Semantics and Verification of Software

Lecture 23: Dataflow Analysis X (Interprocedural Fixpoint Solution & Wrap-Up)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/svsw08/

Winter semester 2008/09

Outline

Extending the Syntax

Syntactic categories:

Category	Domain	Meta variable
Procedure identifiers	$PVar = \{\mathtt{P}, \mathtt{Q}, \ldots\}$	P
Procedure declarations	$PDec$	p
Commands (statements)	Cmd	c

Context-free grammar:

$$p ::= \mathtt{proc}\ [P(\mathtt{val}\ x, \mathtt{res}\ y)]^{l_n}\ \mathtt{is}\ c\ [\mathtt{end}]^{l_x}\mathtt{;}p \mid \varepsilon \in PDec$$

$$c ::= [\mathtt{skip}]^l \mid [x\ \mathtt{:=}\ a]^l \mid c_1\mathtt{;}c_2 \mid \mathtt{if}\ [b]^l\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ [b]^l\ \mathtt{do}\ c \mid [\mathtt{call}\ P(a, x)]^{l_c}_{l_r} \in Cmd$$

- All labels and procedure names in program $p\,c$ distinct
- In $\mathtt{proc}\ [P(\mathtt{val}\ x, \mathtt{res}\ y)]^{l_n}\ \mathtt{is}\ c\ [\mathtt{end}]^{l_x}$, l_n (l_x) refers to the entry (exit) of P
- In $[\mathtt{call}\ P(a, x)]^{l_c}_{l_r}$, l_c (l_r) refers to the call of (return from) P
- First parameter call-by-value, second call-by-result

Making Context Explicit

- **Goal:** adapt fixpoint solution to avoid invalid paths
- **Approach:** encode call history into data flow properties (use stacks D^+ as dataflow version of runtime stack)
- Non-procedural constructs (`skip`, assignments, tests): operate only on topmost element
- `call`: computes new topmost entry from current and pushes it
- return: removes topmost entry and combines it with underlying entry

The Interprocedural Extension I

Definition (Interprocedural extension (forward analysis))

Let $S = (L, E, F, (D, \sqsubseteq), \iota, \varphi)$ be a dataflow system. The interprocedural extension of S is given by

$$\hat{S} := (L, E, F, (\hat{D}, \hat{\sqsubseteq}), \hat{\iota}, \hat{\varphi})$$

where

- $\hat{D} := D^+$
- $d_1 \dots d_n \mathrel{\hat{\sqsubseteq}} d'_1 \dots d'_n$ iff $d_i \sqsubseteq d'_i$ for every $1 \leq i \leq n$
- $\hat{\iota} := \iota \in D^+$
- for each $l \in L \setminus \{l_c, l_n, l_x, l_r \mid (l_c, l_n, l_x, l_r) \in IF\}$, $\hat{\varphi}_l : D^+ \to D^+$ is given by $\hat{\varphi}_l(dw) := \varphi_l(d)w$
- for each $(l_c, l_n, l_x, l_r) \in IF$, $\hat{\varphi}_l : D^+ \to D^+$ is given by
  - $\hat{\varphi}_{l_c}(dw) := \varphi_{l_c}(d)dw$
  - $\hat{\varphi}_{l_n}(w) := w$
  - $\hat{\varphi}_{l_x}(w) := w$
  - $\hat{\varphi}_{l_r}(d'dw) := \varphi_{l_r}(d', d)w$

The Interprocedural Extension II

Remark: the schema

1. $\hat{\varphi}_{l_c}(dw) := \varphi_{l_c}(d)dw$
2. $\hat{\varphi}_{l_n}(w) := w$
3. $\hat{\varphi}_{l_x}(w) := w$
4. $\hat{\varphi}_{l_r}(d'dw) := \varphi_{l_r}(d', d)w$

can be generalized by allowing a modification of the topmost entry in 2. and 3. (local variables, ...)

The Interprocedural Extension III

Example (Constant Propagation (cf. Lecture 19))

$\hat{S} := (L, E, F, (\hat{D}, \hat{\sqsubseteq}), \hat{\iota}, \hat{\varphi})$ is determined by

- $D := \{\delta \mid \delta : Var_c \to \mathbb{Z} \cup \{\bot, \top\}\}$
- $\bot \sqsubseteq z \sqsubseteq \top$
- $\iota := \delta_\top \in D$
- for each $l \in L \setminus \{l_c, l_n, l_x, l_r \mid (l_c, l_n, l_x, l_r) \in IF\}$,
  $$\varphi_l(\delta) := \begin{cases} \delta & \text{if } B^l = \texttt{skip} \text{ or } B^l \in BExp \\ \delta[x \mapsto \mathfrak{A}[\![a]\!]\delta] & \text{if } B^l = (x \texttt{ := } a) \end{cases}$$
- whenever $p\,c$ contains $[\texttt{call } P(a,z)]_{l_r}^{l_c}$ and $\texttt{proc } [P(\texttt{val } x, \texttt{res } y)]^{l_n} \texttt{ is } c \; [\texttt{end}]^{l_x}$,
  - `call`: set input parameter and reset output parameter
    $\varphi_{l_c}(\delta) := \delta[x \mapsto \mathfrak{A}[\![a]\!]\delta, y \mapsto \top]$
  - return: propagate output parameter to caller by overwriting old value
    $\varphi_{l_r}(\delta', \delta) := \delta[z \mapsto \delta'(y)]$

The Equation System I

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

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

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

- for actual dataflow information: $\mathsf{AI}_l \in D^+$ (extension of intraprocedural AI)
- for transfer functions of single nodes: $f_l : D^+ \to D^+$ (extension of intraprocedural transfer functions)
- for transfer functions of complete procedures: $F_l : D^+ \to D^+$ ($F_l(w)$ yields information at l if surrounding procedure is called with information $w \implies$ full procedure represented by F_{l_x})

The Equation System II

Formal definition:

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

(if l not an exit label)

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

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

(if l occurs in procedure)

As before: induces monotonic functional on lattice with ACC

$\Longrightarrow$ least fixpoint effectively computable

Outline

The Equation System III

Example 23.1 (Constant Propagation)

Program:

```
proc [P(val x, res y)]^1 is
  [y := 2*(x-1)]^2;
[end]^3;
[call P(2, z)]^4_5;
[call P(z, z)]^6_7;
[skip]^8
```

Dataflow equations:

$$\begin{aligned}
\mathsf{AI}_1 &= \hat{\varphi}_4(\mathsf{AI}_4) \sqcup \hat{\varphi}_6(\mathsf{AI}_6)\\
\mathsf{AI}_2 &= f_1(\mathsf{AI}_1)\\
\mathsf{AI}_3 &= f_2(\mathsf{AI}_2)\\
\mathsf{AI}_4 &= \iota = \top\top\top\\
\mathsf{AI}_6 &= f_4(\mathsf{AI}_4)\\
\mathsf{AI}_8 &= f_6(\mathsf{AI}_6)
\end{aligned}$$

Node transfer functions:

$$\begin{aligned}
f_1(\delta w) &= \hat{\varphi}_1(\delta w) = \delta w\\
f_2(\delta w) &= \hat{\varphi}_2(\delta w) = \delta[y \mapsto \mathfrak{A}[\![\texttt{2*(x-1)}]\!]\delta]w\\
f_3(\delta w) &= \hat{\varphi}_3(\delta w) = \delta w\\
f_4(\delta w) &= \hat{\varphi}_5(F_3(\hat{\varphi}_4(\delta w)))\\
f_6(\delta w) &= \hat{\varphi}_7(F_3(\hat{\varphi}_6(\delta w)))\\
f_8(\delta w) &= \hat{\varphi}_8(\delta w) = \delta w
\end{aligned}$$

$$\begin{aligned}
\hat{\varphi}_4(\delta w) &= \delta[x \mapsto 2, y \mapsto \top]\delta w\\
\hat{\varphi}_5(\delta'\delta w) &= \delta[z \mapsto \delta'(y)]w\\
\hat{\varphi}_6(\delta w) &= \delta[x \mapsto \delta(z), y \mapsto \top]\delta w\\
\hat{\varphi}_7(\delta'\delta w) &= \delta[z \mapsto \delta'(y)]w
\end{aligned}$$

Procedure transfer functions:

$$\begin{aligned}
F_1(\delta w) &= \delta w\\
F_2(\delta w) &= f_1(F_1(\delta w)) = \delta w\\
F_3(\delta w) &= f_2(F_2(\delta w)) = \delta[y \mapsto \mathfrak{A}[\![\texttt{2*(x-1)}]\!]\delta]w
\end{aligned}$$

Fixpoint iteration:

on the board

The Fixpoint Iteration

For the fixpoint iteration it is important that the auxiliary functions only operates on the topmost element of the stack (without proof):

Lemma 23.2

For every $l \in L$, $d \in D$, and $w \in D^*$,

$$f_l(dd'w) = f_l(d)w \text{ and } F_l(dw) = F_l(d)w$$

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

Outline

Context-Sensitive Dataflow Analysis

- **Observation:** MVP and fixpoint solution maintain proper relationship between procedure calls and returns

Context-Sensitive Dataflow Analysis

- **Observation:** MVP and fixpoint solution maintain proper relationship between procedure calls and returns
- **But:** do not distinguish between different procedure calls

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

  - information about calling states combined for all call sites
  - procedure body only analyzed once using combined information
  - resulting information used at all return points

$\Longrightarrow$ "context-insensitive"

Context-Sensitive Dataflow Analysis

- **Observation:** MVP and fixpoint solution maintain proper relationship between procedure calls and returns
- **But:** do not distinguish between different procedure calls

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

  - information about calling states combined for all call sites
  - procedure body only analyzed 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"
  - more precise
  - more costly

Shape Analysis I

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

Shape Analysis I

- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps

Shape Analysis I

- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps
- **Goal:** shape analysis = approximative analysis of heap data structures

Shape Analysis I

- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps
- **Goal:** shape analysis = approximative analysis of heap data structures
- Interesting information:
  - data types (to avoid type errors, such as dereferencing `nil`)
  - sharing (different pointer variables referencing same address; aliasing)
  - reachability of nodes (garbage collection)
  - disjointness of heap regions (parallelizability)
  - shapes (lists, trees, absence of cycles, ...)

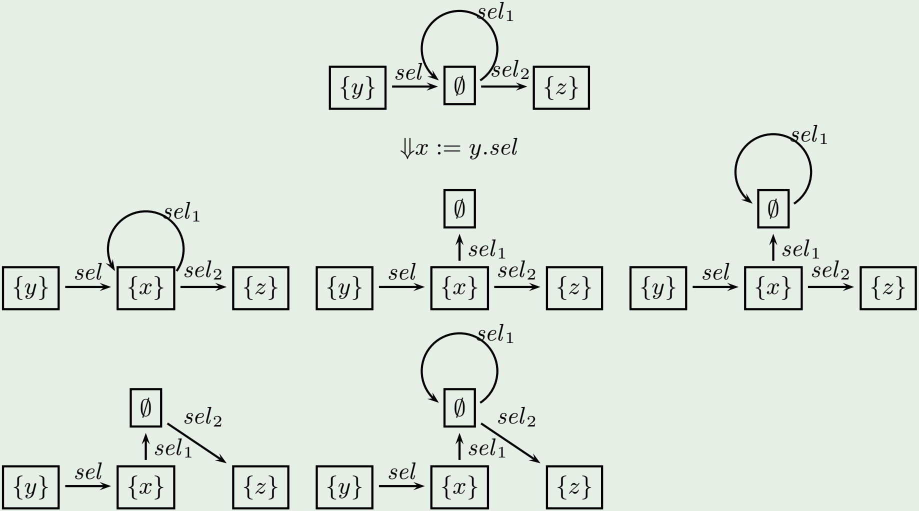
Shape Analysis I

- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps
- **Goal:** shape analysis = approximative analysis of heap data structures
- Interesting information:
  - data types (to avoid type errors, such as dereferencing `nil`)
  - sharing (different pointer variables referencing same address; aliasing)
  - reachability of nodes (garbage collection)
  - disjointness of heap regions (parallelizability)
  - shapes (lists, trees, absence of cycles, ...)
- Representation of (infinitely many) concrete heap states by (finitely many) abstract shape graphs
  - abstract nodes A = sets of variables (interpretation: $x \in A$ iff x points to concrete node represented by A)
  - $\emptyset$ represents all concrete nodes that are not directly reachable
  - transfer functions transform (sets of) shape graphs

Shape Analysis I

- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps
- **Goal:** shape analysis = approximative analysis of heap data structures
- Interesting information:
  - data types (to avoid type errors, such as dereferencing `nil`)
  - sharing (different pointer variables referencing same address; aliasing)
  - reachability of nodes (garbage collection)
  - disjointness of heap regions (parallelizability)
  - shapes (lists, trees, absence of cycles, ...)
- Representation of (infinitely many) concrete heap states by (finitely many) abstract shape graphs
  - abstract nodes A = sets of variables (interpretation: $x \in A$ iff x points to concrete node represented by A)
  - $\emptyset$ represents all concrete nodes that are not directly reachable
  - transfer functions transform (sets of) shape graphs
- see [Nielson/Nielson/Hankin 2005, Sct. 2.6]

Example 23.3



Correctness of Analyses

- **So far:** semantics and dataflow analysis of programs independent

Correctness of Analyses

- **So far:** semantics and dataflow analysis of programs independent
- Of course both are (and should be) related!

Correctness of Analyses

- **So far:** semantics and dataflow analysis of programs independent
- Of course both are (and should be) related!
- To this aim: introduce small-step operational semantics operating on program labels

$$\langle l_0, \sigma_0 \rangle \to \ldots \langle l_n, \sigma_n \rangle \to \sigma_{n+1}$$

where $l_i \in L$ and $\sigma_i : \mathit{Var} \to \mathbb{Z}$

Correctness of Analyses

- **So far:** semantics and dataflow analysis of programs independent
- Of course both are (and should be) related!
- To this aim: introduce small-step operational semantics operating on program labels

  $$\langle l_0, \sigma_0 \rangle \rightarrow \ldots \langle l_n, \sigma_n \rangle \rightarrow \sigma_{n+1}$$

  where $l_i \in L$ and $\sigma_i : \mathit{Var} \rightarrow \mathbb{Z}$
- **Example:** correctness of Constant Propagation

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

Correctness of Analyses

- **So far:** semantics and dataflow analysis of programs independent
- Of course both are (and should be) related!
- To this aim: introduce small-step operational semantics operating on program labels

$$\langle l_0, \sigma_0 \rangle \to \ldots \langle l_n, \sigma_n \rangle \to \sigma_{n+1}$$

  where $l_i \in L$ and $\sigma_i : Var \to \mathbb{Z}$
- **Example:** correctness of Constant Propagation

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