Semantics and Verification of Software

Lecture 23: Wrap–Up

Thomas Noll

Lehrstuhl für Informatik 2
RWTH Aachen University
noll@cs.rwth-aachen.de

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

Summer semester 2007

Outline

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

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

where

- $\hat{D} := D^+$
- $d_1 \dots d_n \,\hat{\sqsubseteq}\, d'_1 \dots d'_n$ iff $d_i \sqsubseteq d'_i$ for every $1 \leq i \leq n$
- $\hat{\iota} := \iota \in D^+$
- for each $l \in Lab \setminus \{l_c, l_n, l_x, l_r \mid (l_c, l_n, l_x, l_r) \in IF\}$, $\hat{\varphi}_l : D^+ \to D^+$ 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
  - $\hat{\varphi}_{l_c}(dw) := \varphi_{l_c}(d)dw$
  - $\hat{\varphi}_{l_n}(w) := w$
  - $\hat{\varphi}_{l_x}(w) := w$
  - $\hat{\varphi}_{l_r}(d'dw) := d''w$ where $d'' := \varphi^1_{l_r}(d) \sqcup \varphi^2_{l_r}(d')$

The Interprocedural Extension II

Example (Constant Propagation (cf. Lecture 19))

$\hat{S} := (Lab, E, F, (\hat{D}, \hat{\sqsubseteq}), \hat{\iota}, \hat{\varphi})$ 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 $l \in Lab \setminus \{l_c, l_n, l_x, l_r \mid (l_c, l_n, l_x, l_r) \in IF\}$,
  $$\varphi_l(\delta) := \begin{cases} \delta & \text{if } B^l = \texttt{skip} \text{ or } B^l \in BExp \\ \delta[x \mapsto \mathfrak{A}[\![a]\!]\delta] & \text{if } B^l = (x \texttt{ := } a) \end{cases}$$
- whenever $p\,c$ contains $[\texttt{call } P(a, z)]^{l_c}_{l_r}$ 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 resetting old z value and copying y to z
    $\varphi^1_{l_r}(\delta) := \delta[z \mapsto \bot]$ $\quad$ $\varphi^2_{l_r}(\delta') := \delta'[x \mapsto \bot, y \mapsto \bot, z \mapsto \delta'(y)]$

The Equation System

For an interprocedural dataflow system $\hat{S} := (Lab, E, F, (\hat{D}, \hat{\sqsubseteq}), \hat{\iota}, \hat{\varphi})$, 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 $l \in Lab$):

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

Outline

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

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

  - 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

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

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

  - 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

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

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

  - 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)
- **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]

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]

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]

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]

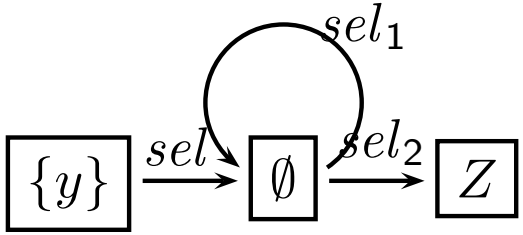
Shape Analysis I

- **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]

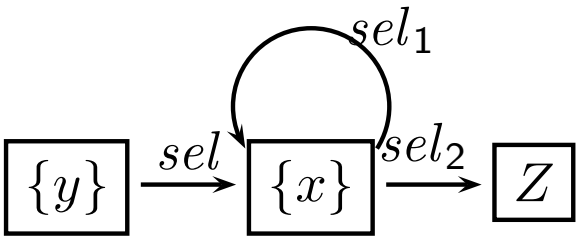
Shape Analysis I

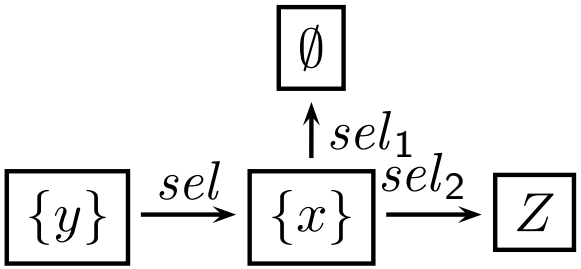
- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps
- **Goal:** shape analysis = approximative analysis of heap data structures
- Interesting information:
  - data types (to avoid type errors, such as dereferencing `nil`)
  - 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]

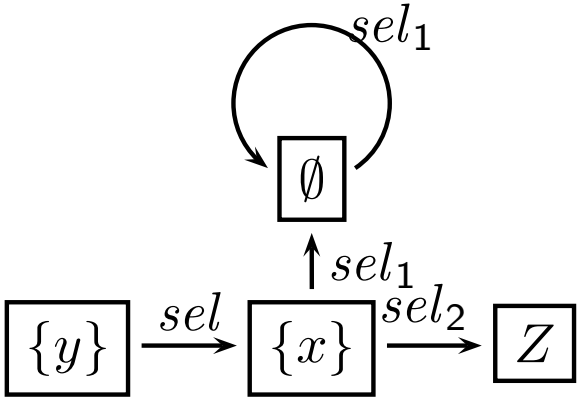
Shape Analysis II

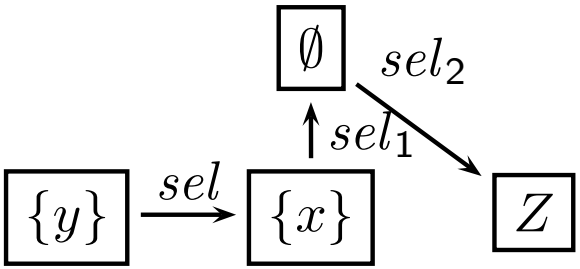


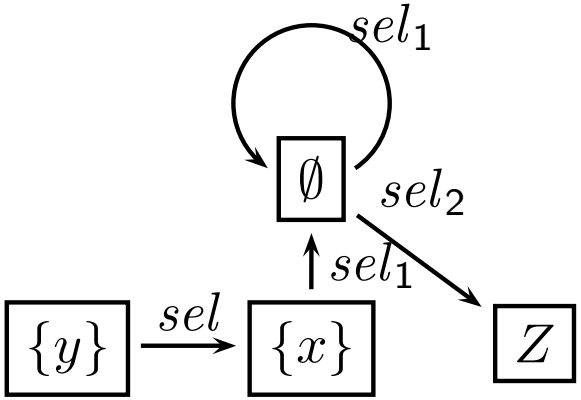
$\Downarrow x \leftarrow y.sel$











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

  Let $c_0 \in Cmd$ with $l_0 = \mathsf{init}(c_0)$, and let $l \in Lab_{c_0}$, $x \in Var$, 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]

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

  Let $c_0 \in Cmd$ with $l_0 = \mathsf{init}(c_0)$, and let $l \in Lab_{c_0}$, $x \in Var$, 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]

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 $l_i \in Lab$ and $\sigma_i : Var \to \mathbb{Z}$

- **Example:** correctness of Constant Propagation

Let $c_0 \in Cmd$ with $l_0 = \mathsf{init}(c_0)$, and let $l \in Lab_{c_0}$, $x \in Var$, 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]

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

  Let $c_0 \in Cmd$ with $l_0 = \mathsf{init}(c_0)$, and let $l \in Lab_{c_0}$, $x \in Var$, 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]

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

  Let $c_0 \in Cmd$ with $l_0 = \mathsf{init}(c_0)$, and let $l \in Lab_{c_0}$, $x \in Var$, 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]

Outline

Semantics of Functional Languages I

- Program = list of function definitions
- Simplest setting: first–order function definitions of the form

$$f(x_1, \ldots, x_n) = t$$

  - function name f
  - formal parameters $x_1, \ldots, x_n$
  - term t over (base and defined) function calls and $x_1, \ldots, x_n$
- **Operational semantics** (only function calls)
  - call–by–value case:

$$\frac{t_1 \to z_1 \ \ldots \ t_n \to z_n \ \ t[x_1 \mapsto z_1, \ldots, x_n \mapsto z_n] \to z}{f(t_1, \ldots, t_n) \to z}$$

  - call–by–name case:

$$\frac{t[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n] \to z}{f(t_1, \ldots, t_n) \to z}$$

Semantics of Functional Languages I

- Program = list of function definitions
- Simplest setting: first–order function definitions of the form

$$f(x_1, \ldots, x_n) = t$$

  - function name f
  - formal parameters $x_1, \ldots, x_n$
  - term t over (base and defined) function calls and $x_1, \ldots, x_n$
- **Operational semantics** (only function calls)
  - call–by–value case:

$$\frac{t_1 \to z_1 \;\ldots\; t_n \to z_n \;\; t[x_1 \mapsto z_1, \ldots, x_n \mapsto z_n] \to z}{f(t_1, \ldots, t_n) \to z}$$

  - call–by–name case:

$$\frac{t[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n] \to z}{f(t_1, \ldots, t_n) \to z}$$

Semantics of Functional Languages I

- Program = list of function definitions
- Simplest setting: first–order function definitions of the form
$$f(x_1, \ldots, x_n) = t$$
  - function name f
  - formal parameters $x_1, \ldots, x_n$
  - term t over (base and defined) function calls and $x_1, \ldots, x_n$
- **Operational semantics** (only function calls)
  - call–by–value case:
$$\frac{t_1 \to z_1 \ \ldots \ t_n \to z_n \ \ t[x_1 \mapsto z_1, \ldots, x_n \mapsto z_n] \to z}{f(t_1, \ldots, t_n) \to z}$$
  - call–by–name case:
$$\frac{t[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n] \to z}{f(t_1, \ldots, t_n) \to z}$$

Semantics of Functional Languages II

- **Denotational semantics**
  - program induces call–by–value and call–by–name functional
  - monotonic and continuous w.r.t. graph inclusion
  - semantics := least fixpoint (Tarski/Knaster Theorem)
  - coincides with operational semantics
- Extensions: higher–order types, data types, ...
- see *Functional Programming* course [Giesl] and [Winskel 1996, Sct. 9]

Semantics of Functional Languages II

- **Denotational semantics**
  - program induces call–by–value and call–by–name functional
  - monotonic and continuous w.r.t. graph inclusion
  - semantics := least fixpoint (Tarski/Knaster Theorem)
  - coincides with operational semantics
- Extensions: higher–order types, data types, ...
- see *Functional Programming* course [Giesl] and [Winskel 1996, Sct. 9]

Semantics of Functional Languages II

- **Denotational semantics**
  - program induces call–by–value and call–by–name functional
  - monotonic and continuous w.r.t. graph inclusion
  - semantics := least fixpoint (Tarski/Knaster Theorem)
  - coincides with operational semantics
- Extensions: higher–order types, data types, ...
- see *Functional Programming* course [Giesl] and [Winskel 1996, Sct. 9]

Semantics of Concurrent Languages

- **Problem:** "classical" view of sequential systems

  $$\mathsf{Program} : \mathsf{Input} \to \mathsf{Output}$$

  not adequate for concurrent settings
- Missing: aspect of interaction
- Typical approach:
  - concurrency modelled by interleaving
  - interaction modelled by (explicit) communication
- Example: Milner's *Calculus of Communicating Systems (CCS)*
- Syntax: $P ::= 0 \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid \ldots$
- (Operational) Semantics: labelled transition systems defined by transition rules of the form

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \quad \frac{P \xrightarrow{\alpha} P' \;\; Q \xrightarrow{\bar{\alpha}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \ldots$$

- see course on *Modelling Concurrent and Probabilistic Systems* in WS 2007/08 [Katoen, Noll] and [Winskel 1996, Sct. 14]

Semantics of Concurrent Languages

- **Problem:** "classical" view of sequential systems

$$\mathsf{Program} : \mathsf{Input} \to \mathsf{Output}$$

  not adequate for concurrent settings
- Missing: aspect of interaction
- Typical approach:
  - concurrency modelled by interleaving
  - interaction modelled by (explicit) communication
- Example: Milner's *Calculus of Communicating Systems (CCS)*
- Syntax: $P ::= 0 \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid \ldots$
- (Operational) Semantics: labelled transition systems defined by transition rules of the form

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \quad \frac{P \xrightarrow{\alpha} P' \; Q \xrightarrow{\bar{\alpha}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \ldots$$

- see course on *Modelling Concurrent and Probabilistic Systems* in WS 2007/08 [Katoen, Noll] and [Winskel 1996, Sct. 14]

Semantics of Concurrent Languages

- **Problem:** "classical" view of sequential systems

$$\mathsf{Program} : \mathsf{Input} \to \mathsf{Output}$$

  not adequate for concurrent settings
- Missing: aspect of interaction
- Typical approach:
  - concurrency modelled by interleaving
  - interaction modelled by (explicit) communication
- Example: Milner's *Calculus of Communicating Systems (CCS)*
- Syntax: $P ::= 0 \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid \ldots$
- (Operational) Semantics: labelled transition systems defined by transition rules of the form

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \frac{P \xrightarrow{\alpha} P' \; Q \xrightarrow{\bar{\alpha}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \ldots$$

- see course on *Modelling Concurrent and Probabilistic Systems* in WS 2007/08 [Katoen, Noll] and [Winskel 1996, Sct. 14]

Semantics of Concurrent Languages

- **Problem:** "classical" view of sequential systems

$$\mathsf{Program} : \mathsf{Input} \to \mathsf{Output}$$

  not adequate for concurrent settings
- Missing: aspect of interaction
- Typical approach:
  - concurrency modelled by interleaving
  - interaction modelled by (explicit) communication
- Example: Milner's *Calculus of Communicating Systems (CCS)*
- Syntax: $P ::= 0 \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid \ldots$
- (Operational) Semantics: labelled transition systems defined by transition rules of the form

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \frac{P \xrightarrow{\alpha} P' \; Q \xrightarrow{\bar{\alpha}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \; \ldots$$

- see course on *Modelling Concurrent and Probabilistic Systems* in WS 2007/08 [Katoen, Noll] and [Winskel 1996, Sct. 14]

Semantics of Concurrent Languages

- **Problem:** "classical" view of sequential systems

$$\mathsf{Program} : \mathsf{Input} \to \mathsf{Output}$$

  not adequate for concurrent settings
- Missing: aspect of interaction
- Typical approach:
  - concurrency modelled by interleaving
  - interaction modelled by (explicit) communication
- Example: Milner's *Calculus of Communicating Systems (CCS)*
- Syntax: $P ::= 0 \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid \ldots$
- (Operational) Semantics: labelled transition systems defined by transition rules of the form

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \frac{P \xrightarrow{\alpha} P' \; Q \xrightarrow{\bar{\alpha}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \ldots$$

- see course on *Modelling Concurrent and Probabilistic Systems* in WS 2007/08 [Katoen, Noll] and [Winskel 1996, Sct. 14]

Semantics of Concurrent Languages

- **Problem:** "classical" view of sequential systems

$$\mathsf{Program} : \mathsf{Input} \to \mathsf{Output}$$

  not adequate for concurrent settings
- Missing: aspect of interaction
- Typical approach:
  - concurrency modelled by interleaving
  - interaction modelled by (explicit) communication
- Example: Milner's *Calculus of Communicating Systems (CCS)*
- Syntax: $P ::= 0 \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid \ldots$
- (Operational) Semantics: labelled transition systems defined by transition rules of the form

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \quad \frac{P \xrightarrow{\alpha} P' \;\; Q \xrightarrow{\bar{\alpha}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \ldots$$

- see course on *Modelling Concurrent and Probabilistic Systems* in WS 2007/08 [Katoen, Noll] and [Winskel 1996, Sct. 14]

Semantics of Concurrent Languages

- **Problem:** "classical" view of sequential systems

$$\mathsf{Program} : \mathsf{Input} \to \mathsf{Output}$$

  not adequate for concurrent settings
- Missing: aspect of interaction
- Typical approach:
  - concurrency modelled by interleaving
  - interaction modelled by (explicit) communication
- Example: Milner's *Calculus of Communicating Systems (CCS)*
- Syntax: $P ::= 0 \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid \ldots$
- (Operational) Semantics: labelled transition systems defined by transition rules of the form

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \quad \frac{P \xrightarrow{\alpha} P' \;\; Q \xrightarrow{\bar{\alpha}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \ldots$$

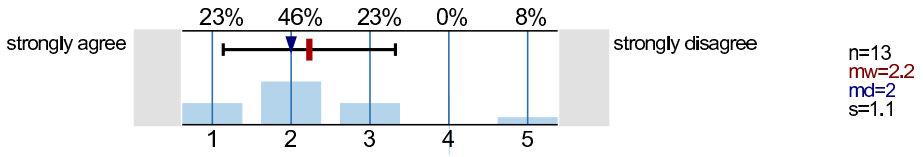
- see course on *Modelling Concurrent and Probabilistic Systems* in WS 2007/08 [Katoen, Noll] and [Winskel 1996, Sct. 14]

Outline

Overall Evaluation of Lecture I

Overall evaluation (SS 2007)

1. I learned a lot in this course.

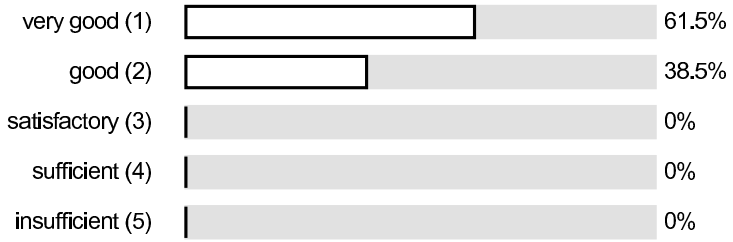


09.07.2007 EvaSys Auswertung Seite 5

(see `http://www-i2.informatik.rwth-aachen.de/i2/svsw/` for full reports)

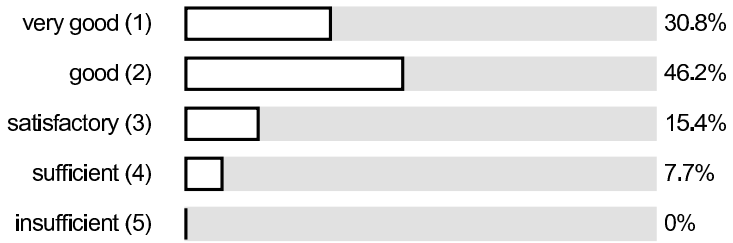
Overall Evaluation of Lecture II

2. I would grade the instructor with:



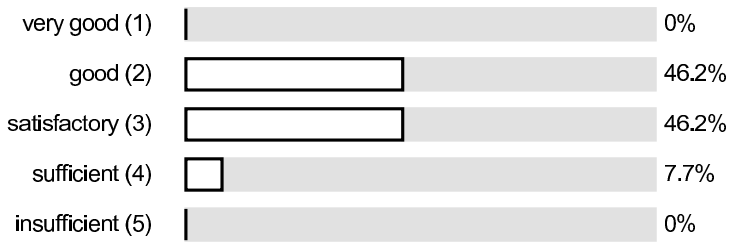
n=13
mw=1.4
s=0.5

3. I would grade the course with:



n=13
mw=2
s=0.9

4. I would grade my own effort with:

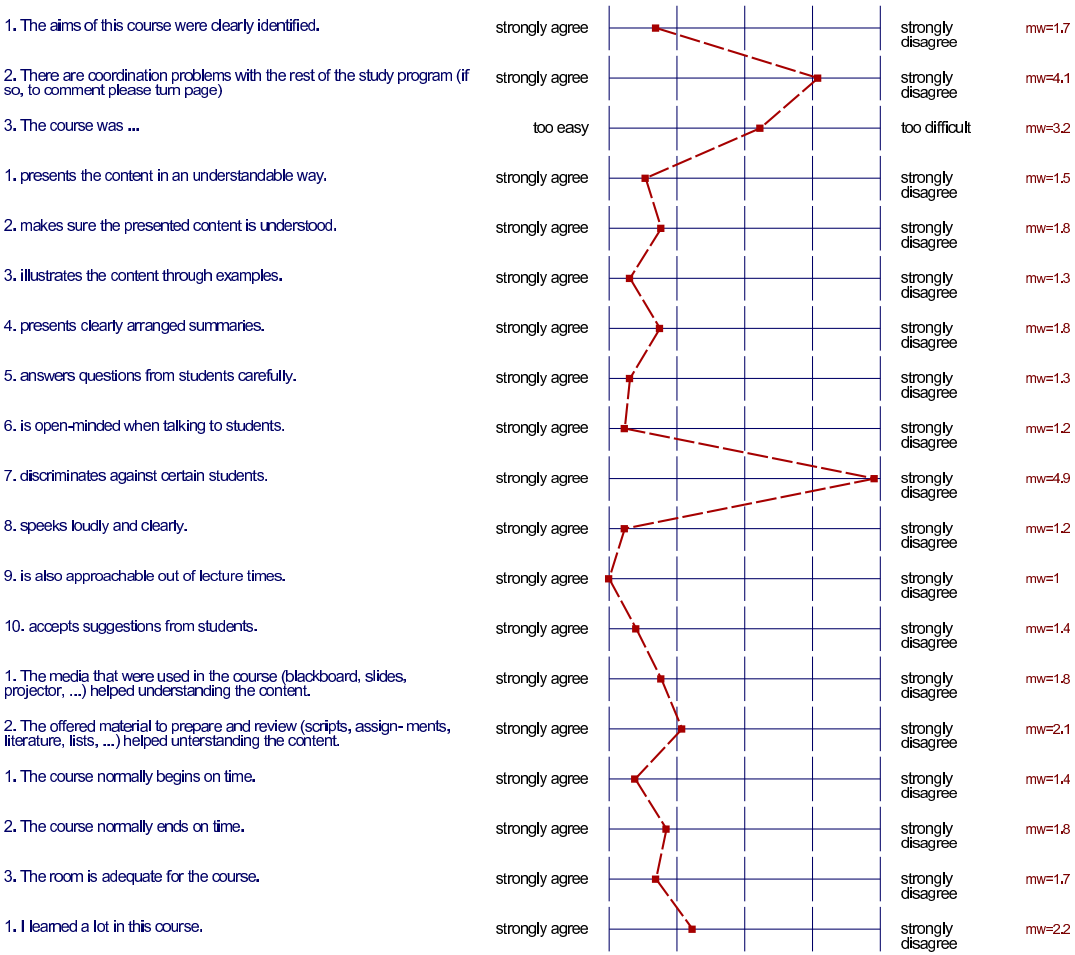


n=13
mw=2.6
s=0.7

Evaluation Profile of Lecture

Profillinie

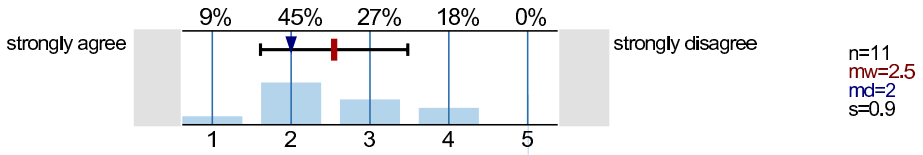
Teilbereich: Informatik
Name der/des Lehrenden: Priv.-Doz. Dr.rer.nat. Thomas Noll
Titel der Lehrveranstaltung: (Name der Umfrage) Semantik und Verifikation von Software (1028771)



Overall Evaluation of Exercise Class I

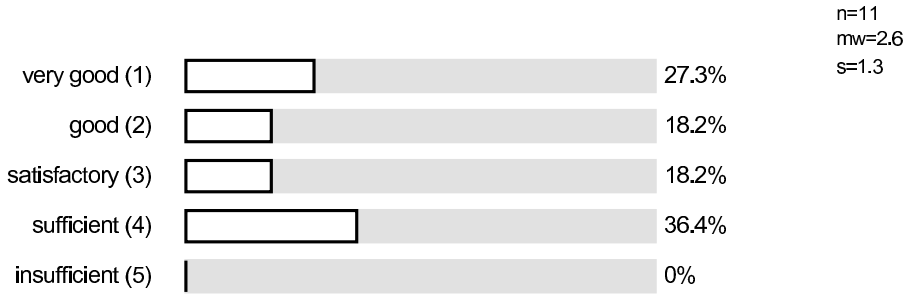
Overall evaluation (SS 2007)

1. I learned a lot in this course.

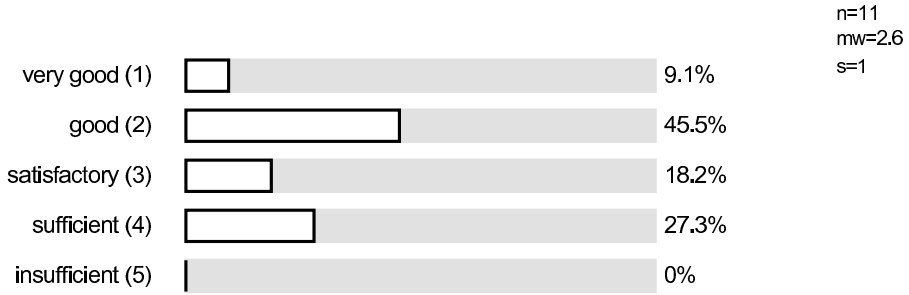


Overall Evaluation of Exercise Class II

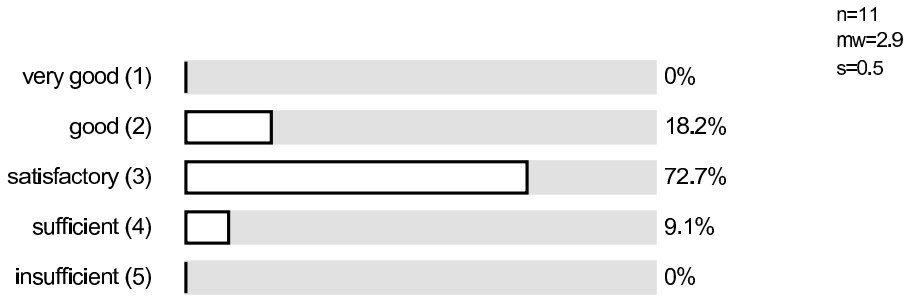
2. I would grade the instructor with:



3. I would grade the course with:



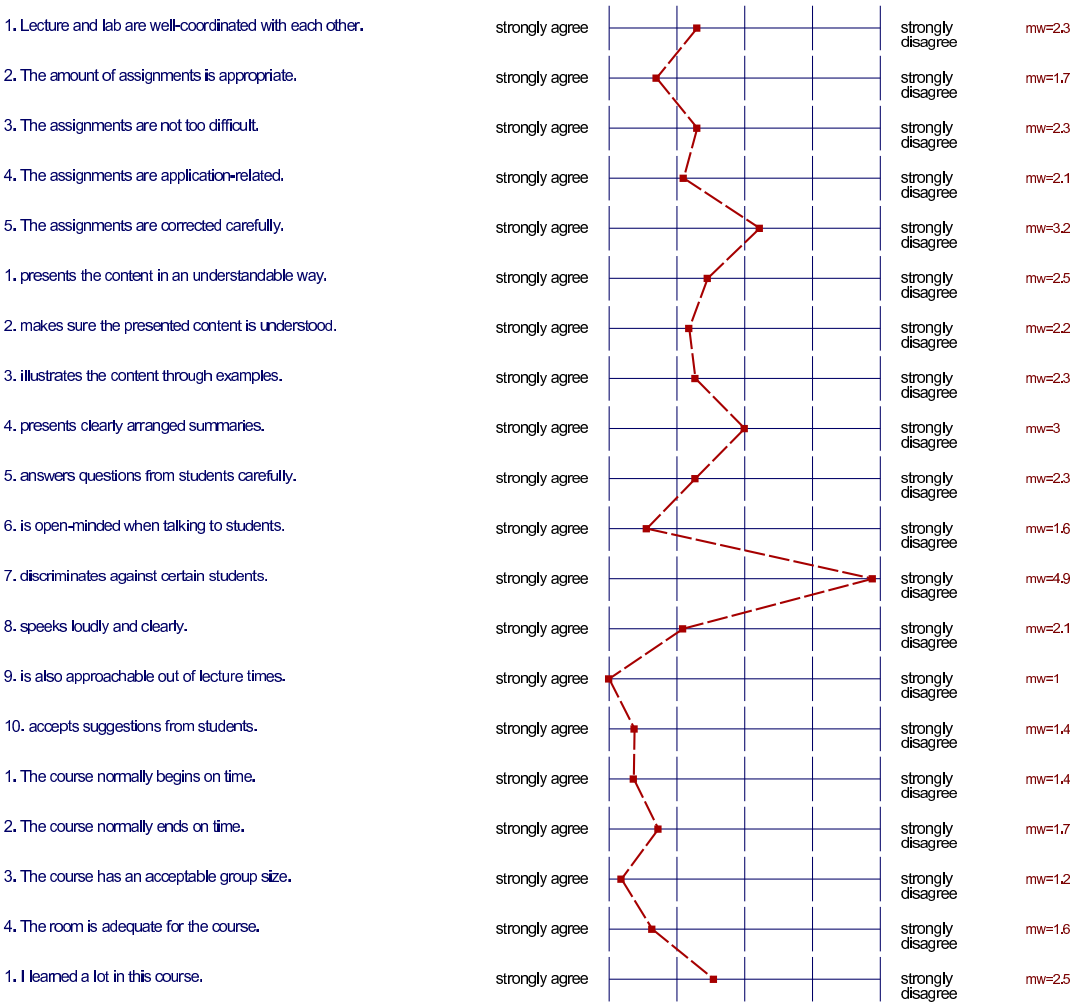
4. I would grade my own effort with:



Evaluation Profile of Exercise Class

Profillinie

Teilbereich: Informatik
Name der/des Lehrenden: Priv.-Doz. Dr.rer.nat. Thomas Noll
Titel der Lehrveranstaltung: Semantik und Verifikation von Software
(Name der Umfrage)



Possible Improvements

Lecture:

1. Slides more "self–contained"
2. Repetition of proof principles
   - good for beginners
   - boring for experienced listeners
3. Double covering of denotational semantics by *Functional Programming* course

Points 2 and 3 perhaps call for a separate, introductory course on "Universal Algebra"

Exercise class:

1. Questions and example solutions more "reliable"
2. Comments and suggestions in correction of exercises
3. More explanations to solutions?

Possible Improvements

Lecture:

1. Slides more "self–contained"
2. Repetition of proof principles
   - good for beginners
   - boring for experienced listeners
3. Double covering of denotational semantics by *Functional Programming* course

Points 2 and 3 perhaps call for a separate, introductory course on "Universal Algebra"

Exercise class:

1. Questions and example solutions more "reliable"
2. Comments and suggestions in correction of exercises
3. More explanations to solutions?