Analysis of concurrent programs

Abstract Interpretation Course

Antoine Miné

ENS & CNRS
Paris, France

29 November 2013
East China Normal University
Shanghai, P.R. China
Motivation: concurrent programming

**Concurrent programming:**
Decompose a program into a set of processes each following its own instruction sequence independently and (loosely) interacting

**Why concurrent programs?**
- **exploit** parallelism in current computers (multi-processors, multi-cores, hyper-threading)
  - “Free lunch is over”
  - change in Moore’s law ($\times 2$ transistors every 2 years)
- **exploit** several computers (distributed computing)
- **ease of programming** (GUI, network code, reactive programs)
  $\Rightarrow$ found in embedded critical applications (event-driven)
Motivation: concurrent program verification

Concurrent programs are hard to design and hard to verify:

- concurrent programs are highly non-deterministic
  (execution scheduling: many possible interleavings of threads)

- errors appear in corner cases
  (e.g., on a single interleaving, appearing very rarely)
  \[\Rightarrow\] testing is costly and ineffective, with low coverage

- new kinds of errors (data-races, deadlocks)

- new kinds of semantics: weakly consistent memories
  (no more total order of memory operations;
  exhibit behaviours outside of possible interleavings)
Overview

- **State-based semantics**
  - sequential semantics
  - concurrent semantics

- **Modular concurrent semantics**
  - rely/guarantee reasoning
  - rely/guarantee abstractions
  - interference-based denotational analysis

- Advanced topics on interferences
  - weakly consistent memories
  - mutual exclusion primitives

Based on [Miné 12] and [Miné 14]
Sequential semantics
Transition systems

**Formal model of programs** \((\Sigma, \tau, I)\)

- \(\Sigma\): set of program states
- \(\tau \subseteq \Sigma \times \Sigma\): transition relation; \(\langle \sigma, \sigma' \rangle \in \tau\) is noted \(\sigma \rightarrow \sigma'\) (execution step)
- \(I \subseteq \Sigma\): set of initial states
**Sequential semantics**

**Transition systems**

*Formal model of programs* \((\Sigma, \tau, I)\)

- \(\Sigma\): set of program states
- \(\tau \subseteq \Sigma \times \Sigma\): transition relation; \(\langle \sigma, \sigma' \rangle \in \tau\) is noted \(\sigma \rightarrow \sigma'\) (execution step)
- \(I \subseteq \Sigma\): set of initial states

**Example**

\[
\begin{align*}
\ell_1 & \quad i \leftarrow 2 \\
\ell_2 & \quad n \leftarrow \text{input} \ [-100, 100] \\
\ell_3 & \quad \text{while } \ell_4 i \leq n \text{ do} \\
& \hspace{1cm} \text{if random()} \text{ then} \\
& \hspace{2cm} i \leftarrow i + 2 \\
\ell_5 & \quad 
\end{align*}
\]

\(\Sigma = \{1, 2, 3, 4, 5\} \times \mathbb{Z}^2\)

\(I = \{(1, 0, 0)\}\)
Finite trace semantics

**Partial finite execution traces** $\mathcal{T}$
- set of finite traces, in $\mathcal{P}(\Sigma^*)$
- following the transitions in $\tau$
- starting in $\mathcal{I}$
- stopping anywhere (partial or total executions)

$\mathcal{T} \overset{\text{def}}{=} \text{lfp } F$ where

$F(\mathcal{T}) \overset{\text{def}}{=} \mathcal{I} \cup \{ \langle \sigma_0, \ldots, \sigma_{n+1} \rangle \mid \langle \sigma_0, \ldots, \sigma_n \rangle \in \mathcal{T} \land \sigma_n \rightarrow \sigma_{n+1} \}$

**Expressiveness:**
computing $\mathcal{T}$ is equivalent to exhaustive test
$\implies$ can answer question about program safety

**Cost:**
$\mathcal{T}$ is often very large or unbounded
$\implies$ well-defined mathematically but not computable
State semantics

**State semantics $\mathcal{S}$:**
- set of reachable states, in $\mathcal{P}(\Sigma)$
- $\mathcal{S} \overset{\text{def}}{=} \operatorname{lfp} R$ where $R(S) \overset{\text{def}}{=} \mathcal{I} \cup \{ \sigma \mid \exists \sigma' \in S: \sigma' \rightarrow \sigma \}$

**Abstraction** of the trace semantics:
- $\mathcal{S} = \alpha_{\text{state}}(\mathcal{T})$ where
  $$\alpha_{\text{state}}(T) \overset{\text{def}}{=} \{ \sigma_i \mid \exists \langle \sigma_0, \ldots, \sigma_n \rangle \in T: i \in [0, n] \}$$

**Expressiveness:**
- forget the ordering of states in traces:
  $$\alpha_{\text{state}}(\{\bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet\}) = \{\bullet, \bullet, \bullet\}$$
- still sufficient to prove safety properties
  (the program never reaches an error state)
- not expressive enough for liveness properties
  (e.g., termination)
Sequential semantics

Instantiation on a simple language

Language syntax (with explicit locations)

\[
\ell \text{stat} \ ::= \ell X \leftarrow \text{expr} \quad \text{(assignment)} \\
| \ell \text{if } \text{expr} \triangleleft 0 \text{ then } \ell \text{stat} \quad \text{(conditional)} \\
| \ell \text{while } \text{expr} \triangleleft 0 \text{ do } \ell \text{stat} \quad \text{(loop)} \\
| \ell \text{stat}; \ell \text{stat} \quad \text{(sequence)} \\
\]

\[
\text{expr} \ ::= X \mid [c_1, c_2] \mid - \text{expr} \mid \text{expr} \diamond \text{expr}
\]

\[c_1, c_2 \in \mathbb{R} \cup \{+\infty, -\infty\}, \diamond \in \{+, -, \times, /\}, \triangleleft \in \{=, \leq, \ldots\}\]

Idealized language:

- fixed, finite set \(\mathbb{V}\) of numeric variables (with value in \(\mathbb{R}\))
- fixed set of statement locations \(\mathbb{L}\)
- sequential (no concurrency yet)
Semantics

**States:** \( \Sigma \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{E} \)

- control state \( \ell \in \mathcal{L} \) (syntactic location)
- memory state \( \sigma \in \mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \rightarrow \mathbb{R} \) (maps variables to values)

**Expression semantics:** \( E[\text{expr}] : \mathcal{E} \rightarrow \mathcal{P}(\mathbb{R}) \)

- \( E[ [c_1, c_2] ] \rho \overset{\text{def}}{=} \{ \nu \in \mathbb{R} | c_1 \leq \nu \leq c_2 \} \)
- \( E[ X ] \rho \overset{\text{def}}{=} \{ \rho(X) \} \)
- \( E[ -e_1 ] \rho \overset{\text{def}}{=} \{ -\nu | \nu \in E[e_1] \} \)
- \( E[ e_1 \diamond e_2 ] \rho \overset{\text{def}}{=} \{ \nu_1 \diamond \nu_2 | \nu_i \in E[e_i] \rho, \diamond \neq / \lor \nu_2 \neq 0 \} \)

**Command semantics:** \( C[\text{stat}] : \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E}) \)

- \( C[ V \leftarrow e ] R \overset{\text{def}}{=} \{ \rho[V \mapsto \nu] | \rho \in R, \nu \in E[\rho] \} \)
- \( C[ e \bowtie 0 ] R \overset{\text{def}}{=} \{ \rho | \rho \in R, \exists \nu \in E[\rho] : \nu \bowtie 0 \} \)

**Transitions:** if \( \rho' \in C[\ell \text{stat} \ell'] \{\rho\} \) then \( \langle \ell, \rho \rangle \rightarrow \langle \ell', \rho' \rangle \)
State semantic as equation systems

\begin{align*}
\ell_1 & \quad i \leftarrow 2 \\
\ell_2 & \quad n \leftarrow \text{input} \, [-100, 100] \\
\ell_3 & \quad \text{while } \ell_4 \quad i \leq n \text{ do} \\
\ell_5 & \quad \text{if random()} \text{ then} \\
& \quad \quad \quad i \leftarrow i + 2 \ell_6 \\
\ell_7 & \quad \\
\end{align*}

\begin{align*}
\mathcal{X}_1 & = \{ (0, 0) \} \\
\mathcal{X}_2 & = \mathcal{C}[i \leftarrow 2] \mathcal{X}_1 \\
\mathcal{X}_3 & = \mathcal{C}[n \leftarrow [-100, 100]] \mathcal{X}_2 \\
\mathcal{X}_4 & = \mathcal{X}_3 \cup \mathcal{X}_6 \\
\mathcal{X}_5 & = \mathcal{C}[i \leq n] \mathcal{X}_4 \\
\mathcal{X}_6 & = \mathcal{X}_5 \cup \mathcal{C}[i \leftarrow i + 2] \mathcal{X}_5 \\
\mathcal{X}_7 & = \mathcal{C}[i > n] \mathcal{X}_4 \\
\end{align*}

where:

- \( \forall \ell \in \mathcal{L}: \mathcal{X}_\ell \subseteq \mathcal{E} \) (states are partitioned by control location)

- (recursive) equation system stems from the program syntax

- program semantics is the least solution of the system
  (least fixpoint \( \Rightarrow \) most precise invariant)

- it can be solved by increasing iteration:
  \( \forall \ell \in \mathcal{L}: \mathcal{X}_\ell^0 = \emptyset, \quad \forall i > 0: \mathcal{X}_\ell^{i+1} = F_\ell(\mathcal{X}_1^i, \ldots, \mathcal{X}_{|\mathcal{L}|}^i) \)
  (may require transfinite iterations! \( \Rightarrow \) not computable)
Sequential semantics

Static analysis

\[\ell_1 i \leftarrow 2\]

\[\ell_2 n \leftarrow \text{input } [-100, 100]\]

\[\ell_3 \text{while } \ell_4 i \leq n \text{ do}\]

\[\ell_5 \text{if random}() \text{ then }\]

\[i \leftarrow i + 2 \ell_6\]

\[\ell_7\]

\[\chi_1^{\# i+1} \overset{\text{def}}{=} \{ (0, 0) \}\]

\[\chi_2^{\# i+1} \overset{\text{def}}{=} C\left[ i \leftarrow 2 \right] \chi_1^{\# i}\]

\[\chi_3^{\# i+1} \overset{\text{def}}{=} C\left[ n \leftarrow [-100, 100] \right] \chi_2^{\# i}\]

\[\chi_4^{\# i+1} \overset{\text{def}}{=} \chi_4^{\# i} \bigtriangleup (\chi_3^{\# i} \cup \chi_6^{\# i})\]

\[\chi_5^{\# i+1} \overset{\text{def}}{=} C\left[ i \leq n \right] \chi_4^{\# i}\]

\[\chi_6^{\# i+1} \overset{\text{def}}{=} \chi_5^{\# i} \cup C\left[ i \leftarrow i + 2 \right] \chi_5^{\# i}\]

\[\chi_7^{\# i+1} \overset{\text{def}}{=} C\left[ i > n \right] \chi_4^{\# i}\]

- abstract variables \(\chi^\# \ell \in \mathcal{E}^\#\) replace concrete ones \(\chi \ell \in \mathcal{P}(\mathcal{E})\)
- abstract operators are used: \(C\left[ \cdot \right] : \mathcal{E}^\# \to \mathcal{E}^\#, \cup^\# : \mathcal{E}^\# \times \mathcal{E}^\# \to \mathcal{E}^\#\)
- the system is solved by iterations
  \[\chi_{\ell}^{\# 0} \overset{\text{def}}{=} \emptyset^\#, \chi_{\ell}^{\# i+1} \overset{\text{def}}{=} F_{\ell}^\#(\chi_1^{\# i}, \ldots, \chi_{\vert \mathcal{L} \vert}^{\# i})\]
- widening \(\bigtriangleup\) is used to force convergence in finite time
  (e.g.: put unstable bounds to \(\infty\))

\[\implies\text{effective, terminating, sound static analyzer}\]
Sequential semantics

Semantics in denotational form

\( \text{C}[\text{stat}] : \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E}) \)

\( \text{C}[ V \leftarrow e ] R \) \quad \text{def} \quad \{ \rho[V \mapsto v] \mid \rho \in R, \; v \in e[\rho] \} 

\( \text{C}[ e \trianglerightleftarrow 0 ] R \) \quad \text{def} \quad \{ \rho \mid \rho \in R, \; \exists v \in e[\rho] : v \trianglerightleftarrow 0 \} 

\( \text{C}[ \text{if } e \trianglerightleftarrow 0 \text{ then } s ] R \) \quad \text{def} \quad (\text{C}[s] \circ \text{C}[e \trianglerightleftarrow 0?]) R \cup \text{C}[e \trianglerightleftarrow 0?] R 

\( \text{C}[ \text{while } e \trianglerightleftarrow 0 \text{ do } s ] R \) \quad \text{def} \quad \text{C}[e \trianglerightleftarrow 0?] \left( \text{lfp}\lambda X. R \cup (\text{C}[s] \circ \text{C}[e \trianglerightleftarrow 0?]) X \right) 

\( \text{C}[ s_1 ; s_2 ] \) \quad \text{def} \quad \text{C}[s_2] \circ \text{C}[s_1] 

- output memory environments after statement
- structured: based on the syntax of the program
- big-step: forget information on intermediate locations \( \ell \)
  (more efficient in memory)
- can be abstracted in abstract domains, as equation systems
  (add \( \# \) everywhere, add \( \triangledown \) to approximate lfp)
Sequential semantics

Abstraction summary for sequential programs

abstract states \( x^\# \in \mathcal{L} \rightarrow \mathcal{E}^\# \)

\( x^e \)

states \( S \in \mathcal{P}({\Sigma}) \simeq \mathcal{L} \rightarrow \mathcal{P}(\mathcal{E}) \)

\( x^\text{state} \)

finite traces \( T \in \mathcal{P}({\Sigma}^*) \)

Implementing a static analysis

reachability

executions
Concurrent semantics
Language extension:

\[ prog ::= \ell_1^{i_1} \text{stat}_1^{x_1} \mathrel{||} \cdots \mathrel{||} \ell_n^{i_n} \text{stat}_n^{x_n} \]

- finite, fixed set of threads \( \text{stat}_t, t \in \mathcal{T} \) running in parallel
- all variables \( \forall \) are shared

Execution model:

Non-deterministic **interleaving** of thread execution steps
(sequential consistency \cite{Lamport78} with atomic assignments and tests)
Labelled transition system: \((\Sigma, \mathcal{A}, \tau, \mathcal{I})\)

- \(\Sigma\): set of program states
- \(\mathcal{A}\): set of actions
- \(\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma\): transition relation, denoted as \(\sigma \xrightarrow{a} \sigma'\)
- \(\mathcal{I} \subseteq \Sigma\): set of initial states

Labelled traces:
sequences of states interspersed with actions
\[\sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \cdots \sigma_n \xrightarrow{a_n} \sigma_{n+1}\]
Concurrent semantics

Labelled transition system

From programs to labelled transition systems:

- states: $\Sigma \overset{\text{def}}{=} (\mathcal{T} \rightarrow \mathcal{L}) \times \mathcal{E}$
  - thread-local control state in $\mathcal{T} \rightarrow \mathcal{L}$
  - shared memory in $\mathcal{E}$

- initial state: $\mathcal{I} \overset{\text{def}}{=} \{ \langle \lambda t. \ell_t^i, \lambda V.0 \rangle \}$

- actions: thread identifiers $\mathcal{A} \overset{\text{def}}{=} \mathcal{T}$

- transitions: $\forall t \in \mathcal{T}$:
  $\langle L[t \mapsto \ell], \rho \rangle \overset{t}{\rightarrow} \langle L[t \mapsto \ell'], \rho' \rangle \iff \langle \ell, \rho \rangle \overset{\text{stat}_t}{\rightarrow} \langle \ell', \rho' \rangle$
  - derived from the transitions of individual threads
  - update the control state $L(t)$ of a single thread $t$
  - update the memory state $\rho$

ECNU, 29/11/2013
Analysis of concurrent programs
Antoine Miné
p. 18 / 61
Trace and state semantics

**Labelled trace semantics:**

- set of interleaved execution traces, with thread labels
- $$\mathcal{T} \overset{\text{def}}{=} \text{lfp } F$$ where
  $$F(\mathcal{T}) \overset{\text{def}}{=} \mathcal{I} \cup \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_i} \sigma_{i+1} \mid \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{i-1}} \sigma_i \in \mathcal{T} \land \sigma_i \xrightarrow{t_i} \sigma_{i+1} \}$$

**State semantics:** (as before)

- $$\mathcal{S} \overset{\text{def}}{=} \text{lfp } R$$ where $$R(\mathcal{S}) \overset{\text{def}}{=} \mathcal{I} \cup \{ \sigma \mid \exists \sigma', t: \sigma' \xrightarrow{t} \sigma \}$$
- $$\mathcal{S} = \alpha_{\text{state}}(\mathcal{T})$$ where
  $$\alpha_{\text{state}}(\mathcal{T}) \overset{\text{def}}{=} \{ \sigma_i \mid \exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in \mathcal{T} : i \in [0, n] \}$$

**Idea:** (will not scale up)

forget about threads and labels
analyze as a **sequential program** interleaving thread statements
Concurrent semantics

Equational state semantic example

Example: inferring $0 \leq x \leq y \leq 10$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $\ell_1$ true do $\ell_2$ if $x &lt; y$ then $\ell_3$ $x \leftarrow x + 1$</td>
<td>while $\ell_4$ true do $\ell_5$ if $y &lt; 10$ then $\ell_6$ $y \leftarrow y + 1$</td>
</tr>
</tbody>
</table>

- attach variables $\mathcal{X}_L \in \mathcal{P}(\mathcal{E})$ to control locations $L \in \mathcal{T} \rightarrow \mathcal{L}$
- synthesize equations $\mathcal{X}_L = F_L(\mathcal{X}_{(1,\ldots,1)}, \ldots, \mathcal{X}_{(|\mathcal{L}|,\ldots,|\mathcal{L}|)}$ from thread equations $\mathcal{X}_{\ell,t} = F_{\ell,t}(\mathcal{X}_1,t, \ldots, \mathcal{X}_{|\mathcal{L}|,t})$
## Equational state semantic example

Example: inferring $0 \leq x \leq y \leq 10$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>while</strong> $\ell_1$ true do</td>
<td><strong>while</strong> $\ell_4$ true do</td>
</tr>
<tr>
<td>$\ell_2$ if $x &lt; y$ then</td>
<td>$\ell_5$ if $y &lt; 10$ then</td>
</tr>
<tr>
<td>$\ell_3$ $x \leftarrow x + 1$</td>
<td>$\ell_6$ $y \leftarrow y + 1$</td>
</tr>
</tbody>
</table>

(Simplified) concrete equation system:

\[
\begin{align*}
\mathcal{X}_{1,4} &= \mathcal{I} \cup \mathcal{C}[x \leftarrow x + 1] \mathcal{X}_{3,4} \cup \mathcal{C}[x \geq y] \mathcal{X}_{2,4} \\
& \quad \cup \mathcal{C}[y \leftarrow y + 1] \mathcal{X}_{1,6} \cup \mathcal{C}[y \geq 10] \mathcal{X}_{1,5} \\
\mathcal{X}_{2,4} &= \mathcal{X}_{1,4} \cup \mathcal{C}[y \leftarrow y + 1] \mathcal{X}_{2,6} \cup \mathcal{C}[y \geq 10] \mathcal{X}_{2,5} \\
\mathcal{X}_{3,4} &= \mathcal{C}[x < y] \mathcal{X}_{2,4} \cup \mathcal{C}[y \leftarrow y + 1] \mathcal{X}_{3,6} \cup \mathcal{C}[y \geq 10] \mathcal{X}_{3,5} \\
\mathcal{X}_{1,5} &= \mathcal{C}[x \leftarrow x + 1] \mathcal{X}_{3,5} \cup \mathcal{C}[x \geq y] \mathcal{X}_{2,5} \cup \mathcal{X}_{1,4} \\
\mathcal{X}_{2,5} &= \mathcal{X}_{1,5} \cup \mathcal{X}_{2,4} \\
\mathcal{X}_{3,5} &= \mathcal{C}[x < y] \mathcal{X}_{2,5} \cup \mathcal{X}_{3,4} \\
\mathcal{X}_{1,6} &= \mathcal{C}[x \leftarrow x + 1] \mathcal{X}_{3,6} \cup \mathcal{C}[x \geq y] \mathcal{X}_{2,6} \cup \mathcal{C}[y < 10] \mathcal{X}_{1,5} \\
\mathcal{X}_{2,6} &= \mathcal{X}_{1,6} \cup \mathcal{C}[y < 10] \mathcal{X}_{2,5} \\
\mathcal{X}_{3,6} &= \mathcal{C}[x < y] \mathcal{X}_{2,6} \cup \mathcal{C}[y < 10] \mathcal{X}_{3,5}
\end{align*}
\]
Equational state semantic example

Example: inferring $0 \leq x \leq y \leq 10$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $\ell_1$ true do</td>
<td></td>
</tr>
<tr>
<td>$\ell_2$ if $x &lt; y$ then</td>
<td></td>
</tr>
<tr>
<td>$\ell_3$ $x \leftarrow x + 1$</td>
<td></td>
</tr>
<tr>
<td>while $\ell_4$ true do</td>
<td></td>
</tr>
<tr>
<td>$\ell_5$ if $y &lt; 10$ then</td>
<td></td>
</tr>
<tr>
<td>$\ell_6$ $y \leftarrow y + 1$</td>
<td></td>
</tr>
</tbody>
</table>

**Pros:**
- easy to construct
- easy to further abstract in an abstract domain $\mathcal{E}^\#$

**Cons:**
- explosion of the number of variables and equations
- explosion of the size of equations (efficiency issues)
- the equation system does *not* reflect the program structure (not defined by induction on the concurrent program)
Wish-list

We would like to:

- keep information attached to syntactic program locations (control points in $L$, not control point tuples in $T \rightarrow L$)
- be able to abstract away control information (precision/cost trade-off control)
- avoid duplicating thread instructions
- have a computation structure based on the program syntax (denotational style)

Ideally:

thread-modular denotational-style semantics
(analyze each thread independently by induction on its syntax)
Rely/guarantee proof method
Rely/guarantee proof method

Floyd–Hoare logic

Logic to prove properties about **sequential** programs [Hoare 69].

**Hoare triples:** \{P\} stat \{Q\}

- annotate programs with logic assertions \{P\} stat \{Q\}
  (if \(P\) holds before \textit{stat}, then \(Q\) holds after \textit{stat})
- check that \{P\} stat \{Q\} is derivable with the following rules

\[
\begin{align*}
\{P[e/x]\} X & \leftarrow e \{P\} \\
\{P\} s_1 \{Q\} & \quad \{Q\} s_2 \{R\} \\
\{P\} s_1; s_2 \{R\} & \\
\{P\} s \{Q\} & \\
\{P\} \text{ if } e \not\prec 0 \text{ then } s \{Q\} & \\
\{P\} \text{ while } e \not\triangleleft 0 \text{ do } s \{P \land e \not\triangleleft 0\} & \\
\{P\} s \{Q\}' & \quad P \implies P' \quad Q' \implies Q \\
\{P\} s \{Q\} &
\end{align*}
\]
Rely/guarantee proof method

Floyd–Hoare logic

Logic to prove properties about **sequential** programs \([\text{Hoare 69}].\)

**Hoare triples:** \(\{P\} \text{ stat } \{Q\}\)

- annotate programs with **logic assertions** \(\{P\} \text{ stat } \{Q\}\)
  - (if \(P\) holds before \(\text{stat}\), then \(Q\) holds after \(\text{stat}\))
- check that \(\{P\} \text{ stat } \{Q\}\) is derivable with the following rules

\[
\frac{\{P[e/x]\} X \leftarrow e \{P\}}{
\{P\} s_1 \{Q\} \quad \{Q\} s_2 \{R\}
}
\]

\[
\frac{\{P\} \{P \land e \not\in 0\} s \{Q\} \quad P \land e \not\in 0 \Rightarrow Q}{\{P\} \text{ if } e \not\in 0 \text{ then } s \{Q\}}
\]

\[
\frac{\{P\} \text{ while } e \not\in 0 \text{ do } s \{P \land e \not\in 0\}}{
\{P\} \{P \land e \not\in 0\} s \{P\}
}
\]

\[
\frac{\{P\} s \{Q\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q}{\{P\}' s \{Q'\}}
\]

**Link with abstract interpretation:** \([\text{Cousot 84}]\)

- inferred program invariants are valid assertions
- the concrete reachable states \(\mathcal{S}\) are the optimal assertions
Jones’ rely-guarantee proof method

[Jones 81]: extension of Hoare logic to parallel programs

**Uses quintuple:** \[ R, G \vdash \{ P \} \text{stat} \{ Q \} \]

- if \( P \) is true before \( \text{stat} \) is executed
- and the effect of other threads is included in \( R \)  \((R : \text{rely})\)
- then \( Q \) is true after \( \text{stat} \)
- and the effect of \( \text{stat} \) is included in \( G \)  \((G : \text{guarantee})\)

where:

- \( P \) and \( Q \) are assertions on states  \((\text{in } \mathcal{P}(\Sigma))\)
- \( R \) and \( G \) are assertions on transitions  \((\text{in } \mathcal{P}(\Sigma \times A \times \Sigma))\)

Hoare rules are unchanged, we add the parallel composition rule:

\[
R \lor G_2, G_1 \vdash \{ P_1 \} s_1 \{ Q_1 \} \quad R \lor G_1, G_2 \vdash \{ P_2 \} s_2 \{ Q_2 \}
\]

\[
R, G_1 \lor G_2 \vdash \{ P_1 \land P_2 \} s_1 \parallel s_2 \{ Q_1 \land Q_2 \}
\]
Rely-guarantee example

Example: proving $0 \leq x \leq y \leq 10$

checking $t_1$

\begin{align*}
\ell_1 \quad & \text{while } 0 = 0 \text{ do} \\
\ell_2 \quad & \text{if } x < y \text{ then} \\
\ell_3 \quad & x \leftarrow x + 1
\end{align*}

at $\ell_1, \ell_2: 0 \leq x \leq y \leq 10$

at $\ell_3: 0 \leq x < y \leq 10$

checking $t_2$

\begin{align*}
\ell_4 \quad & \text{while } 0 = 0 \text{ do} \\
\ell_5 \quad & \text{if } y < 10 \text{ then} \\
\ell_6 \quad & y \leftarrow y + 1
\end{align*}

at $\ell_4, \ell_5: 0 \leq x \leq y \leq 10$

at $\ell_6: 0 \leq x \leq y < 10$

In this example: guarantee exactly what is relied on ($R_1 = G_1$ and $R_2 = G_2$) rely and guarantee are global assertions Benefits of rely-guarantee: compact and modular invariants are still local to threads checking a thread does not require looking at other threads, only at an abstraction of their semantics
Rely-guarantee example

Example: proving $0 \leq x \leq y \leq 10$

**Checking $t_1$**

- $\ell_1$ **while** $0 = 0$ **do**
  - $\ell_2$ **if** $x < y$ **then**
    - $\ell_3$ $x \leftarrow x + 1$
  - $x$ unchanged
  - $y$ incremented
- $y \leq 10$

At $\ell_1, \ell_2: 0 \leq x \leq y \leq 10$
At $\ell_3: 0 \leq x < y \leq 10$

**Checking $t_2$**

- $y$ unchanged
- $x \leq y$

At $\ell_4, \ell_5: 0 \leq x \leq y \leq 10$
At $\ell_6: 0 \leq x \leq y < 10$

In this example:

- guarantee exactly what is relied on ($R_1 = G_1$ and $R_2 = G_2$)
- rely and guarantee are global assertions

**Benefits of rely-guarantee:** compact and modular

- invariants are still local to threads
- checking a thread does not require looking at other threads, only at an abstraction of their semantics
Goal: prove \( \{ x = 0 \} \, t_1 \| \, t_2 \, \{ x = 2 \} \).
**Auxiliary variables**

**Example**

<table>
<thead>
<tr>
<th>(t_1)</th>
<th>(t_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ell_1) (x \leftarrow x + 1) (\ell_2)</td>
<td>(\ell_3) (x \leftarrow x + 1) (\ell_4)</td>
</tr>
</tbody>
</table>

**Goal:** prove \(\{x = 0\}\) \(t_1 \parallel t_2 \{x = 2\}\).

we must rely on and guarantee that each thread increments \(x\) exactly once!

**Solution:** auxiliary variables do not change the semantics but store extra information: e.g., program counter of other threads \((pc_t)\)

**Example:** for \(t_1\): \(\{(pc_2 = \ell_3 \land x = 0) \lor (pc_2 = \ell_4 \land x = 1)\}\)

\[x \leftarrow x + 1\]

\(\{(pc_2 = \ell_3 \land x = 1) \lor (pc_2 = \ell_4 \land x = 2)\}\)
Rely-guarantee as abstract interpretation
Goal

Formalization as abstract interpretation: [Miné 2014]

- constructive design
- infer invariants and guarantees (instead of only checking)
- exploit existing abstractions (numeric domains)
- keep modularity
Local invariants

**State projection** $\mathcal{S}\ell(t)$: on a thread $t \in T$

- add auxiliary variables $\forall_t \overset{\text{def}}{=} \forall \cup \{ pc_u \mid u \in T, u \neq t \}$
- enriched environments for $t$: $\mathcal{E}_t \overset{\text{def}}{=} \forall_t \rightarrow \mathbb{R}$
  (for simplicity, $pc_u$ are numeric variables, i.e., $\mathcal{L} \subseteq \mathbb{R}$)
- local states: $\Sigma_t \overset{\text{def}}{=} \mathcal{L} \times \mathcal{E}_t$
  (recall that $\Sigma \overset{\text{def}}{=} (T \rightarrow \mathcal{L}) \times \mathcal{E}$)
- projection: $\pi_t \langle \mathcal{L}, \rho \rangle \overset{\text{def}}{=} \langle \mathcal{L}(t), \rho[\forall u \neq t: pc_u \mapsto \mathcal{L}(u)] \rangle$
  extended naturally to $\pi_t : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma_t)$

Local invariants on $t$: $\mathcal{S}\ell(t) \overset{\text{def}}{=} \pi_t(\mathcal{S})$

(where $\mathcal{S}$ is the reachable state abstraction)

Note: $\pi_t$ is a bijection, no information is lost
Interferences $A(t)$: caused by a thread $t \in T$

$$A \in T \rightarrow \mathcal{P}(\Sigma \times \Sigma)$$

$$A(t) \overset{\text{def}}{=} \alpha^{itf}(\top)(t)$$

where $\alpha^{itf}(X)(t) \overset{\text{def}}{=} \{ \langle \sigma, \sigma' \rangle \mid \exists \cdots \sigma \xrightarrow{t} \sigma' \cdots \in X \}$

subset of the transition system $\tau$

- spawned by $t$ and
- actually observed in some execution trace
  (recall that $\top$ is the prefix trace abstraction)
**Fixpoint form**

**Goal:** express $S\ell$

- constructively (fixpoint)
- in a modular way (without computing $S$)

**Local state fixpoint:**

- we express $S\ell(t)$ as a function of $A$ and thread $t \in \mathcal{T}$:

$$S\ell(t) = \text{lfp } R_t(A)$$ where

$$R_t(Y)(X) \overset{\text{def}}{=} \pi_t(I) \cup$$

$$\{ \pi_t(\sigma') | \exists \pi_t(\sigma) \in X: \sigma \xrightarrow{t} \tau \sigma' \lor \exists u \neq t: (\sigma, \sigma') \in Y(u) \}$$

A state is reachable if it is initial, or reachable by transitions from $t$ or from the environment $A$.

$R_t$ only looks into the syntax of thread $t$.

$R_t$ is parameterized by the interferences from other threads $Y$. 
Interferences:

we express $A(t)$ as a function of $\mathcal{S}_\ell$ and thread $t \in \mathcal{T}$:

$$A(t) = B(\mathcal{S}_\ell)(t)$$

where

$$B(Z)(t) \overset{\text{def}}{=} \{ (\sigma, \sigma') \mid \pi_t(\sigma) \in Z(t) \land \sigma \xrightarrow{t} \sigma' \}$$

Collect transitions starting from reachable states.

No fixpoint needed.
Nested fixpoint characterization:

1. recall: $S \ell(t) = \text{lfp } R_t(A)$
2. recall: $A(t) = B(S \ell)(t)$
3. $\implies$ mutual dependency between $S \ell$ and $A$
Nested fixpoint characterization:

1. recall: $S_\ell(t) = \text{lfp } R_t(A)$

2. recall: $A(t) = B(S_\ell)(t)$

3. \(\Rightarrow\) mutual dependency between $S_\ell$ and $A$

   \(\Rightarrow\) solved using a fixpoint:

   $S_\ell = \text{lfp } H$ where

   $H(Z)(t) \overset{\text{def}}{=} \text{lfp } R_t(B(Z))$
Constructive fixpoint form:

Use Kleene’s iteration to construct fixpoints:

- $\ell = \text{Ifp } H = \bigcup_{n \in \mathbb{N}} H^n(\lambda t.\emptyset)$
  in the pointwise powerset lattice $\prod_{t \in T} \{t\} \rightarrow \mathcal{P}(\Sigma_t)$

- $H(Z)(t) = \text{Ifp } R_t(B(Z)) = \bigcup_{n \in \mathbb{N}} (R_t(B(Z)))^n(\emptyset)$
  in the powerset lattice $\mathcal{P}(\Sigma_t)$
  (similar to the sequential semantics of thread $t$ in isolation)

nested fixpoints $\implies$ nested iterations
Abstract rely-guarantee

**Algorithm:** nested abstract iterations with acceleration \( \nabla \)

Once abstract domains for states and interferences are chosen

- Start from \( S^{\ell}_0 \stackrel{\text{def}}{=} A_0 \stackrel{\text{def}}{=} \lambda t. \bot \)

- While \( A_n \) is not stable
  - Compute \( \forall t \in T: S^{\ell}_{n+1}(t) \stackrel{\text{def}}{=} \operatorname{lfp} R_t^\#(A_n) \)
    - By iteration with widening \( \nabla \)
    - \( (\simeq \text{separate analysis of each thread}) \)
  - Compute \( A_{n+1} \stackrel{\text{def}}{=} A_n \nabla B^\#(S^{\ell}_{n+1}) \)
  - When \( A_n = A_{n+1} \), return \( S^{\ell}_{n} \)

\[ \implies \text{thread-modular analysis} \]
\[ \text{parameterized by abstract domains} \]
\[ \text{able to easily reuse existing sequential analyses} \]
Flow-insensitive abstraction

**Idea:**
- reduce as much control information as possible
- but keep flow-sensitivity on each thread’s control location

**Local state abstraction:** remove auxiliary variables

\[
\alpha^\text{nf}_S : \mathcal{P}(\Sigma_t) \to \mathcal{P}(\mathcal{L} \times \mathcal{E})
\]

\[
\alpha^\text{nf}_S (X) \overset{\text{def}}{=} \{ (\ell, \rho|_\mathcal{V}) \mid (\ell, \rho) \in X \}
\]

**Interference abstraction:** remove all control state

\[
\alpha^\text{nf}_A : \mathcal{P} (\Sigma \times \Sigma) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})
\]

\[
\alpha^\text{nf}_A (Y) \overset{\text{def}}{=} \{ (\rho, \rho') \mid \exists L, L' \in \mathcal{T} \to \mathcal{L} : ((L, \rho), (L', \rho')) \in Y \} 
\]
Rely-guarantee as abstract interpretation

Flow-insensitive abstraction (cont.)

Flow-insensitive fixpoint semantics:

We apply $\alpha_{S}^{nf}$ and $\alpha_{A}^{nf}$ to the operators $R_{t}$ and $B$:

$R_{t}^{nf}(Y)(X) \overset{\text{def}}{=} R_{t}^{loc}(X) \cup A_{t}^{nf}(Y)(X)$ where

$R_{t}^{loc}(X) \overset{\text{def}}{=} \{(\ell_{t}^{i}, \lambda V.0)\} \cup \{(\ell', \rho') | \exists (\ell, \rho) \in X: (\ell, \rho) \rightarrow_{t} (\ell', \rho')\}$

$A_{t}^{nf}(Y)(X) \overset{\text{def}}{=} \{(\ell, \rho') | \exists \rho, u \neq t: (\ell, \rho) \in X \land (\rho, \rho') \in Y(u)\}$

$B^{nf}(Z)(t) \overset{\text{def}}{=} \{(\rho, \rho') | \exists \ell, \ell' \in \mathcal{L}: (\ell, \rho) \in Z(t) \land (\ell, \rho) \rightarrow_{t} (\ell', \rho')\}$

where $\rightarrow_{t}$ is the transition relation for thread $t$ alone

Cost/precision trade-off:

- less variables
  \implies subsequent numeric abstractions are more efficient
- sufficient to analyze our first example (p. 20)
- insufficient to analyze $x \leftarrow x + 1 \parallel x \leftarrow x + 1$
Non-relational interference abstraction

Idea: simplify further flow-insensitive interferences

- numeric relations are more costly than numeric sets
  $\Rightarrow$ remove input sensitivity
- relational domains are more costly than non-relational
  $\Rightarrow$ abstract the interference on each variable separately

Non-relational interference abstraction:

$$\alpha_{nr}^A : \mathcal{P}(\mathcal{E} \times \mathcal{E}) \to (\forall \to \mathcal{P}(\mathbb{R}))$$

$$\alpha_{nr}^A (Y) \overset{\text{def}}{=} \lambda V. \{ x \in V | \exists (\rho, \rho') \in Y : \rho(V) \neq x \land \rho'(V) = x \}$$

(remember which variables are modified and their new values)

We get, in the abstract reachability computation:

$$R_{nr}^t (Y)(X) \overset{\text{def}}{=} R_{loc}^t (X) \cup A_{nr}^t (Y)(X)$$

$$A_{nr}^t (Y)(X) \overset{\text{def}}{=} \{ (\ell, \rho[V \mapsto v]) | (\ell, \rho) \in X, V \in V, \exists u \neq t : v \in Y(u)(V) \}$$
A note on unbounded threads

**Extension:** relax the finiteness constraint on $\mathcal{T}$

- there is still a finite syntactic set of threads $\mathcal{T}_s$
- some threads $\mathcal{T}_\infty \subseteq \mathcal{T}_s$ can have several instances
  (possibly an unbounded number)

**Flow-insensitive analysis:**

- local state and interference domains have finite dimensions
  $(\mathcal{E}_t$ and $(\mathcal{L} \times \mathcal{E}) \times (\mathcal{L} \times \mathcal{E})$, as opposed to $\mathcal{E}$ and $\mathcal{E} \times \mathcal{E}$)
- all instances of a thread $t \in \mathcal{T}_s$ are isomorphic
  $\implies$ iterate the analysis on the finite set $\mathcal{T}_s$ (instead of $\mathcal{T}$)
- we must handle self-interferences for threads in $\mathcal{T}_\infty$:

  $$A_t^{nf}(Y)(X) \overset{\text{def}}{=} \{ (\ell, \rho') \mid \exists \rho, \ u : (u \neq t \lor t \in \mathcal{T}_\infty) \land (\ell, \rho) \in X \land (\rho, \rho') \in Y(u) \}$$
Abstraction summary for thread-modular analyses

Abstract states

\[(T \times L) \rightarrow E^\#\]

\[\alpha^E\]

Local states

\[(T \times L) \rightarrow \mathcal{P}(E)\]

\[\alpha^{nf}_{\mathcal{L}}\]

Interleaved finite traces

\[\mathcal{P}(\Sigma^*)\]

Abstract interferences

\[T \rightarrow E^\#\]

Non-relational interferences

\[T \rightarrow \mathcal{P}(E)\]

\[\alpha^E\]

\[\alpha^{nr}_{\mathcal{L}}\]

Flow-insensitive interferences

\[T \rightarrow \mathcal{P}(E \times E)\]

\[\alpha^{nf}_{\mathcal{L}}\]

\[\alpha^{nf}_{\mathcal{A}}\]

Interferences

\[A : T \rightarrow \mathcal{P}(\Sigma \times \Sigma)\]

\[\alpha^{itf}\]

Rely-guarantee

(rely-guarantee (without aux. variables))

\[\pi_t\]

Rely-guarantee

(rely-guarantee (with aux. variables))

Static analyzer

Interleaved executions
Rely-guarantee as abstract interpretation

Compare with sequential analyses...

Abstract states

\[ \mathcal{L} \rightarrow \mathcal{E}^\# \]

States

\[ S \in \mathcal{P}(\Sigma) \]

Finite traces

\[ T \in \mathcal{P}(\Sigma^*) \]

Static analyzer

Reachability

Executions

ECNU, 29/11/2013

Analysis of concurrent programs

Antoine Miné

p. 41 / 61
Interference-based denotational analysis
Interferences in $I \overset{\text{def}}{=} T \times V \times R$

$\langle t, X, v \rangle$ means: $t$ can store the value $v$ into the variable $X$

We define the analysis of a thread $t$ with respect to a set of interferences $I \subseteq I$.

**Expressions with interference:** for thread $t$

$E_t[ \text{expr} ] : (E \times \mathcal{P}(I)) \rightarrow \mathcal{P}(R)$

- Apply interferences to read variables:
  $$E_t[ X ] \langle \rho, I \rangle \overset{\text{def}}{=} \{ \rho(X) \} \cup \{ v | \exists u \neq t: \langle u, X, v \rangle \in I \}$$

- Pass recursively $I$ down to sub-expressions:
  $$E_t[ -e_1 ] \langle \rho, I \rangle \overset{\text{def}}{=} \{ -v | v \in E_t[ e_1 ] \langle \rho, I \rangle \}$$

\[ \ldots \]
**Statements with interference:** for thread $t$

$$C_t[\text{stat}] : (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(I)) \rightarrow (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(I))$$

- Pass interferences to expressions
- Collect new interferences due to assignments
- Accumulate interferences from inner statements

$$C_t[ X \leftarrow e ] \langle R, I \rangle \overset{\text{def}}{=} \langle \emptyset, I \rangle \sqcup \bigsqcup_{\rho \in R} \langle \{ \rho[X \mapsto v] \mid v \in E_t[e] \langle \rho, I \rangle \}, \{ \langle t, X, v \rangle \mid v \in V_\rho \} \rangle$$

$$C_t[s_1; s_2] \overset{\text{def}}{=} C_t[s_2] \circ C_t[s_1]$$

$$\ldots$$

$$(\sqcup \text{ is now the element-wise } \cup \text{ in } \mathcal{P}(\mathcal{E}) \times \mathcal{P}(I))$$
Denotational semantics with interferences (cont.)

Program semantics:

Given \( \text{prog} ::= \text{stat}_1 \parallel \cdots \parallel \text{stat}_n \) we:

1. Compute the interference fixpoint:
   \[ A \overset{\text{def}}{=} \text{lfp} \lambda I. \bigcup_{t \in T} [C_t[\text{stat}_t] \langle E_0, I \rangle] \]

   (analyze threads, keep only the interference outputs, join them, iterate until stabilization)

2. Then compute the environments output by each thread:
   \[ \lambda t. [C_t[\text{stat}_t] \langle E_0, A \rangle]_{P(E)} \]
   (analyze each thread once on the stable interferences, keep only the output environments)

   where the initial environments are \( E_0 \overset{\text{def}}{=} \{ \lambda \text{V}.0 \} \)
**Concrete interference semantics:**

iteration 1

\[ I = \emptyset \]

\( \ell_1 : x = 0, y = 0 \)

\( \ell_4 : x = 0, y \in [0, 10] \)

new \( I = \{ \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \} \)
### Example

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>while</strong> $\ell_1$ $0 = 0$ <strong>do</strong></td>
<td><strong>while</strong> $\ell_4$ $0 = 0$ <strong>do</strong></td>
</tr>
<tr>
<td>$\ell_2$ <strong>if</strong> $x &lt; y$ <strong>then</strong></td>
<td>$\ell_5$ <strong>if</strong> $y &lt; 10$ <strong>then</strong></td>
</tr>
<tr>
<td>$\ell_3$ $x \leftarrow x + 1$</td>
<td>$\ell_6$ $y \leftarrow y + 1$</td>
</tr>
</tbody>
</table>

---

**Concrete interference semantics:**

Iteration 2

$l = \{ \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \}$

$\ell_1 : x \in [0, 10], y = 0$

$\ell_4 : x = 0, y \in [0, 10]$

New $l = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \}$
### Concrete interference semantics:

**iteration 3**

\[ l = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \} \]

- \( \ell_1 : x \in [0, 10], y = 0 \)
- \( \ell_4 : x = 0, y \in [0, 10] \)
- new \( l = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \} \)
Concrete interference semantics:

iteration 3

\[ I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \} \]

\[ \ell_1 : x \in [0, 10], \ y = 0 \]

\[ \ell_4 : x = 0, \ y \in [0, 10] \]

new \[ I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \} \]

Note: we don’t get that \[ x \leq y \] at \[ \ell_1 \], only that \[ x, y \in [0, 10] \]
Interference abstraction

**Abstract interferences $\mathcal{I}^\#$**

\[ \mathcal{P}(\emptyset) \overset{\text{def}}{=} \mathcal{P}({\mathcal{T} \times \mathcal{V} \times \mathcal{R}}) \text{ is abstracted as } \mathcal{I}^\# \overset{\text{def}}{=} (\mathcal{T} \times \mathcal{V}) \rightarrow \mathcal{R}^\# \]

where $\mathcal{R}^\#$ abstracts $\mathcal{P}(\mathcal{R})$ (e.g. intervals)

**Abstract semantics with interferences $C_t^[[s]]$**

derived from the non-parallel semantics $C^[[s]]$ in a generic way:

**Example:**

\[ C_t^[[X \leftarrow e]](R^\#, \mathcal{I}^\#) \]

- for each $Y$ in $e$, get its interference $Y^\#_\mathcal{R} = \bigsqcup_{\mathcal{R}} \{ \mathcal{I}^\#(u, Y) \mid u \neq t \}$
- if $Y^\#_\mathcal{R} \neq \bot^\#_\mathcal{R}$, replace $Y$ in $e$ with $\text{get}(Y, R^\#) \sqcup^\#_{\mathcal{R}} Y^\#_\mathcal{R}$
  (where $\text{get}(Y, R^\#)$ extracts the abstract values in $\mathcal{R}^\#$ of a variable $Y$ from $R^\# \in \mathcal{E}^\#$)
- compute $R^{\#'} = C^[[e]] R^\#$
- enrich $\mathcal{I}^\#(t, X)$ with $\text{get}(X, R^{\#'})$
Abstract interference analysis:

\[ A^\# \overset{\text{def}}{=} \lim \lambda I^\# . I^\# \blacktriangledown \bigcup_{t \in T} \left[ C_t [\text{stat}_t] \langle E_0^\#, I^\# \rangle \right] \# \]

- effective analysis by structural induction
- termination ensured by a widening \( \blacktriangledown \)
- parametrized by a choice of abstract domains \( R^\#, E^\# \)

- interferences are flow-insensitive and non-relational in \( R^\# \)
- thread analysis remains flow-sensitive and relational in \( E^\# \)

See [Miné 12]
Advanced interferences
Issues with weak consistency

The program written:

\[
\begin{align*}
F_1 & \leftarrow 1; \\
\text{if } F_2 = 0 \text{ then } & S_1 \\
F_2 & \leftarrow 1; \\
\text{if } F_1 = 0 \text{ then } & S_2
\end{align*}
\]

(simplified Dekker mutual exclusion algorithm)

\(S_1\) and \(S_2\) cannot execute simultaneously.
Issues with weak consistency

(simplified Dekker mutual exclusion algorithm)

\[ F_1 \leftarrow 1; \]
\[ \text{if } F_2 = 0 \text{ then } \]
\[ S_1 \]
\[ F_2 \leftarrow 1; \]
\[ \text{if } F_1 = 0 \text{ then } \]
\[ S_2 \]
\[ \rightarrow \]
\[ \text{if } F_2 = 0 \text{ then } \]
\[ F_1 \leftarrow 1; \]
\[ S_1 \]
\[ \text{if } F_1 = 0 \text{ then } \]
\[ F_2 \leftarrow 1; \]
\[ S_2 \]

\( S_1 \) and \( S_2 \) can execute simultaneously.
Not a sequentially consistent behavior!

Caused by:
- write FIFOs, caches, distributed memory
- hardware or compiler optimizations, transformations
- ...

behavior accepted by Java [Manson 05]
Atomicity and granularity

We assumed that assignments are atomic...
Atomicity and granularity

We assumed that assignments are atomic... but that may not be the case.

The second program admits more behaviors e.g.: $X = 1$ at the end of the program

[Reynolds 04]
Soundness theorem: [Miné 2012]

For flow-insensitive interference abstractions, the analysis is invariant by a wide range of thread transformations:

- inserting FIFO buffers (TSO hardware model)
- reordering of “independent” statements
- common sub-expression elimination
- change of granularity (non-atomic expression evaluation)
- ...

ECNU, 29/11/2013
Scheduling

Synchronization primitives

\[
\text{stat} ::= \text{lock}(m) \mid \text{unlock}(m)
\]

\( m \in M \): finite set of non-recursive mutexes

Scheduling

- Mutexes ensure mutual exclusion
  - At each instant, each mutex can be locked by a single thread

- Mutexes enforce memory consistency and atomicity
  - No optimization across `lock` and `unlock` instructions
  - Memory caches and buffer are flushed
Advanced interferences

Mutual exclusion

No interference unless:
- write / read not protected by a common mutex (data-races), or
- last write before unlocking affects first read after lock

Solution:
- partition interferences wrt. mutexes;
- $I_{def} = T \times V \times R$ becomes $I_{def} = T \times P(M) \times V \times R$
- extract / apply interferences at critical section boundaries

ECNU, 29/11/2013
No interference unless:

- write / read not protected by a common mutex (data-races), or
Mutual exclusion

No interference unless:
- write / read not protected by a common mutex (data-races), or
- last write before unlocking affects first read after lock
Mutual exclusion

No interference unless:
- write / read not protected by a common mutex (data-races), or
- last write before unlocking affects first read after lock

Solution:
- partition interferences wrt. mutexes;
- \[ I \overset{\text{def}}{=} T \times \mathbb{V} \times \mathbb{R} \text{ becomes } I \overset{\text{def}}{=} T \times \mathcal{P}(M) \times \mathbb{V} \times \mathbb{R} \]
- extract / apply interferences at critical section boundaries
Example analysis

Abstract consumer/producer

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{while} 0=0 do</td>
<td>\textbf{while} 0=0 do</td>
</tr>
<tr>
<td>$\texttt{lock}(m)$; $\ell^1$</td>
<td>$\texttt{lock}(m)$;</td>
</tr>
<tr>
<td>\textbf{if} $X &gt; 0$ \textbf{then} $\ell^2 X \leftarrow X - 1$;</td>
<td>$X \leftarrow X + 1$;</td>
</tr>
<tr>
<td>$\texttt{unlock}(m)$;</td>
<td>\textbf{if} $X &gt; 10$ \textbf{then} $X \leftarrow 10$;</td>
</tr>
<tr>
<td>$\ell^3 Y \leftarrow X$</td>
<td>$\texttt{unlock}(m)$</td>
</tr>
</tbody>
</table>

- at $\ell^1$, the \texttt{unlock} − \texttt{lock} effect from $t_2$ imports $\{X\} \times [1, 10]$
- at $\ell^2$, $X \in [1, 10]$, no effect from $t_2$: $X \leftarrow X - 1$ is safe
- at $\ell^3$, $X \in [0, 9]$, and $t_2$ has the effects $\{X\} \times [1, 10]$
  so, $Y \in [0, 10]$
Relational lock invariants

example

```
while true do
  lock(m);
  if X > 0 then
    X ← X − 1;
    Y ← Y − 1;
  unlock(m)
```

```
while true do
  lock(m);
  if X < 10 then
    X ← X + 1;
    Y ← Y + 1;
  unlock(m)
```

Non-relational interferences find $X \in [0, 10]$, but no bound on $Y$
Actually, $Y \in [0, 10]$
### Relational lock invariants

**Example**

```plaintext
code
while true do
    lock(m);
    if X > 0 then
        X ← X − 1;
        Y ← Y − 1;
    end if
   _unlock(m)
end while
```

```plaintext
code
while true do
    lock(m);
    if X < 10 then
        X ← X + 1;
        Y ← Y + 1;
    end if
    _unlock(m)
end while
```

Non-relational interferences find $X \in [0, 10]$, but no bound on $Y$.
Actually, $Y \in [0, 10]$.

**Solution:** infer the relational invariant $X = Y$ at lock boundaries.

\[ \alpha_{rel}(X) \overset{\text{def}}{=} \{ \rho \mid \exists \rho': (\rho, \rho') \in X \lor (\rho', \rho) \in X \} \in P(E) \]

(keep only constraints that are respected by the critical section)

[Miné 2014]
Summary
We presented a static analysis that is:

- inspired from thread-modular proof methods
- sound for all interleavings
- sound for weakly consistent memory semantics
- aware of scheduling and synchronization
- parametrized by abstract domains

On-going work: leverage the connection with rely-guarantee

- relational interferences
  (especially for synchronized program parts)
- flow-sensitive interferences and invariants
Bibliography


