecture 20

Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

Effectiveness and Correctness

Context-Sensitive Interprocedural Dataflow Analysis

Pointer Analysis

Introducing Pointers

Shape Graphs
Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

The Interprocedural Extension

Flow of information:
1. \( \hat{\varphi}_l(d \cdot w) = \varphi_l(d) \cdot d \cdot w \)
2. \( \hat{\varphi}_n(d' \cdot d \cdot w) = \varphi_n(d') \cdot d \cdot w \)
3. \( \hat{\varphi}_x(d' \cdot d \cdot w) = \varphi_x(d') \cdot d \cdot w \)
4. \( \hat{\varphi}_r(d' \cdot d \cdot w) = \varphi_r(d', d) \cdot w \)
Types of Equations

For an interprocedural dataflow system \( \hat{S} := (\text{Lab}, E, F, (\hat{D}, \hat{\alpha}), \hat{i}, \hat{\phi}) \), the intraprocedural equation system (cf. Definition 4.9)

\[
\text{Al}_l = \begin{cases} 
  \cup \{ \varphi_{l'}(\text{Al}_{l'}) \mid (l', l) \in F \} & \text{if } l \in E \\
  \text{otherwise} & \text{otherwise}
\end{cases}
\]

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

- for actual dataflow information: \( \text{Al}_l \in \hat{D} \)
  - counterpart of intraprocedural \( \text{Al} \)
- for transfer functions of single nodes: \( f_l : \hat{D} \rightarrow \hat{D} \)
  - extension of intraprocedural transfer functions by special handling of procedure calls
- for transfer functions of complete procedures: \( F_l : \hat{D} \rightarrow \hat{D} \)
  - \( F_l(w) \) yields information at \( l \) if corresponding procedure is called with information \( w \)
  - thus complete procedure represented by \( F_{lx} \) (“procedure summary”)

Recap: Interprocedural Dataflow Analysis – Fixpoint Solution
Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

Formal Definition of Equation System

Dataflow equations:

\[ A_I = \begin{cases} 
  l & \text{if } l \in E \\
  A_{I_c} & \text{if } l = l_r \text{ for some } (l_c, l_n, l_x, l_r) \in \text{iflow} \\
  \bigsqcup \{ f_{l'}(A_{I'}) \mid (l', l) \in F \} & \text{otherwise}
\end{cases} \]

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

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

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

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

As before: induces monotonic functional on lattice with ACC

\[ \Rightarrow \text{ least fixpoint effectively computable} \]
Effectiveness and Correctness

Outline of Lecture 20

Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

Effectiveness and Correctness

Context-Sensitive Interprocedural Dataflow Analysis

Pointer Analysis

Introducing Pointers

Shape Graphs
Effectiveness and Correctness

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 \( l \in \text{Lab}, d \in D, \) and \( w \in D^\ast, \)

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

Proof.


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

7 of 24 Static Program Analysis Winter Semester 2016/17 Lecture 20: Wrap-Up Interprocedural DFA & Pointer Analysis
Effectiveness and Correctness

Soundness and Completeness

The following results carry over from the intraprocedural case:

**Theorem 20.2**

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

1. (cf. Theorem 6.3)
   \[
   \text{mvp}(\hat{S}) \hat{\subseteq} \text{fix}(\Phi_{\hat{S}})
   \]

2. (cf. Theorem 6.6)
   \[
   \text{mvp}(\hat{S}) = \text{fix}(\Phi_{\hat{S}}) \text{ if all } \hat{\varphi}_l \text{ are distributive}
   \]

**Proof.**

Context-Sensitive Interprocedural Dataflow Analysis

Outline of Lecture 20

Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

Effectiveness and Correctness

Context-Sensitive Interprocedural Dataflow Analysis

Pointer Analysis

Introducing Pointers

Shape Graphs
Context-Sensitive Interprocedural Dataflow Analysis

Context-Sensitive Interprocedural DFA

- **Observation:** MVP and fixpoint solution maintain proper relationship between procedure calls and returns
Context-Sensitive Interprocedural Dataflow Analysis

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

⇒ “context-insensitive”
Context-Sensitive Interprocedural Dataflow Analysis

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
  ⇒ “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 of Lecture 20

Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

Effectiveness and Correctness

Context-Sensitive Interprocedural Dataflow Analysis

Pointer Analysis

Introducing Pointers

Shape Graphs
Pointer Analysis

Pointer Analysis

- **So far:** only static data structures (variables)
Pointer Analysis

Pointer Analysis

- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps
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
Next, we will discuss self-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
Pointer Analysis

The Shape Analysis Approach

• **Goal:** determine the possible shapes of a dynamically allocated data structure at given program point
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, ...)
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?
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]
Introducing Pointers

Outline of Lecture 20

Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

Effectiveness and Correctness

Context-Sensitive Interprocedural Dataflow Analysis

Pointer Analysis

Introducing Pointers

Shape Graphs
Introducing Pointers

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>
### Introducing Pointers

#### 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:**

\[
a ::= z \mid x \mid a_1 + a_2 \mid \ldots \mid p \mid \text{nil} \in AExp
\]

\[
b ::= t \mid a_1 = a_2 \mid b_1 \land b_2 \mid \ldots \mid \text{is-nil}(p) \in BExp
\]

\[
p ::= x \mid x.sel
\]

\[
c ::= [\text{skip}]' \mid [p := a]' \mid c_1; c_2 \mid \text{if } [b]' \text{ then } c_1 \text{ else } c_2 \text{ end} \mid \text{while } [b]' \text{ do } c \text{ end} \mid [\text{malloc } p]' \in Cmd
\]
Introducing Pointers

An Example

Example 20.3 (List reversal)

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

\[
\begin{align*}
[y := \text{nil}]^1; \\
\text{while } [\neg \text{is-nil}(x)]^2 \text{ do} \\
[z := y]^3; \\
[y := x]^4; \\
x := x.\text{next}^5; \\
[y.\text{next} := z]^6 \\
\text{end}; \\
z := \text{nil}^7;
\end{align*}
\]
Example 20.3 (List reversal)

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

\[
\begin{align*}
[y & := \text{nil}]^1; \\
\text{while } \neg \text{is-nil}(x)[^2] \text{ do} \\
[z & := y]^3; \\
[y & := x]^4; \\
[x & := x.\text{next}]^5; \\
[y.\text{next} & := z]^6 \\
\text{end;} \\
[z & := \text{nil}]^7;
\end{align*}
\]
Introducing Pointers

An Example

Example 20.3 (List reversal)

Program that reverses list pointed to by \textit{x} and leaves result in \textit{y}:

\begin{verbatim}
[y := nil];
while [\neg is-nil(x)] do
    [z := y];
    [y := x];
    [x := x.next];
    [y.next := z];
end;
[z := nil];
\end{verbatim}
Example 20.3 (List reversal)

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

$[y := \text{nil}]$;
while $[\neg \text{is-nil}(x)]$ do
$[z := y]$;
$[y := x]$;
$[x := x.\text{next}]$;
$[y.\text{next} := z]$;
end;
$[z := \text{nil}]$;

![Diagram of list reversal]
Introducing Pointers

An Example

Example 20.3 (List reversal)

Program that reverses list pointed to by \textit{x} and leaves result in \textit{y}:

\begin{verbatim}
[y := nil];
while [\neg is-nil(x)] do
  [z := y];
  [y := x];
  [x := x.next];
  [y.next := z];
end;
[z := nil];
\end{verbatim}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{example_diagram}
\caption{Diagram of list reversal program.}
\end{figure}
Introducing Pointers

An Example

Example 20.3 (List reversal)

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

\[ y := \text{nil} \]

\[ \text{while } \neg \text{is-nil}(x) \text{ do } \]

\[ z := y \]

\[ y := x \]

\[ x := x.\text{next} \]

\[ y.\text{next} := z \]

end;

\[ z := \text{nil} \]
### Introducing Pointers

### An Example

#### Example 20.3 (List reversal)

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

\[
\begin{align*}
[y &:= \text{nil}]^1; \\
\text{while } [\neg \text{is-nil}(x)]^2 \text{ do} \\
[z &:= y]^3; \\
[y &:= x]^4; \\
[x &:= x\text{.next}]^5; \\
[y\text{.next} &:= z]^6 \\
\text{end;} \\
[z &:= \text{nil}]^7;
\end{align*}
\]

![Diagram of list reversal process](diagram.png)

\( x \rightarrow n_2 \xrightarrow{\text{next}} n_3 \xrightarrow{\text{next}} \Diamond \)

\( y \rightarrow n_1 \xrightarrow{\text{next}} \Diamond \)

\( z \xrightarrow{\Diamond} \)

(after 4 steps)

\( x \rightarrow n_3 \xrightarrow{\text{next}} \Diamond \)

\( y \rightarrow n_2 \xrightarrow{\text{next}} n_1 \xrightarrow{\text{next}} \Diamond \)

\( z \rightarrow \)
Introducing Pointers

An Example

Example 20.3 (List reversal)

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

\[
\begin{align*}
[y := \text{nil}]^1; \\
\text{while } [\neg \text{is-nil}(x)]^2 \text{ do} \\
[z := y]^3; \\
[y := x]^4; \\
[x := x.\text{next}]^5; \\
[y.\text{next} := z]^6 \\
\text{end}; \\
[z := \text{nil}]^7;
\end{align*}
\]
Introducing Pointers

An Example

Example 20.3 (List reversal)

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

\[
\begin{align*}
[y := \text{nil}]^1; \\
\text{while } [\neg\text{is-nil}(x)]^2 \text{ do} \\
[z := y]^3; \\
[y := x]^4; \\
x := x.\text{next}]^5; \\
y.\text{next} := z]^6 \\
\text{end;} \\
[z := \text{nil}]^7;
\end{align*}
\]

(after 4 steps)

\[
\begin{align*}
x & \rightarrow \text{ } \diamond \\
y & \rightarrow n_3 \text{ next } n_2 \text{ next } n_1 \text{ next } \diamond \\
z & \\
x & \rightarrow \text{ } \diamond \\
y & \rightarrow n_3 \text{ next } n_2 \text{ next } n_1 \text{ next } \diamond \\
z & \rightarrow \diamond
\end{align*}
\]
Introducing Pointers

Heap Configurations

Definition 20.4 (Heap configuration)

A (concrete) heap configuration is given by

\[ H = (Nod, Sel, Var, \sigma, \rightarrow) \]

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
- \( \sigma : Var \rightarrow \mathbb{Z} \cup Nod^\diamond \) is a variable valuation (with \( Nod^\diamond := Nod \cup \{\diamond\} \))
- \( \rightarrow : Nod \times Sel \rightarrow Nod^\diamond \) is a (concrete) heap
  - notation: \( \ n_1 \xrightarrow{sel} n_2 \) for \( ((n_1, sel), n_2) \in \rightarrow \)
Outline of Lecture 20

Recap: Interprocedural Dataflow Analysis – Fixpoint Solution

Effectiveness and Correctness

Context-Sensitive Interprocedural Dataflow Analysis

Pointer Analysis

Introducing Pointers

Shape Graphs
Shape Graphs

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** $X \xrightarrow{sel} Y$ (similarly for $X = \emptyset$ or $Y = \emptyset$)
- **transfer functions** transform (sets of) shape graphs
Example 20.5 (List reversal; cf. Example 20.3)

Concrete heap

\[
\begin{align*}
x & \rightarrow n_1 \xrightarrow{\text{next}} n_2 \xrightarrow{\text{next}} n_3 \xrightarrow{\text{next}} \diamond \\
y & \\
z &
\end{align*}
\]

Shape graph

\[
\begin{align*}
\{x\} & \xrightarrow{\text{next}} \emptyset
\end{align*}
\]
## Shape Graphs II

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

**Concrete heap**

\[
\begin{align*}
x & \rightarrow n_1 \xrightarrow{\text{next}} n_2 \xrightarrow{\text{next}} n_3 \xrightarrow{\text{next}} \Diamond \\
y & \\
z & \\
x & \rightarrow n_1 \xrightarrow{\text{next}} n_2 \xrightarrow{\text{next}} n_3 \xrightarrow{\text{next}} \Diamond \\
y & \\
z & \\
\end{align*}
\]

**Shape graph**

\[
\begin{align*}
\{x\} & \xrightarrow{\text{next}} \emptyset \\
\{x\} & \xrightarrow{\text{next}} \emptyset
\end{align*}
\]
Example 20.5 (List reversal; cf. Example 20.3)

Concrete heap

\[
\text{x} \rightarrow n_1 \xrightarrow{\text{next}} n_2 \xrightarrow{\text{next}} n_3 \xrightarrow{\text{next}} \Diamond \\
\text{y} \rightarrow \Diamond \\
\text{z} \\
\text{x} \rightarrow n_1 \xrightarrow{\text{next}} n_2 \xrightarrow{\text{next}} n_3 \xrightarrow{\text{next}} \Diamond \\
\text{y} \rightarrow \Diamond \\
\text{z} \rightarrow \Diamond
\]
Example 20.5 (List reversal; cf. Example 20.3)

Concrete heap

```
x -> n1 -> next -> n2 -> next -> n3 -> next
```

Shape graph

```
{x} -> next -> ∅
```

```
{x, y} -> next -> ∅
```
Example 20.5 (List reversal; cf. Example 20.3)

Concrete heap

Shape graph
Example 20.5 (List reversal; cf. Example 20.3)

Concrete heap

Shape graph
Example 20.5 (List reversal; cf. Example 20.3)

Concrete heap

\[ \begin{align*}
  x & \xrightarrow{} n_2 \xrightarrow{\text{next}} n_3 \xrightarrow{\text{next}} \Diamond \\
  y & \xrightarrow{} n_1 \xrightarrow{\text{next}} \Diamond \\
  z & \xrightarrow{} \Diamond
\end{align*} \]

Shape graph

\[ \begin{align*}
  \{x\} & \xrightarrow{\text{next}} \emptyset \\
  \{y\} & \xrightarrow{} \{z\}
\end{align*} \]
Example 20.5 (List reversal; cf. Example 20.3)

Concrete heap

(after 4 steps)

\[ \text{x} \rightarrow n_3 \xrightarrow{\text{next}} \]
\[ \text{y} \rightarrow n_2 \xrightarrow{\text{next}} n_1 \xrightarrow{\text{next}} \]
\[ \text{z} \]

Shape graph

\[ \{x\} \]
\[ \{y\} \xrightarrow{\text{next}} \{z\} \]
\[ \{y\} \xrightarrow{\text{next}} \{z\} \xrightarrow{\text{next}} \emptyset \]
Example 20.5 (List reversal; cf. Example 20.3)

Concrete heap

(after 4 steps)

Shape graph
Definition 20.6 (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: \( X \xrightarrow{\text{sel}} Y \) 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 \( X \xrightarrow{\text{sel}} Y \) and \( X \xrightarrow{\text{sel}} Z \Rightarrow Y = Z \)
(target location is unique if source node is unique)

\( \text{SG} \) denotes the set of all shape graphs.
Definition 20.7

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

- Concrete:

$$y \rightarrow \bullet \leftarrow \bullet$$

$$z \rightarrow \bullet \leftarrow \bullet$$

- Abstract:

$$Y = \{y\} \xleftrightarrow{sel} X = \emptyset \xleftrightarrow{sel} Z = \{z\}$$
Shape Graphs

Shape Graphs and Concrete Heap Properties

Example 20.8

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

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

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

- $x.\text{sel1} = y.\text{sel2} \neq \text{nil}$ (sharing)
  \[\implies \exists X, Y, Z \in \text{Abs} : x \in X, y \in Y, X \rightarrow Z \leftrightarrow Y\]

("\leftarrow\rightarrow" only valid if $Z \neq \emptyset$)