Static Analysis of Concurrent Programs

MPRI 2–6: Abstract Interpretation, application to verification and static analysis

Antoine Miné

year 2014–2015

course 11
12 November 2013
Concurrent programming

**Idea:**
Decompose a program into a set of (loosely) interacting processes.

**Why concurrent programs?**
- **exploit** parallelism in current computers
  (multi-processors, multi-cores, hyper-threading)
  “Free lunch is over”
- **exploit** several computers
  (distributed computing)
- **ease** of programming
  (GUI, network code, reactive programs)
Many models:

- process calculi: CSP, $\pi$-calculus, join calculus
- message passing
- shared memory (threads)
- transactional memory
- combination of several models

Example implementations:

- C, C++ with a thread library (POSIX threads, Win32)
- C, C++ with a message library (MPI, OpenMP)
- Java (native threading API)
- Erlang (based on $\pi$-calculus)
- JoCaml (OCaml + join calculus)
- processor-level (interrupts, test-and-set instructions)
Introduction

Scope

In this course: **static thread model**

- implicit communication through *shared memory*
- explicit communication through *synchronisation primitives*
- **fixed** number of threads (no dynamic creation of threads)
- numeric programs (real-valued variables)

**Goal:** static analysis

- infer numeric program *invariants*
- discover possible *run-time errors* (e.g., division by 0)
- parametrized by a choice of numeric abstract domains
Outline

- **State-based analyses**
  - sequential programs *(reminders)*
  - concurrent programs

- **Toward thread-modular analyses**
  - detour through proof methods
    - (Floyd–Hoare, Owicki–Gries, Jones)
  - rely-guarantee in abstract interpretation form

- **Interference-based abstract analyses**
  - denotational-style analysis
  - weakly consistent memory models
  - synchronisation
Simple structured numeric language

- finite set of (toplevel) threads: \( \text{prog}_1 \) to \( \text{prog}_n \)
- finite set of numeric program variables \( V \in \mathbb{V} \)
- finite set of statement locations \( \ell \in \mathbb{L} \)
- finite set of potential error locations \( \omega \in \Omega \)

Structured language syntax

\[
\begin{align*}
\text{parprog} & ::= \ell \text{ prog}_1 \| \ldots \| \ell \text{ prog}_n \\
\ell \text{ prog} & ::= \ell V := \text{exp} \\
& \quad | \ell \text{ if exp} \triangleright 0 \text{ then prog} \ell \text{ fi} \\
& \quad | \ell \text{ while exp} \triangleright 0 \text{ do prog} \ell \text{ done} \\
& \quad | \ell \text{ prog}; \ell \text{ prog} \\
\text{exp} & ::= V \mid [c_1, c_2] \mid - \text{exp} \mid \text{exp} \diamond \omega \text{exp} \\
\end{align*}
\]

\( c_1, c_2 \in \mathbb{R} \cup \{+\infty, -\infty\} \), \( \diamond \in \{+,-,\times,\div\} \), \( \triangleright \in \{=,\lt,\ldots\} \)
State-based analyses
Sequential program semantics (reminders)
Transition systems

**Transition system:** \((\Sigma, \tau, \mathcal{I})\)

- \(\Sigma\): set of program states
- \(\tau \subseteq \Sigma \times \Sigma\): transition relation
  - we note \((\sigma, \sigma') \in \tau\) as \(\sigma \rightarrow_\tau \sigma'\)
- \(\mathcal{I} \subseteq \Sigma\): set of initial states

**Traces:** sequences of states \(\sigma_0, \ldots, \sigma_n, \ldots\)

- \(\Sigma^*\): finite traces
- \(\Sigma^\omega\): infinite countable traces
- \(\Sigma^\infty \overset{\text{def}}{=} \Sigma^* \cup \Sigma^\omega\): finite or infinite countable traces
- \(u \preceq t\): \(u\) is a prefix of \(t\)

We view program semantics and properties as sets of traces.
Traces of a transition system

Maximal trace semantics: \( M_{\infty} \in \mathcal{P}(\Sigma^\infty) \)

- set of total executions \( \sigma_0, \ldots, \sigma_n, \ldots \)
  - starting in an initial state \( \sigma_0 \in \mathcal{I} \) and either
  - ending in a blocking state in \( \mathcal{B} \) \( \overset{\text{def}}{=} \{ \sigma \mid \forall \sigma' : \sigma \not\rightarrow \tau \sigma' \} \)
  - or infinite

\[
M_{\infty} \overset{\text{def}}{=} \{ \sigma_0, \ldots, \sigma_n \mid \sigma_0 \in \mathcal{I} \land \sigma_n \in \mathcal{B} \land \forall i < n : \sigma_i \rightarrow \tau \sigma_{i+1} \} \cup \\
\{ \sigma_0, \ldots, \sigma_n \ldots \mid \sigma_0 \in \mathcal{I} \land \forall i < \omega : \sigma_i \rightarrow \tau \sigma_{i+1} \}
\]

- able to express many properties of programs, e.g.:
  - state safety: \( M_{\infty} \subseteq S^\infty \) \( \text{(executions stay in } S) \)
  - ordering: \( M_{\infty} \subseteq S_1^\infty \cdot S_2^\infty \) \( \text{(} S_2 \text{ can only occur after } S_1) \)
  - termination: \( M_{\infty} \subseteq \Sigma^* \) \( \text{(executions are finite)} \)
  - inevitability: \( M_{\infty} \subseteq \Sigma^* \cdot S \cdot \Sigma^\infty \) \( \text{(executions pass through } S) \)
Traces of a transition system

**Finite prefix trace semantics:** \( \mathcal{T}_p \in \mathcal{P}(\Sigma^*) \)

- set of **finite prefixes** of executions:
  \[
  \mathcal{T}_p \overset{\text{def}}{=} \{ \sigma_0, \ldots, \sigma_n \mid n \geq 0, \sigma_0 \in \mathcal{I}, \forall i < n: \sigma_i \rightarrow_{\tau} \sigma_{i+1} \}
  \]

- \( \mathcal{T}_p \) is an abstraction of the maximal trace semantics
  \[
  \mathcal{T}_p = \alpha_{\preceq}(\mathcal{M}_\infty) \quad \text{where} \quad \alpha_{\preceq}(X) \overset{\text{def}}{=} \{ t \in \Sigma^* \mid \exists u \in X: t \preceq u \}
  \]

- \( \mathcal{T}_p \) **can prove state safety** properties: \( \mathcal{T}_p \subseteq S^* \)
  (executions stay in \( S \))

- \( \mathcal{T}_p \) **can prove ordering** properties: \( \mathcal{T}_p \subseteq S_1^* \cdot S_2^* \)
  (if \( S_1 \) and \( S_2 \) occur, \( S_2 \) occurs after \( S_1 \))

- \( \mathcal{T}_p \) **cannot prove termination** nor **inevitability** properties

- **fixpoint characterisation:** \( \mathcal{T}_p = \text{lfp } F_p \) where
  \[
  F_p(X) = \mathcal{I} \cup \{ \sigma_0, \ldots, \sigma_{n+1} \mid \sigma_0, \ldots, \sigma_n \in X \land \sigma_n \rightarrow_{\tau} \sigma_{n+1} \}
  \]
State abstraction

**Reachable state semantics:** \( \mathcal{R} \in \mathcal{P}(\Sigma) \)

- set of states **reachable** in any execution:
  \[ \mathcal{R} \overset{\text{def}}{=} \{ \sigma | \exists n \geq 0, \sigma_0, \ldots, \sigma_n: \sigma_0 \in I, \forall i < n: \sigma_i \rightarrow_{\tau} \sigma_{i+1} \land \sigma = \sigma_n \} \]

- \( \mathcal{R} \) is an abstraction of the finite trace semantics: \( \mathcal{R} = \alpha_p(\mathcal{T}_p) \)
  where \( \alpha_p(X) \overset{\text{def}}{=} \{ \sigma | \exists \sigma_0, \ldots, \sigma_n \in X: \sigma = \sigma_n \} \)

- \( \mathcal{R} \) **can prove state safety** properties: \( \mathcal{R} \subseteq S \)
  (executions stay in \( S \))

- \( \mathcal{R} \) **cannot prove ordering, termination, inevitability** properties

- fixpoint characterisation: \( \mathcal{R} = \text{lfp } F_{\mathcal{R}} \)
  where
  \[ F_{\mathcal{R}}(X) = I \cup \{ \sigma | \exists \sigma' \in X: \sigma' \rightarrow_{\tau} \sigma \} \]
States of a sequential program

Simple sequential numeric programs: \( \text{parprog} ::= \ell^i \text{prog}^{\ell^x} \).

**Program states:**  \( \Sigma \overset{\text{def}}{=} (L \times E) \cup \Omega \)

- a control state in \( L \), and
- either a memory state: an environment in \( E \overset{\text{def}}{=} \mathbb{V} \rightarrow \mathbb{R} \)
- or an error state in \( \Omega \)

**Initial states:**

start at the first control point \( \ell^i \) with variables set to 0:

\( I \overset{\text{def}}{=} \{ (\ell^i, \lambda\mathbb{V}.0) \} \)

Note that \( \mathcal{P}(\Sigma) \simeq (L \rightarrow \mathcal{P}(E)) \times \mathcal{P}(\Omega) \):

- a state property in \( \mathcal{P}(E) \) at each program point in \( L \)
- and a set of errors in \( \mathcal{P}(\Omega) \)
Expression semantics with errors

**Expression semantics:** \( E[\exp] : \mathcal{E} \rightarrow (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\Omega)) \)

1. \( E[V] \rho \) \( \overset{\text{def}}{=} \langle \{ \rho(V) \}, \emptyset \rangle \)
2. \( E[[c_1, c_2]] \rho \) \( \overset{\text{def}}{=} \langle \{ x \in \mathbb{R} | c_1 \leq x \leq c_2 \}, \emptyset \rangle \)
3. \( E[-e] \rho \) \( \overset{\text{def}}{=} \text{let } \langle V, O \rangle = E[e] \rho \text{ in } \langle \{-v | v \in V\}, O \rangle \)
4. \( E[e_1 \diamond \omega e_2] \rho \) \( \overset{\text{def}}{=} \text{let } \langle V_1, O_1 \rangle = E[e_1] \rho \text{ in } \text{let } \langle V_2, O_2 \rangle = E[e_2] \rho \text{ in } \langle \{ v_1 \diamond v_2 | v_i \in V_i, \diamond \neq / \vee v_2 \neq 0 \}, O_1 \cup O_2 \cup \{ \omega \text{ if } \diamond = / \wedge 0 \in V_2 \} \rangle \)

- defined by structural induction on the syntax
- evaluates in an environment \( \rho \) to a set of values
- also returns a set of accumulated errors
  (here, only divisions by zero)
Reminders: semantics in equational form

**Principle:** (without handling errors in $\Omega$)

- see $\text{lfp } f$ as the least solution of an equation $x = f(x)$
- partition states by control: $\mathcal{P}(L \times E) \simeq L \rightarrow \mathcal{P}(E)$
  
  $X_\ell \in \mathcal{P}(E)$: invariant at $\ell \in L$

\[
\forall \ell \in L : X_\ell \overset{\text{def}}{=} \{ m \in E \mid (\ell, m) \in R \}
\]

$\implies$ set of (recursive) equations on $X_\ell$

**Example:**

\[
\begin{align*}
\ell^1 & \quad i := 2; & \quad X_1 = \mathcal{I} \\
\ell^2 & \quad n := [-\infty, +\infty]; & \quad X_2 = \mathcal{C}[i := 2] X_1 \\
\ell^3 & \quad \text{while } i < n \text{ do} & \quad X_3 = \mathcal{C}[n := [-\infty, +\infty]] X_2 \\
\ell^4 & \quad \text{if } [0, 1] = 0 \text{ then} & \quad X_4 = X_3 \cup X_7 \\
\ell^5 & \quad i := i + 1 & \quad X_5 = \mathcal{C}[i < n] X_4 \\
\ell^6 & \quad \text{fi} & \quad X_6 = X_5 \\
\ell^7 & \quad \text{done} & \quad X_7 = X_5 \cup \mathcal{C}[i := i + 1] X_6 \\
\ell^8 & \quad \text{done} & \quad X_8 = \mathcal{C}[i \geq n] X_4
\end{align*}
\]
State-based analyses
Sequential program semantics (reminders)

Semantics in denotational form

Input-output function $C[\text{prog}]$

$$C[\text{prog}] : (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \to (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega))$$

$$C[X := e] \langle R, O \rangle \overset{\text{def}}{=} \langle \emptyset, O \rangle \sqcup \bigsqcup_{\rho \in R} \langle \{ \rho[X \mapsto v] | v \in V_\rho \}, O_\rho \rangle$$

$$C[e \triangleright 0?] \langle R, O \rangle \overset{\text{def}}{=} \langle \emptyset, O \rangle \sqcup \bigsqcup_{\rho \in R} \langle \{ \rho | \exists v \in V_\rho : v \triangleright 0 \}, O_\rho \rangle$$

where $\langle V_\rho, O_\rho \rangle \overset{\text{def}}{=} E[e] \rho$

$$C[\text{if } e \triangleright 0 \text{ then } s \text{ fi }] X \overset{\text{def}}{=} (C[s] \circ C[e \triangleright 0?]) X \sqcup C[e \triangleright 0?] X$$

$$C[\text{while } e \triangleright 0 \text{ do } s \text{ done }] X \overset{\text{def}}{=} C[e \triangleright 0?](\text{lfp}\lambda Y.X \sqcup (C[s] \circ C[e \triangleright 0?]) Y)$$

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

- mutate memory states in $\mathcal{E}$, accumulate errors in $\Omega$
  ($\sqcup$ is the element-wise $\cup$ in $\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)$)
- structured: nested loops yield nested fixpoints
- big-step: forget information on intermediate locations $\ell$
Abstract semantics in denotational form

Extend a numeric abstract domain $\mathcal{E}^\#$ abstracting $\mathcal{P}(\mathcal{E})$
to $\mathcal{D}^\# \overset{\text{def}}{=} \mathcal{E}^\# \times \mathcal{P}(\Omega)$.

$$C^\#[\text{prog}] : \mathcal{D}^\# \to \mathcal{D}^\#$$

$C^\#[X := e] \langle R^\#, O \rangle$ and $C^\#[e \triangleright 0?] \langle R^\#, O \rangle$ are given

$C^\#[\text{if } e \triangleright 0 \text{ then } s \text{ fi}] X^\# \overset{\text{def}}{=} (C^\#[s] \circ C^\#[e \triangleright 0?]) X^\# \sqcup C^\#[e \triangleright 0?] X^\#$

$C^\#[\text{while } e \triangleright 0 \text{ do } s \text{ done}] X^\# \overset{\text{def}}{=} C^\#[e \triangleright 0?] (\lim \lambda Y^#. Y^# \triangleright (X^# \sqcup (C^\#[s] \circ C^\#[e \triangleright 0?] Y^#)))$

$C^\#[s_1; s_2] \overset{\text{def}}{=} C^\#[s_2] \circ C^\#[s_1]$ 

- the abstract interpreter mimicks an actual interpreter
- efficient in memory (intermediate invariants are not kept)
- less flexibility in the iteration scheme (iteration order, widening points, etc.)
Concurrent program semantics
Labelled transition systems: \((\Sigma, \mathcal{A}, \tau, \mathcal{I})\)

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

Labelled traces: sequences of states interspersed with actions

denoted as \(\sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \cdots \sigma_n \xrightarrow{a_n} \sigma_{n+1}\)
From concurrent programs to labelled transition systems

Notations:

- concurrent program:
  \[
  \text{parprog ::= } \ell_1^{i_1} \text{prog}_1 \ell_1^{x_1} \ || \cdots \ || \ell_n^{i_n} \text{prog}_n \ell_n^{x_n}
  \]
- threads identifiers: \( T \overset{\text{def}}{=} \{1, \ldots, n\} \)

Program states: \( \Sigma \overset{\text{def}}{=} ((T \to L) \times E) \cup \Omega \)

- a control state \( L(t) \in L \) for each thread \( t \in T \) and
- a single shared memory state \( \rho \in E \)
- or an error state \( \omega \in \Omega \)

Initial states:

threads start at their first control point \( \ell_t^i \), variables are set to 0:
\( I \overset{\text{def}}{=} \{ (\lambda t.\ell_t^i, \lambda V.0) \} \)

Actions: thread identifiers: \( A \overset{\text{def}}{=} T \)
Transition relation: \( \tau \subseteq \Sigma \times A \times \Sigma \)

\[
(L, \rho) \xrightarrow{t} (L', \rho') \iff (L(t), \rho) \xrightarrow{\tau[\text{prog}_t]} (L'(t), \rho') \land \\
\forall u \neq t: L(u) = L'(u)
\]

\[
(L, \rho) \xrightarrow{t} \omega \iff (L(t), \rho) \xrightarrow{\tau[\text{prog}_t]} \omega
\]

- based on the transition relation of individual threads seen as sequential processes \( \text{prog}_t \):
  \( \tau[\text{prog}] \subseteq (L \times \mathcal{E}) \times ((L \times \mathcal{E}) \cup \Omega) \)
  - choose a thread \( t \) to run
  - update \( \rho \) and \( L(t) \)
  - leave \( L(u) \) intact for \( u \neq t \)

(See course 3 for the full definition of \( \tau[\text{prog}] \).)

- each \( \sigma \rightarrow \sigma' \) in \( \tau[\text{prog}_t] \) leads to many transitions in \( \tau \)!
Interleaved trace semantics

Maximal and finite prefix trace semantics are as before:

**Blocking states:** \( B \overset{\text{def}}{=} \{ \sigma | \forall \sigma': \forall t: \sigma \overset{t}{\rightarrow}_{\tau} \sigma' \} \)

**Maximal traces:** \( \mathcal{M}_{\infty} \) (finite or infinite)

\[ \mathcal{M}_{\infty} \overset{\text{def}}{=} \{ \sigma_0 \overset{t_0}{\rightarrow} \cdots \overset{t_{n-1}}{\rightarrow} \sigma_n | n \geq 0 \land \sigma_0 \in \mathcal{I} \land \sigma_n \in \mathcal{B} \land \forall i < n: \sigma_i \overset{t_i}{\rightarrow}_{\tau} \sigma_{i+1} \} \cup \{ \sigma_0 \overset{t_0}{\rightarrow} \sigma_1 \cdots | n \geq 0 \land \sigma_0 \in \mathcal{I} \land \forall i < \omega: \sigma_i \overset{t_i}{\rightarrow}_{\tau} \sigma_{i+1} \} \]

**Finite prefix traces:** \( \mathcal{T}_p \)

\[ \mathcal{T}_p \overset{\text{def}}{=} \{ \sigma_0 \overset{t_0}{\rightarrow} \cdots \overset{t_{n-1}}{\rightarrow} \sigma_n | n \geq 0 \land \sigma_0 \in \mathcal{I} \land \forall i < n: \sigma_i \overset{t_i}{\rightarrow}_{\tau} \sigma_{i+1} \} \]

Fixpoint form: \( \mathcal{T}_p = \text{lfp } F_p \) where

\[ F_p(X) = \mathcal{I} \cup \{ \sigma_0 \overset{t_0}{\rightarrow} \cdots \overset{t_n}{\rightarrow} \sigma_{n+1} | n \geq 0 \land \sigma_0 \overset{t_0}{\rightarrow} \cdots \overset{t_{n-1}}{\rightarrow} \sigma_n \in X \land \sigma_n \overset{t_n}{\rightarrow}_{\tau} \sigma_{n+1} \} \]

Abstraction: \( \mathcal{T}_p = \alpha_{\leq}(\mathcal{M}_{\infty}) \)


**Fairness conditions:** avoid threads being denied to run

Given \( enabled(\sigma, t) \) defined as \( \exists \sigma' \in \Sigma: \sigma \xrightarrow{t} \sigma' \), an infinite trace \( \sigma_0 \xrightarrow{t_0} \cdots \sigma_n \xrightarrow{t_n} \cdots \) is:

- **Weakly fair** if \( \forall t \in T: \)  
  \[
  (\exists i: \forall j \geq i: enabled(\sigma_j, t)) \implies (\forall i: \exists j \geq i: a_j = t)
  \]
  (no thread can be continuously enabled without running)

- **Strongly fair** if \( \forall t \in T: \)  
  \[
  (\forall i: \exists j \geq i: enabled(\sigma_j, t)) \implies (\forall i: \exists j \geq i: a_j = t)
  \]
  (no thread can be infinitely often enabled without running)

**Proofs under fairness conditions** given:
- the maximal traces \( M_\infty \) of a program
- a property \( X \) to prove (as a set of traces)
- the set \( F \) of all (weakly or strongly) fair and of finite traces

\[ \implies \text{prove } M_\infty \cap F \subseteq X \text{ instead of } M_\infty \subseteq X \]
Fairness (cont.)

Example: while $x \geq 0$ do $x:=x+1$ done || $x:=-1$

- may not terminate without fairness
- always terminates under weak and strong fairness

Finite prefix trace abstraction

$\mathcal{M}_\infty \cap F \subseteq X$ is abstracted into testing $\alpha\star\preceq(\mathcal{M}_\infty \cap F) \subseteq \alpha\star\preceq(X)$

for all fairness conditions $F$, $\alpha\star\preceq(\mathcal{M}_\infty \cap F) = \alpha\star\preceq(\mathcal{M}_\infty) = T_p$

$\implies$ fairness-dependent properties cannot be proved with finite prefixes only

In the following, we ignore fairness conditions.

(see [Cous85])
Equational state semantics

**State abstraction \( \mathcal{R} \):** as before

\[
\mathcal{R} \overset{\text{def}}{=} \{ \sigma \mid \exists n \geq 0, \sigma_0 \xrightarrow{t_0} \cdots \sigma_n : \sigma_0 \in \mathcal{I} \ \forall i < n : \sigma_i \xrightarrow{t_i} \tau \sigma_{i+1} \land \sigma = \sigma_n \}
\]

\[
\mathcal{R} = \alpha_p(\mathcal{T}_p) \text{ where } \alpha_p(X) \overset{\text{def}}{=} \{ \sigma \mid \exists n \geq 0, \sigma_0 \xrightarrow{t_0} \cdots \sigma_n \in X : \sigma = \sigma_n \}
\]

\[
\mathcal{R} = \text{lfp } F_{\mathcal{R}} \text{ where } F_{\mathcal{R}}(X) = \mathcal{I} \cup \{ \sigma \mid \exists \sigma' \in X, t \in \mathbb{T} : \sigma' \xrightarrow{t} \tau \sigma \}
\]

**Equational form:** (without handling errors in \( \Omega \))

- for each \( L \in \mathbb{T} \rightarrow \mathcal{L} \), a variable \( X_L \) with value in \( \mathcal{E} \)
- equations are derived from thread equations \( eq(\text{prog}_t) \) as:
  \[
  X_{L_1} = \bigcup_{t \in \mathbb{T}} \{ \text{for } \exists (X_{\ell_1} = F(X_{\ell_2}, \ldots, X_{\ell_N})) \in eq(\text{prog}_t):
  \forall i \leq N: L_i(t) = \ell_i, \forall u \neq t: L_i(u) = L_1(u) \}
  \]
  Join with \( \cup \) equations from \( eq(\text{prog}_t) \) updating a single thread \( t \in \mathbb{T} \).
(See course 3 for the full definition of \( eq(\text{prog}) \).)
Equational state semantics (example)

Example: inferring $0 \leq x \leq y \leq 102$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $\ell_1 0 = 0$ do $\ell_2$</td>
<td>while $\ell_4 0 = 0$ do $\ell_5$</td>
</tr>
<tr>
<td>if $x &lt; y$ then</td>
<td>if $y &lt; 100$ then</td>
</tr>
<tr>
<td>$\ell_3 x := x + 1$</td>
<td>$\ell_6 y := y + [1,3]$</td>
</tr>
<tr>
<td>fi</td>
<td>fi</td>
</tr>
<tr>
<td>done</td>
<td>done</td>
</tr>
</tbody>
</table>
Equational state semantics (example)

Example: inferring $0 \leq x \leq y \leq 102$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $\ell^1 0 = 0$ do $\ell^2$</td>
<td>while $\ell^4 0 = 0$ do $\ell^5$</td>
</tr>
<tr>
<td>if $x &lt; y$ then</td>
<td>if $y &lt; 100$ then</td>
</tr>
<tr>
<td>$\ell^3 x := x + 1$</td>
<td>$\ell^6 y := y + [1,3]$</td>
</tr>
<tr>
<td>fi</td>
<td>fi</td>
</tr>
<tr>
<td>done</td>
<td>done</td>
</tr>
</tbody>
</table>

(Simplified) equation system:

$\mathcal{X}_{1,4} = \mathcal{T} \cup C[ x := x + 1 ] \mathcal{X}_{3,4} \cup C[ x \geq y ] \mathcal{X}_{2,4}$

$\cup C[ y := y + [1,3] ] \mathcal{X}_{1,6} \cup C[ y \geq 100 ] \mathcal{X}_{1,5}$

$\mathcal{X}_{2,4} = \mathcal{X}_{1,4} \cup C[ y := y + [1,3] ] \mathcal{X}_{2,6} \cup C[ y \geq 100 ] \mathcal{X}_{2,5}$

$\mathcal{X}_{3,4} = C[ x < y ] \mathcal{X}_{2,4} \cup C[ y := y + [1,3] ] \mathcal{X}_{3,6} \cup C[ y \geq 100 ] \mathcal{X}_{3,5}$

$\mathcal{X}_{1,5} = C[ x := x + 1 ] \mathcal{X}_{3,5} \cup C[ x \geq y ] \mathcal{X}_{2,5} \cup \mathcal{X}_{1,4}$

$\mathcal{X}_{2,5} = \mathcal{X}_{1,5} \cup \mathcal{X}_{2,4}$

$\mathcal{X}_{3,5} = C[ x < y ] \mathcal{X}_{2,5} \cup \mathcal{X}_{3,4}$

$\mathcal{X}_{1,6} = C[ x := x + 1 ] \mathcal{X}_{3,6} \cup C[ x \geq y ] \mathcal{X}_{2,6} \cup C[ y < 100 ] \mathcal{X}_{1,5}$

$\mathcal{X}_{2,6} = \mathcal{X}_{1,6} \cup C[ y < 100 ] \mathcal{X}_{2,5}$

$\mathcal{X}_{3,6} = C[ x < y ] \mathcal{X}_{2,6} \cup C[ y < 100 ] \mathcal{X}_{3,5}$
Equational state semantics (example)

Example: inferring $0 \leq x \leq y \leq 102$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $\ell^1 0 = 0$ do $\ell^2$&lt;br&gt;if $x &lt; y$ then&lt;br&gt;$\ell^3 x := x + 1$&lt;br&gt;fi&lt;br&gt;done</td>
<td>while $\ell^4 0 = 0$ do $\ell^5$&lt;br&gt;if $y &lt; 100$ then&lt;br&gt;$\ell^6 y := y + [1, 3]$&lt;br&gt;fi&lt;br&gt;done</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
- $\implies$ 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 $\mathcal{L}$, not control point tuples in $\mathbb{T} \rightarrow \mathcal{L}$)

- 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)
Detour through proof methods
Floyd–Hoare logic

Logic to prove properties about **sequential** programs [Hoar69].

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

- annotate programs with **logic assertions** \( \{P\} \text{prog}\{Q\} \)
  
  (if \( P \) holds before \( \text{prog} \), then \( Q \) holds after \( \text{prog} \))

- check that \( \{P\} \text{prog}\{Q\} \) is derivable with the following rules
  
  (the assertions are program invariants)

\[
\begin{align*}
\{P[e/X]\} X := e \{P\} & \quad \{P \land e \triangleright 0\} \ s \{Q\} \quad P \land e \triangleright 0 \Rightarrow Q \\
\{P\} s_1 \{Q\} \quad \{Q\} s_2 \{R\} & \quad \{P\} \text{if } e \triangleright 0 \text{ then } s \text{fi} \{Q\} \\
\{P\} s_1 ; s_2 \{R\} & \quad \{P \land e \triangleright 0\} s \{P\} \\
\{P'\} s \{Q'\} & \quad P \Rightarrow P' \quad Q' \Rightarrow Q \\
\{P\} s \{Q\} & \quad \{P \land e \triangleright 0\} \ s \{P\} \\
\end{align*}
\]
Towards thread-modular analyses

Floyd–Hoare logic as abstract interpretation

Link with the equational state semantics:

Correspondence between $\ell \text{prog}^{\ell'}$ and $\{P\} \text{prog} \{Q\}$:

- if $P$ (resp. $Q$) models exactly the points in $X_\ell$ (resp. $X_{\ell'}$) then $\{P\} \text{prog} \{Q\}$ is a derivable Hoare triple

- if $\{P\} \text{prog} \{Q\}$ is derivable, then $X_\ell \models P$ and $X_{\ell'} \models Q$
  (all the points in $X_\ell$ (resp. $X_{\ell'}$) satisfy $P$ (resp. $Q$))

$\Rightarrow X_\ell$ provide the most precise Hoare assertions in a constructive form

- $\gamma(X^\#)$ provide (less precise) Hoare assertions in a computable form

Link with the denotational semantics:

both $C[\text{prog}]$ and the proof tree for $\{P\} \text{prog} \{Q\}$ reflect the syntactic structure of prog
(compositional methods)
Owicki–Gries proof method

Extension of Floyd–Hoare to concurrent programs [Owic76].

**Principle:** add a new rule, for $||$

$$
\begin{align*}
\{ P_1 \} s_1 \{ Q_1 \} & \quad \{ P_2 \} s_2 \{ Q_2 \} \\
\{ P_1 \land P_2 \} s_1 || s_2 \{ Q_1 \land Q_2 \}
\end{align*}
$$
Towards thread-modular analyses

Detour through proof methods

Owicki–Gries proof method

Extension of Floyd–Hoare to concurrent programs [Owic76].

**Principle:** add a new rule, for $||$

\[
\begin{align*}
\{P_1\} & \text{ } s_1 \{Q_1\} & \{P_2\} & \text{ } s_2 \{Q_2\} \\
\{P_1 \land P_2\} & s_1 \text{ } || \text{ } s_2 \{Q_1 \land Q_2\}
\end{align*}
\]

This rule is not always sound!

e.g., we have \(\{X = 0, Y = 0\} X := 1 \{X = 1, Y = 0\}\)
and \(\{X = 0, Y = 0\} \text{ if } X = 0 \text{ then } Y := 1 \text{ fi } \{X = 0, Y = 1\}\)
but not \(\{X = 0, Y = 0\} X := 1 \text{ || if } X = 0 \text{ then } Y := 1 \text{ fi } \{false\}\)

\[\Rightarrow \text{ we need a side-condition to the rule: }\]
\(\{P_1\} s_1 \{Q_1\} \text{ and } \{P_2\} s_2 \{Q_2\} \text{ must not interfere}\)
Towards thread-modular analyses

Detour through proof methods

Owicki–Gries proof method (cont.)

interference freedom

given proofs $\Delta_1$ and $\Delta_2$ of $\{P_1\} s_1 \{Q_1\}$ and $\{P_2\} s_2 \{Q_2\}$

$\Delta_1$ does not interfere with $\Delta_2$ if:

- for any $\Phi$ appearing before a statement in $\Delta_1$
- for any $\{P'_2\} s'_2 \{Q'_2\}$ appearing in $\Delta_2$
- $\{\Phi \land P'_2\} s'_2 \{\Phi\}$ holds
- and moreover $\{Q_1 \land P'_2\} s'_2 \{Q_1\}$

i.e.: the assertions used to prove $\{P_1\} s_1 \{Q_1\}$ are stable by $s_2$

e.g.,

\[
\{X = 0, Y \in [0, 1]\} X := 1 \{X = 1, Y \in [0, 1]\}
\]

\[
\{X \in [0, 1], Y = 0\} \text{ if } X = 0 \text{ then } Y := 1 \text{ fi } \{X \in [0, 1], Y \in [0, 1]\}
\]

\[
\Longrightarrow \{X = 0, Y = 0\} X := 1 \mid\mid \text{ if } X = 0 \text{ then } Y := 1 \text{ fi } \{X = 1, Y \in [0, 1]\}
\]

Summary:

- **pros**: the invariants are local to threads
- **cons**: the proof is **not compositional**

(proving one thread requires delving into the proof of other threads)

\[\Longrightarrow \text{ not satisfactory}\]
Jones’ rely-guarantee proof method

**Idea:** explicit interferences with (more) annotations [Jone81].

Rely-guarantee “quintuples”: \( R, G \vdash \{ P \} \text{prog} \{ Q \} \)

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

where:

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

The parallel composition rule becomes:

\[
\begin{align*}
R \lor G_2, G_1 & \vdash \{ P_1 \} \ s_1 \ \{ Q_1 \} \quad R \lor G_1, G_2 & \vdash \{ P_2 \} \ s_2 \ \{ Q_2 \} \\
R, G_1 \lor G_2 & \vdash \{ P_1 \land P_2 \} \ s_1 \ || \ s_2 \ \{ Q_1 \land Q_2 \}
\end{align*}
\]
Rely-guarantee example

Example: proving $0 \leq x \leq y \leq 102$

**Checking $t_1$**

```plaintext
while $\ell_1 0 = 0$ do
  if $x < y$ then
    $\ell_3 x := x + 1$
  fi
done
```

at $\ell_1, \ell_2$: $x, y \in [0, 102], x \leq y$

at $\ell_3$: $x \in [0, 101], y \in [1, 102], x < y$

**Checking $t_2$**

```plaintext
while $\ell_4 0 = 0$ do
  if $y < 100$ then
    $\ell_6 y := y + [1, 3]$
  fi
done
```

at $\ell_4, \ell_5$: $x, y \in [0, 102], x \leq y$

at $\ell_6$: $x \in [0, 99], y \in [0, 99], x \leq y$

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:
- 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 $0 \leq x \leq y \leq 102$

checking $t_1$

while $\ell_1 0 = 0$ do 
  if $x < y$ then 
    $\ell_3 x := x + 1$
  fi
  done

at $\ell_1$, $\ell_2$ : $x, y \in [0, 102]$, $x \leq y$

checking $t_2$

while $\ell_4 0 = 0$ do 
  $\ell_5 y := y + [1, 3]$
  fi
done

at $\ell_4$, $\ell_5$ : $x, y \in [0, 102]$, $x \leq y$

In this example:

- guarantee exactly what is relied on \((R_1 = G_1 \text{ and } R_2 = G_2)\)
- rely and guarantee are global assertions

**Benefits of rely-guarantee:**

- invariants are still local to threads
- checking a thread does not require looking at other threads, only at an abstraction of their semantics
Auxiliary variables

**Example**

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell^1$ $x := x + 1$ $\ell^2$</td>
<td>$\ell^3$ $x := x + 1$ $\ell^4$</td>
</tr>
</tbody>
</table>

**Goal:** prove $\{x = 0\} t_1 \parallel t_2 \{x = 2\}$. 
Towards thread-modular analyses

Auxiliary variables

Example

<table>
<thead>
<tr>
<th></th>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_1$</td>
<td>$x := x + 1$</td>
<td>$\ell_3$</td>
</tr>
<tr>
<td>$\ell_2$</td>
<td></td>
<td>$\ell_4$</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:

- past values of variables  (history of the computation)
- program counter of other threads  ($pc_t$)

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

\[
x := x + 1
\]

$\{(pc_2 = \ell_3 \wedge x = 1) \lor (pc_2 = \ell_4 \wedge x = 2)\}$
Rely-guarantee as abstract interpretation
Local states

**State projection:** on a thread $t \in \mathbb{T}$

- add auxiliary variables $\mathbb{V}_t \overset{\text{def}}{=} \mathbb{V} \cup \{ pc_{t'} \mid t' \in \mathbb{T}, t' \neq t \}$
- enriched environments for $t$: $\mathcal{E}_t \overset{\text{def}}{=} \mathbb{V}_t \rightarrow \mathbb{R}$ (for simplicity, $pc_{t'}$ are numeric variables, i.e., $\mathcal{L} \subseteq \mathbb{R}$)
- local states: $\Sigma_t \overset{\text{def}}{=} (\mathcal{L} \times \mathcal{E}_t) \cup \Omega$
  recall that $\Sigma \overset{\text{def}}{=} ((\mathbb{T} \rightarrow \mathcal{L}) \times \mathcal{E}) \cup \Omega$
  $\Sigma_t$ has a simpler, sequential control state
- **projection:** $\pi_t(L, \rho) \overset{\text{def}}{=} (L(t), \rho[\forall t' \neq t: pc_{t'} \mapsto L(t')])$
  from $\Sigma$ to $\Sigma_t$: shift control state to auxiliary variables
  extended naturally to $\pi_t : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma_t)$
  $\pi_t$ is a bijection, no information is lost
Local invariants

Abstraction steps to local reachable states:
- concrete (prefix) labelled trace semantics: $\mathcal{T}_p$
Local invariants

Abstraction steps to local reachable states:
- concrete (prefix) labelled trace semantics: $T_p$
- state reachability abstraction: $R = \alpha_p(T_p) \in \mathcal{P}(\Sigma)$
Abstraction steps to local reachable states:

- concrete (prefix) labelled trace semantics: $\mathcal{T}_p$
- state reachability abstraction: $\mathcal{R} = \alpha_p(\mathcal{T}_p) \in \mathcal{P}(\Sigma)$
- local state reachability: $\mathcal{Rl}(t) \overset{\text{def}}{=} \pi_t(\mathcal{R}) \in \mathcal{P}(\Sigma_t)$

thread’s view of reachable states
**Interference**: \( A \in \mathbb{T} \rightarrow \mathcal{P}(\Sigma \times \Sigma) \) caused by a thread \( t \in \mathbb{T} \)

\[
A(t) \overset{\text{def}}{=} \alpha^{\text{itf}}(\mathcal{T}_p)(t)
\]

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

Subset of the transition system \( \tau \):
- spawned by \( t \)
- and actually observed in some execution trace \((\text{in } \mathcal{T}_p)\)
Towards thread-modular analyses

Rely-guarantee as abstract interpretation

Fixpoint form

Local state fixpoint:

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

$$\mathcal{Rl}(t) = \text{lfp } R_t(A) \text{ where}$$

$$R_t : (\mathbb{T} \rightarrow \mathcal{P}(\Sigma \times \Sigma)) \rightarrow \mathcal{P}(\Sigma_t) \rightarrow \mathcal{P}(\Sigma_t)$$

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

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

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 $Y$. 
Fixpoint form

**Local state fixpoint:** illustration

Ifp $R_t(A)$ interleaves:
- transitions in $\pi_t$ from thread $t$
Fixpoint form

Local state fixpoint: illustration

\[
\text{Ifp } R_t(A) \text{ interleaves:}
\]

- transitions in \( \pi_t \) from thread \( t \)
- transitions in \( A \) from interferences
Towards thread-modular analyses
Rely-guarantee as abstract interpretation

Fixpoint form

**Local state fixpoint:**

\[ \text{lfp} R_t (A) \text{ interleaves:} \]
- transitions in \( \pi_t \) from thread \( t \)
- transitions in \( A \) from interferences
we express $A(t)$ as a function of $\mathcal{RL}$ and thread $t \in T$:

$$A(t) = B(\mathcal{RL})(t) \text{ where}$$

$$B : (\prod_{t \in T} \{t\} \rightarrow \mathcal{P}(\Sigma_t)) \rightarrow T \rightarrow \mathcal{P}(\Sigma \times \Sigma)$$

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

Collect transitions starting from reachable states.
No fixpoint needed.
Fixpoint form (cont.)

Nested fixpoint characterization:

1. $\mathcal{R}l(t) = \text{lfp } R_t(A)$
2. $A(t) = B(\mathcal{R}l)(t)$
3. mutual dependency between $\mathcal{R}l$ and $A$
Fixpoint form (cont.)

Nested fixpoint characterization:

1. $\mathcal{R}l(t) = \text{lfp } R_t(A)$

2. $A(t) = B(\mathcal{R}l)(t)$

3. mutual dependency between $\mathcal{R}l$ and $A$
   $\implies$ solved using a fixpoint:
   $\mathcal{R}l = \text{lfp } H$ where
   $H : (\prod_{t \in T} \{t\} \rightarrow \mathcal{P}(\Sigma_t)) \rightarrow (\prod_{t \in T} \{t\} \rightarrow \mathcal{P}(\Sigma_t))$
   $H(Z)(t) \overset{\text{def}}{=} \text{lfp } R_t(B(Z))$
Towards thread-modular analyses

Rely-guarantee as abstract interpretation

Fixpoint form (cont.)

**Nested fixpoint characterization:**

1. \( R_l(t) = \text{lfp } R_t(A) \)
2. \( A(t) = B(R_l)(t) \)
3. mutual dependency between \( R_l \) and \( A \)
   \( \implies \) solved using a fixpoint:
   \( R_l = \text{lfp } H \) where
   \[
   H : (\prod_{t \in T} \{t\} \to P(\Sigma_t)) \to (\prod_{t \in T} \{t\} \to P(\Sigma_t))
   \]
   \[
   H(Z)(t) \overset{\text{def}}{=} \text{lfp } R_t(B(Z))
   \]

**Completeness:** \( \forall t : R_l(t) \simeq R \) (\( \pi_t \) is bijective thanks to auxiliary variables)
Constructive fixpoint form:

Use Kleene’s iteration to construct fixpoints:

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

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

$\implies$ nested iterations
Abstract rely-guarantee

**Suggested algorithm:** nested iterations with acceleration

once abstract domains for states and interferences are chosen

- start from $\mathcal{R}_0 \equiv A_0 \equiv \lambda t. \bot$
- while $A_n$ is not stable
  - compute $\forall t \in \mathbb{T}: \mathcal{R}_{n+1}(t) \equiv \text{lfp } R_t(A_n)$ by iteration with widening $\nabla$
  - $(\simeq \text{separate analysis of each thread})$
  - compute $A_{n+1} \equiv A_n \nabla B(A_{n+1})$
- when $A_n = A_{n+1}$, return $\mathcal{R}_n$

$\implies$ thread-modular analysis parameterized by abstract domains 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

\[
\alpha^n_{R} : \mathcal{P}(\Sigma_t) \rightarrow \mathcal{P}((\mathcal{L} \times \mathcal{E}) \cup \Omega)
\]

\[
\alpha^n_{R}(X) \overset{\text{def}}{=} \{(l, \rho|_V) \mid (l, \rho) \in X\} \cup (X \cap \Omega)
\]

**Interference abstraction:** remove all control state

\[
\alpha^n_{A} : \mathcal{P}(\Sigma \times \Sigma) \rightarrow \mathcal{P}(\mathcal{E} \times \mathcal{E})
\]

\[
\alpha^n_{A}(Y) \overset{\text{def}}{=} \{(\rho, \rho') \mid \exists L, L' \in \mathbb{T} \rightarrow \mathcal{L}: ((L, \rho), (L', \rho')) \in Y\}
\]
Flow-insensitive abstraction (cont.)

**Flow-insensitive fixpoint semantics:** (omitting errors $\Omega$)

We apply $\alpha_{R}^{nf}$ and $\alpha_{A}^{nf}$ to the nested fixpoint semantics.

$$R_{nf}^{nf} \overset{\text{def}}{=} \text{lfp } \lambda Z. \lambda t. \text{lfp } R_{nf}^{t}(B_{nf}^{t}(Z)),$$  
where

$$B_{nf}^{t}(Z)(t) \overset{\text{def}}{=} \{(\rho, \rho') | \exists \ell, \ell' \in L: (\ell, \rho) \in Z(t) \land (\ell, \rho) \rightarrow_{t} (\ell', \rho') \}$$
(extract interferences from reachable states)

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

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

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

where $\rightarrow_{t}$ is the transition relation for thread $t$ alone: $\tau[\text{prog}_{t}]$

**Cost/precision trade-off:**

- less variables
  $\implies$ subsequent numeric abstractions are more efficient
- sufficient to analyze our first example (slide 26)
- insufficient to analyze $x := x + 1 \parallel x := x + 1$ (slide 35)
Towards thread-modular analyses

Rely-guarantees as abstract interpretation

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:

\[ \alpha^{nr}_A : \mathcal{P}(E \times E) \to (V \to \mathcal{P}(\mathbb{R})) \]

\[ \alpha^{nr}_A(Y) \overset{\text{def}}{=} \lambda V.\{ x \in V \mid \exists (\rho, \rho') \in Y: \rho(V) \neq x \wedge \rho'(V) = x \} \]

(remember which variables are modified and their new values)

To apply interferences, we get, in the nested fixpoint form:

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

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

- there is still a finite syntactic set of threads $\mathbb{T}_s$
- some threads $\mathbb{T}_\infty \subseteq \mathbb{T}_s$ 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 $t \in \mathbb{T}_s$ are isomorphic
  $\implies$ iterate the analysis on the finite set $\mathbb{T}_s$ (instead of $\mathbb{T}$)

- we must handle self-interferences for threads in $\mathbb{T}_\infty$:
  \[
  A_t^{nf}(Y)(X) \overset{\text{def}}{=} \{ (\ell, \rho') \mid \exists \rho, \ u: (u \neq t \lor t \in \mathbb{T}_\infty) \land (\ell, \rho) \in X \land (\rho, \rho') \in Y(u) \} \]
Towards thread-modular analyses  
Rely-guarantee as abstract interpretation

From traces to thread-modular analyses

abstract states
$(\mathbb{T} \times \mathcal{L}) \rightarrow \mathcal{E}^\#$

abstract interferences
$\mathbb{T} \rightarrow \mathcal{E}^\#$

static analyzer

non-relational interferences
$\mathbb{T} \rightarrow \mathcal{P}(\mathcal{E})$

rely-guarantee
(without aux. variables)

flow-insensitive interferences
$\mathbb{T} \rightarrow \mathcal{P}(\mathcal{E} \times \mathcal{E})$

rely-guarantee
(with aux. variables)

local states
$(\mathbb{T} \times \mathcal{L}) \rightarrow \mathcal{P}(\mathcal{E})$

interferences
$A : \mathbb{T} \rightarrow \mathcal{P}(\Sigma \times \Sigma)$

test

interleaved execution trace prefixes
$\mathcal{T}_p \in \mathcal{P}(\Sigma^*)$

local states
$\mathbb{T} \times \mathcal{L} \rightarrow \mathcal{P}(\mathcal{E})$

interferences
$\mathbb{T} \rightarrow \mathcal{P}(\Sigma \times \Sigma)$

\[ \mathcal{R}l : \prod_{t \in \mathbb{T}} \{t\} \rightarrow \mathcal{P}(\Sigma_t) \]

\[ \pi_t \]

\[ \alpha_{\mathcal{E}} \]

\[ \alpha_{\mathcal{E}}^{nf} \]

\[ \alpha_{\mathcal{A}}^{nf} \]

\[ \alpha_{\mathcal{A}}^{itf} \]
Towards thread-modular analyses

Rely-guarantee as abstract interpretation

Compare with sequential analyses

abstract states

$\mathcal{L} \rightarrow \varepsilon^*$

states

$R \in \mathcal{P}(\Sigma)$

execution trace prefixes

$T_p \in \mathcal{P}(\Sigma^*)$

static analyzer

reachability

test

Lecture 11
Static Analysis of Concurrent Programs
Antoine Miné

p. 51 / 81
Construction of an interference-based analysis
Reminder: sequential analysis in denotational form

Expression semantics: \( E[\text{exp}] : \mathcal{E} \rightarrow (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\Omega)) \)

\[
E[X] \rho \overset{\text{def}}{=} \langle \{ \rho(X) \} , \emptyset \rangle \\
E[c_1, c_2] \rho \overset{\text{def}}{=} \langle \{ x \in \mathbb{R} \mid c_1 \leq x \leq c_2 \} , \emptyset \rangle \\
E[-e] \rho \overset{\text{def}}{=} \text{let } \langle V, O \rangle = E[e] \rho \text{ in } \langle \{ -\nu \mid \nu \in V \} , O \rangle \\
E[e_1 \diamond_\omega e_2] \rho \overset{\text{def}}{=} \\
\text{let } \langle V_1, O_1 \rangle = E[e_1] \rho \text{ in } \\
\text{let } \langle V_2, O_2 \rangle = E[e_2] \rho \text{ in } \\
\langle \{ v_1 \diamond v_2 \mid v_i \in V_i ; \diamond \neq \lor v_2 \neq 0 \} , O_1 \cup O_2 \cup \{ \omega \text{ if } \diamond = \lor 0 \in V_2 \} \rangle
\]

Statement semantics: \( C[\text{prog}] : (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \rightarrow (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \)

\[
C[X := e] \langle R, O \rangle \overset{\text{def}}{=} \langle \emptyset , O \rangle \sqcup \bigsqcup_{\rho \in R} \langle \{ \rho[X] \mapsto \nu \} \mid \nu \in V_{\rho} \rangle , O_{\rho} \rangle \\
C[e \gg 0?] \langle R, O \rangle \overset{\text{def}}{=} \langle \emptyset , O \rangle \sqcup \bigsqcup_{\rho \in R} \langle \{ \rho \mid \exists \nu \in V_{\rho} : \nu \gg 0 \} , O_{\rho} \rangle \\
C[\text{if } e \gg 0 \text{ then } s \text{ fi}] X \overset{\text{def}}{=} (C[s] \circ C[e \gg 0?])X \sqcup C[e \gg 0?] X \\
C[\text{while } e \gg 0 \text{ do } s \text{ done}] X \overset{\text{def}}{=} C[e \gg 0?] (\text{lfp } \lambda Y.X \sqcup (C[s] \circ C[e \gg 0?] )Y) \\
C[s_1; s_2] \overset{\text{def}}{=} C[s_2] \circ C[s_1] \\
\text{where } \langle V_{\rho} , O_{\rho} \rangle \overset{\text{def}}{=} E[e] \rho
\]
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 \( t \)

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

- Apply interferences to read variables:
  \[
  E_t[ X ] \langle \rho, I \rangle \overset{\text{def}}{=} \langle \{ \rho(X) \} \cup \{ v | \exists u \neq t: \langle u, X, v \rangle \in I \}, \emptyset \rangle
  \]
- Pass recursively \( I \) down to sub-expressions:
  \[
  E_t[\neg e] \langle \rho, I \rangle \overset{\text{def}}{=} \text{let } \langle V, O \rangle = E_t[ e ] \langle \rho, I \rangle \text{ in } \langle \{ -v | v \in V \}, O \rangle
  \]

\[ \ldots \]
Denotational semantics with interferences (cont.)

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

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

- pass interferences to expressions
- collect new interferences due to assignments
- accumulate interferences from inner statements

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

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

... 

noting $$\langle V_\rho, O_\rho \rangle \overset{\text{def}}{=} E_t[e] \langle \rho, I \rangle$$

$\sqcup$ is now the element-wise $\cup$ in $\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(I)$
Program semantics: \( P[\text{parprog}] \subseteq \Omega \)

Given \( \text{parprog} ::= \text{prog}_1 \parallel \cdots \parallel \text{prog}_n \), we compute:

\[
P[\text{parprog}] \overset{\text{def}}{=} \left[ \text{lfp} \lambda \langle O, I \rangle \cdot \bigcup_{t \in T} [C_t[\text{prog}_t] \langle E_0, \emptyset, I \rangle]_{\Omega, I} \right]_{\Omega}
\]

- each thread analysis starts in an initial environment set \( E_0 \overset{\text{def}}{=} \{ \lambda V.0 \} \)

- \([X]_{\Omega, I}\) projects \( X \in \mathcal{P}(E) \times \mathcal{P}(\Omega) \times \mathcal{P}(\bot) \) on \( \mathcal{P}(\Omega) \times \mathcal{P}(\bot) \) and interferences and errors from all threads are joined (the output environments are ignored)

- \( P[\text{parprog}] \) only outputs the set of possible run-time errors
## Concrete interference semantics:

**iteration 1**

\[ I = \emptyset \]

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

\[ \ell_4 : x = 0, \ y \in [0, 102] \]

\[ \text{new } I = \{ \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 102 \rangle \} \]
Example

Concrete interference semantics:

iteration 2

\[ \mathcal{I} = \{ \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 102 \rangle \} \]

\[ \ell_1 : x \in [0, 102], y = 0 \]

\[ \ell_4 : x = 0, y \in [0, 102] \]

new \( \mathcal{I} = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 102 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 102 \rangle \} \]
Concrete interference semantics:

iteration 3

\[ l = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 102 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 102 \rangle \} \]

\[ \ell^1 : x \in [0, 102], \ y = 0 \]

\[ \ell^4 : x = 0, \ y \in [0, 102] \]

new \( l = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 102 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 102 \rangle \} \)
Construction of an interference-based analysis

Example

```
Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>while ( \ell_1 0 = 0 ) do ( \ell_2 )</td>
<td>while ( \ell_4 0 = 0 ) do ( \ell_5 )</td>
</tr>
<tr>
<td>if ( x &lt; y ) then ( \ell_3 ) ( x := x + 1 )</td>
<td>if ( y &lt; 100 ) then ( \ell_6 ) ( y := y + [1, 3] )</td>
</tr>
<tr>
<td>fi</td>
<td>fi</td>
</tr>
<tr>
<td>done</td>
<td>done</td>
</tr>
</tbody>
</table>
```

Concrete interference semantics:

iteration 3
\[
I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 102 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 102 \rangle \}
\]
\[
\ell_1 : x \in [0, 102], y = 0
\]
\[
\ell_4 : x = 0, y \in [0, 102]
\]
new \( I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 102 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 102 \rangle \} \)

Note: we don't get that \( x \leq y \) at \( \ell_1 \), only that \( x, y \in [0, 102] \)
Abstract interferences \( \# \)

\[ \mathcal{P}(\#) \overset{\text{def}}{=} \mathcal{P}(\mathbb{T} \times \mathbb{V} \times \mathbb{R}) \text{ is abstracted as } \# \overset{\text{def}}{=} (\mathbb{T} \times \mathbb{V}) \to \mathcal{R}\# \]

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

Abstract semantics with interferences \( C_t[\# s] \)

derived from \( C[\# s] \) in a generic way:

**Example:** \( C_t[\# x := e \langle R\#, \Omega, I\# \rangle] \)

- for each \( y \) in \( e \), get its interference \( y_{\mathcal{R}}^{\#} = \bigcup_{\mathcal{R}} \{ I^{\#}(u, y) | u \neq t \} \)
- if \( y_{\mathcal{R}}^{\#} \neq \bot_{\mathcal{R}}^{\#} \), replace \( y \) in \( e \) with \( \text{get}(y, R^{\#}) \bigcup_{\mathcal{R}} y_{\mathcal{R}}^{\#} \)
  (where \( \text{get}(y, R^{\#}) \) extracts the abstract values in \( \mathcal{R}\# \) of a variable \( y \) from \( R^{\#} \in \mathcal{E}\# \))
- compute \( \langle R'^{\#}, O' \rangle = C[\# e \langle R^{\#}, O \rangle] \)
- enrich \( I^{\#}(t, x) \) with \( \text{get}(x, R'^{\#}) \)
Construction of an interference-based analysis

Static analysis with interferences

Abstract analysis

\[
P^\#[\text{parprog}] \overset{\text{def}}{=} \lim_{\lambda} \langle O, I^\# \rangle \cup \bigcup_{t \in T} \left[ C_t^\#[\text{prog}_t] \langle E^\#, 0, I^\# \rangle \right]_{\Omega, \#} \]

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

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

(reminder: \([X]_{\Omega}, [Y]_{\Omega, \#}\) keep only \(X\)’s component in \(\Omega\), \(Y\)’s components in \(\Omega\) and \(\#\))
Path-based semantics
atomic ::= X := exp | exp ⊢◁ 0?

Control paths

\[ \pi : \text{prog} \rightarrow \mathcal{P}(\text{atomic}^*) \]

\[ \pi(X := e) \overset{\text{def}}{=} \{X := e\} \]

\[ \pi(\text{if } e \triangleright 0 \text{ then } s \text{ fi}) \overset{\text{def}}{=} (\{e \triangleright 0?\} \cdot \pi(s)) \cup \{e \triangleright 0?\} \]

\[ \pi(\text{while } e \triangleright 0 \text{ do } s \text{ done}) \overset{\text{def}}{=} (\bigcup_{i \geq 0} (\{e \triangleright 0?\} \cdot \pi(s))^i) \cdot \{e \triangleright 0?\} \]

\[ \pi(s_1; s_2) \overset{\text{def}}{=} \pi(s_1) \cdot \pi(s_2) \]

\[ \pi(\text{prog}) \text{ is a (generally infinite) set of finite control paths} \]
Path-based concrete semantics of sequential programs

Join-over-all-path semantics

\[ \Pi[P] : (\mathcal{P}(E) \times \mathcal{P}(\Omega)) \rightarrow (\mathcal{P}(E) \times \mathcal{P}(\Omega)) \]

\[ \Pi[P] \langle R, O \rangle \overset{\text{def}}{=} \bigsqcup_{s_1 \cdots s_n \in P} (C[s_n] \circ \cdots \circ C[s_1]) \langle R, O \rangle \]

Semantic equivalence

\[ C[\text{prog}] = \Pi[\pi(\text{prog})] \]

(not true in the abstract)

Advantages:

- easily extended to concurrent programs (path interleavings)
- able to model program transformations (weak memory models)
Concurrent control paths

\[ \pi_* \overset{\text{def}}{=} \{ \text{interleavings of } \pi(\text{prog}_t), \ t \in T \} \]
\[ = \{ p \in \text{atomic}^* | \forall t \in T, \ \text{proj}_t(p) \in \pi(\text{prog}_t) \} \]

Interleaving program semantics

\[ P_*[\text{parprog}] \overset{\text{def}}{=} [\prod[p] \langle E_0, \emptyset \rangle]_\Omega \]

\((\text{proj}_t(p) \text{ keeps only the atomic statement in } p \text{ coming from thread } t)\)
Soundness of the interference semantics

Soundness theorem

\[ P_\ast[\text{parprog}] \subseteq P[\text{parprog}] \]

Proof sketch:

- define \( \Pi_{p}[P]X \overset{\text{def}}{=} \bigsqcup \{ C_{t}[s_{1}; \ldots; s_{n}] X | s_{1} \cdot \ldots \cdot s_{n} \in P \} \), then \( \Pi_{t}[\pi(s)] = C_{t}[s] \);

- given the interference fixpoint \( I \subseteq I \) from \( P[\text{parprog}] \), prove by recurrence on the length of \( p \in \pi_{\ast} \) that:
  \[ \forall t \in T, \forall \rho \in [\Pi[p]\langle E_0, \emptyset \rangle]_{\mathcal{E}}, \exists \rho' \in [\Pi_{t}[\text{proj}_{t}(p)]\langle E_0, \emptyset, I \rangle]_{\mathcal{E}} \text{ such that } \forall X \in V, \rho(X) = \rho'(X) \text{ or } \langle u, X, \rho(X) \rangle \in l \text{ for some } u \neq t. \]
  \[ [\Pi[p]\langle E_0, \emptyset \rangle]_{\Omega} \subseteq \bigcup_{t \in T} [\Pi_{t}[\text{proj}_{t}(p)]\langle E_0, \emptyset, I \rangle]_{\Omega} \]

Note: sound but not complete
Weakly consistent memories
Issues with weak consistency

Program written

\begin{align*}
F_1 & := 1; \\
\text{if } F_2 = 0 \text{ then } S_1 \text{ fi} \\
\text{if } F_1 = 0 \text{ then } S_2 \text{ fi}
\end{align*}

(simplified Dekker mutual exclusion algorithm)

$S_1$ and $S_2$ cannot execute simultaneously.
Issues with weak consistency

(simplified Dekker mutual exclusion algorithm)

$S_1$ 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 [Mans05]
Out of thin air principle

\[
\begin{align*}
R_1 & := X; & R_2 & := Y; \\
Y & := R_1 & X & := R_2
\end{align*}
\]

(Example from causality test case #4 for Java by Pugh et al.)

We should not have \( R_1 = 42 \).
### Out of thin air principle

#### Original Program

<table>
<thead>
<tr>
<th>Original Program</th>
<th>&quot;Optimized&quot; Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 := X );</td>
<td>( Y := 42 );</td>
</tr>
<tr>
<td>( Y := R_1 )</td>
<td>( R_1 := X );</td>
</tr>
<tr>
<td>( X := R_2 )</td>
<td>( R_2 := Y );</td>
</tr>
</tbody>
</table>

(Example from causality test case #4 for Java by Pugh et al.)

We should not have \( R_1 = 42 \).

Possible if we allow speculative writes!

\( \Rightarrow \) we **disallow** this kind of program transformations.

(Also forbidden in Java)
Atomicity and granularity

original program

\[ X := X + 1, \quad X := X + 1 \]

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

[Reyn04]
Construction of an interference-based analysis
Weakly consistent memories

Path-based definition of weak consistency

Acceptable control path transformations: \( p \leadsto q \)
only reduce interferences and errors

- **Reordering:** \( X_1 := e_1 \cdot X_2 := e_2 \leadsto X_2 := e_2 \cdot X_1 := e_1 \)
  (if \( X_1 \notin \text{var}(e_2), X_2 \notin \text{var}(e_1), \) and \( e_1 \) does not stop the program)

- **Propagation:** \( X := e \cdot s \leadsto X := e \cdot s[e/X] \)
  (if \( X \notin \text{var}(e), \text{var}(e) \) are thread-local, and \( e \) is deterministic)

- **Factorization:** \( s_1 \cdot \ldots \cdot s_n \leadsto X := e \cdot s_1[X/e] \cdot \ldots \cdot s_n[X/e] \)
  (if \( X \) is fresh, \( \forall i, \text{var}(e) \cap \text{lval}(s_i) = \emptyset \), and \( e \) has no error)

- **Decomposition:** \( X := e_1 + e_2 \leadsto T := e_1 \cdot X := T + e_2 \)
  (change of granularity)

  …

- **but NOT:**
  - “out-of-thin-air” writes: \( X := e \leadsto X := 42 \cdot X := e \)
Soundness of the interference semantics

Interleaving semantics of transformed programs $P'_*[\text{parprog}]$

- $\pi'(s) \overset{\text{def}}{=} \{ p \mid \exists p' \in \pi(s): p' \rightsquigarrow * p \}$
- $\pi' \overset{\text{def}}{=} \{ \text{interleavings of } \pi'(\text{prog}_t), \ t \in T \}$
- $P'_*[\text{parprog}] \overset{\text{def}}{=} \left[ \prod \pi'_* \langle \mathcal{E}_0, \emptyset \rangle \right]_\Omega$

Soundness theorem

$P'_*[\text{parprog}] \subseteq P[\text{parprog}]$

$\implies$ the interference semantics is sound

wrt. weakly consistent memories and changes of granularity
Synchronisation
Scheduling

Synchronization primitives

\[
\text{prog} ::= \text{lock}(m) \\
| \quad \text{unlock}(m)
\]

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

Scheduling

- mutexes ensure mutual exclusion
  at each time, 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
Construction of an interference-based analysis

Synchronisation

Mutual exclusion

Interleaving semantics $P[*[\text{parprog}]]$:
restrict interleavings of control paths

Interference semantics $P[[\text{parprog}]]$, $P^#[[\text{parprog}]]$:
partition wrt. an abstract local view of the scheduler $C$

\[\mathcal{E} \rightsquigarrow \mathcal{E} \times C, \quad \mathcal{E}^# \rightsquigarrow C \rightarrow \mathcal{E}^#\]

\[I \overset{\text{def}}{=} T \times V \times R \rightsquigarrow I \overset{\text{def}}{=} T \times C \times V \times R,\]

\[I^# \overset{\text{def}}{=} (T \times V) \rightarrow \mathcal{R}^# \rightsquigarrow I^# \overset{\text{def}}{=} (T \times C \times V) \rightarrow \mathcal{R}^#\]
Mutual exclusion

Data-race effects

Partition wrt. mutexes $M \subseteq \mathbb{M}$ held by the current thread $t$

- $C_t[ X := e ] \langle \rho, M, I \rangle$ adds
  \[ \{ \langle t, M, X, v \rangle \mid v \in E_t[ X ] \langle \rho, M, I \rangle \} \] to $I$

- $E_t[ X ] \langle \rho, M, I \rangle = \{ \rho(X) \} \cup \{ v \mid \langle t', M', X, v \rangle \in I, t \neq t', M \cap M' = \emptyset \}$

- flow-insensitive, subject to weak memory consistency
Mutual exclusion

Well-synchronized effects

- last write before unlock affects first read after lock
- partition interferences wrt. a protecting mutex $m$ (and $M$)
- $C_t[\text{unlock}(m)] \langle \rho, M, I \rangle$ stores $\rho(X)$ into $I$
- $C_t[\text{lock}(m)] \langle \rho, M, I \rangle$ imports values form $I$ into $\rho$
- imprecision: non-relational, largely flow-insensitive
Construction of an interference-based analysis

Example analysis

<table>
<thead>
<tr>
<th>abstract consumer/producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
</tr>
<tr>
<td>while 0=0 do</td>
</tr>
<tr>
<td>lock(m); $\ell^1$</td>
</tr>
<tr>
<td>if $X &gt; 0$ then $\ell^2$</td>
</tr>
<tr>
<td>$X := X - 1$ fi;</td>
</tr>
<tr>
<td>unlock(m); $\ell^3$</td>
</tr>
<tr>
<td>$Y := X$</td>
</tr>
<tr>
<td>done</td>
</tr>
<tr>
<td>$t_2$</td>
</tr>
<tr>
<td>while 0=0 do</td>
</tr>
<tr>
<td>lock(m);</td>
</tr>
<tr>
<td>if $X &gt; 10$ then $X := 10$ fi;</td>
</tr>
<tr>
<td>unlock(m)</td>
</tr>
<tr>
<td>done</td>
</tr>
</tbody>
</table>

- at $\ell^1$, the unlock-lock effect from $t_2$ imports $\{X\} \times [1, 10]$
- at $\ell^2$, $X \in [1, 10]$, no effect from $t_2$: $X := X - 1$ is safe
- at $\ell^3$, $X \in [0, 9]$, and $t_2$ has the effects $\{X\} \times [1, 10]$ so, $X \in [0, 10]$
Limitations of the interference abstraction
a difficult example

\[ \mathcal{E}_0 : X = Y = 5 \]

<table>
<thead>
<tr>
<th></th>
<th>while 0=0 do</th>
<th>while 0=0 do</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(m);</td>
<td>lock(m);</td>
<td></td>
</tr>
<tr>
<td>if X&gt;0 then</td>
<td>if X&lt;10 then</td>
<td></td>
</tr>
<tr>
<td>X:=X-1;</td>
<td>X:=X+1;</td>
<td></td>
</tr>
<tr>
<td>Y:=Y-1;</td>
<td>Y:=Y+1;</td>
<td></td>
</tr>
<tr>
<td>fi;</td>
<td>fi;</td>
<td></td>
</tr>
<tr>
<td>unlock(m)</td>
<td>unlock(m)</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done</td>
<td></td>
</tr>
</tbody>
</table>

Our analysis finds \( X \in [0, 10] \), but no bound on \( Y \).

Actually \( Y \in [0, 10] \).

To prove this, we would need to infer the relational invariant \( X = Y \) at lock boundaries.
Our analysis finds no bound on $X$.

Actually $X \in [-2, 2]$ at all program points.

To prove this we need to infer an invariant on the history of interleaved executions:

no more than two incrementation (resp. decrementation) can occur without a decrementation (resp. incrementation).
Bibliography


