COMP4161: Advanced Topics in Software Verification

$P || Q$

Gerwin Klein, June Andronick, Christine Rizkallah, Miki Tanaka

S2/2018

Content

- ➜ Intro & motivation, getting started [1]
- ➜ Foundations & Principles
  - Lambda Calculus, natural deduction [1,2]
  - Higher Order Logic [3[a]]
  - Term rewriting [4]
- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction [6, 7]
  - Hoare logic, proofs about programs, invariants [8[b],9]
  - (mid-semester break)
  - C verification [10]
  - CakeML, Isar [11[c]]
  - Concurrency [12]

[a] a1 due; [b] a2 due; [c] a3 due

Program verification so far

If the following true?

$\{x = 0\}$
$y := x;$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Program verification so far

If the following true?

$\{x = 0\}$
$y := x;$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

YES!

Program verification with concurrency

Is it still true?

$$\{x = 0\}$$
$$y := x; \qquad || \quad x := 4$$
$$x := x + 1;$$
$$\{x = 1 \wedge y = 0\}$$

Program verification with concurrency

Is it still true?

$$\{x = 0\}$$
$$y := x; \qquad\qquad || \quad x := 4$$
$$x := x + 1;$$
$$\{x = 1 \wedge y = 0\}$$

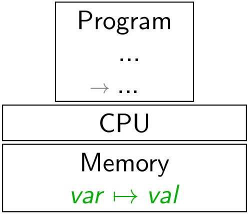
NO!

Program verification so far

So far we have assumed **sequential execution**

$\{x = 0\}$
$y := x;$
$x := x + 1;$
$\{x = 1 \land y = 0\}$

i.e. a single thread of execution accessing the memory state

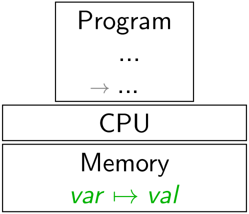


Program verification so far

So far we have assumed **sequential execution**

$\{x = 0\}$	$x \mapsto 0$	$y \mapsto -$
$y := x;$	$x \mapsto 0$	$y \mapsto 0$
$x := x + 1;$	$x \mapsto 1$	$y \mapsto 0$
$\{x = 1 \land y = 0\}$		

i.e. a single thread of execution accessing the memory state

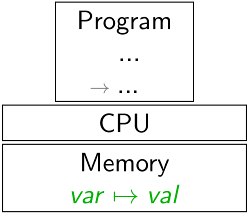


Program verification so far

So far we have assumed **sequential execution**

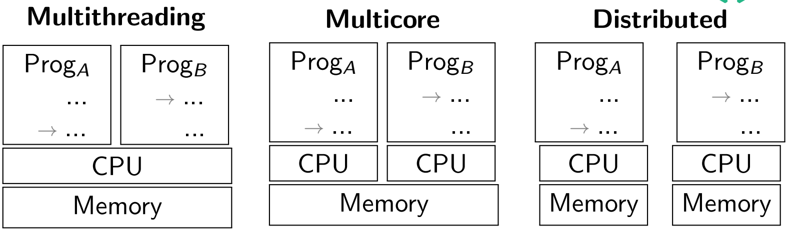
$\{x = 0\}$	$x \mapsto 0$	$y \mapsto -$
$y := x;$	$x \mapsto 0$	$y \mapsto 0$
$x := x + 1;$	$x \mapsto 1$	$y \mapsto 0$
$\{x = 1 \wedge y = 0\}$		

i.e. a single thread of execution accessing the memory state

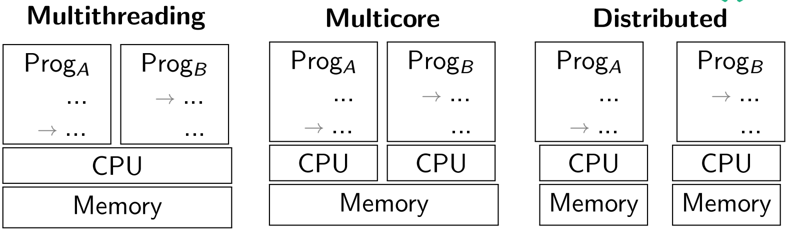


This is not always the case!

Types of concurrency



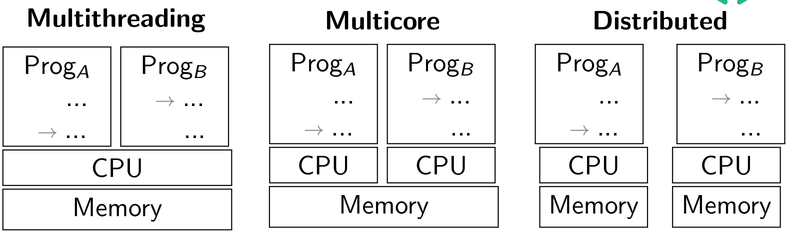
Types of concurrency



All need communication and synchronisation mechanisms

Shared memory | Shared memory | Message passing

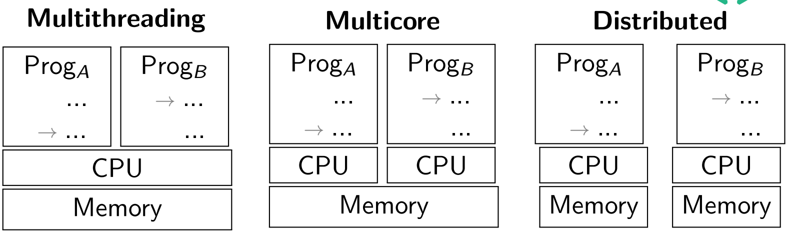
Types of concurrency



All need communication and synchronisation mechanisms

Multithreading	Multicore	Distributed
Shared memory Interleaved execution	Shared memory Parallel execution	Message passing

Types of concurrency

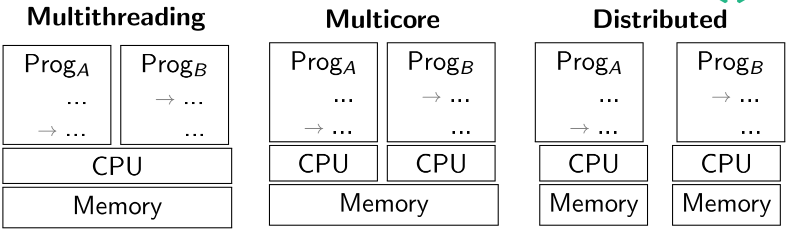


All need communication and synchronisation mechanisms

Multithreading	Multicore	Distributed
Shared memory Interleaved execution	Shared memory Parallel execution	Message passing

Here: we'll look at shared-memory concurrency

Types of concurrency



All need communication and synchronisation mechanisms

Multithreading	Multicore	Distributed
Shared memory Interleaved execution	Shared memory Parallel execution	Message passing

Here: we'll look at shared-memory concurrency

(and we'll ignore further complications such as caches, weak memory...)

Goal

We want to be able to reason about parallel composition of programs:

Prog_A		Prog_B
$\ldots$	$\|\|$	$\rightarrow \ldots$
$\rightarrow \ldots$		$\ldots$

Goal

We want to be able to reason about parallel composition of programs:

$\{precondition\}$

Prog_A		Prog_B
$\dots$	$\parallel$	$\rightarrow \dots$
$\rightarrow \dots$		$\dots$

$\{postcondition\}$

Goal

We want to be able to reason about parallel composition of programs:

$$\{\textit{precondition}\}$$

$$\boxed{\begin{array}{r}\text{Prog}_A \\ \ldots \\ \rightarrow \ldots\end{array}} \quad || \quad \boxed{\begin{array}{r}\text{Prog}_B \\ \rightarrow \ldots \\ \ldots\end{array}}$$

$$\{\textit{postcondition}\}$$

2 kinds of properties:

Safety:
"something bad does not happen"
(no bad state can be reached)

Liveness:
"something good must happen"
(specific states must be reached)

Goal

We want to be able to reason about parallel composition of programs:

$\{\textit{precondition}\}$

Prog_A		Prog_B
$\dots$	$\|$	$\rightarrow \dots$
$\rightarrow \dots$		$\dots$

$\{\textit{postcondition}\}$

2 kinds of properties:

Safety:
"something bad does not happen"
(no bad state can be reached)
e.g. $\{x = 0\}$

Liveness:
"something good must happen"
(specific states must be reached)
e.g. the program terminates

Goal

We want to be able to reason about parallel composition of programs:

$$\{\textit{precondition}\}$$

$$\boxed{\begin{array}{l} \text{Prog}_A \\ \quad\quad \dots \\ \quad \rightarrow \dots \end{array}} \quad \Big\| \quad \boxed{\begin{array}{l} \text{Prog}_B \\ \rightarrow \dots \\ \quad\quad \dots \end{array}}$$

$$\{\textit{postcondition}\}$$

2 kinds of properties:

Safety:
"something bad does not happen"
(no bad state can be reached)
e.g. $\{x = 0\}$

Liveness:
"something good must happen"
(specific states must be reached)
e.g. the program terminates

With concurrency: much harder!
(set of reachable states much bigger)

Goal

We want to be able to reason about parallel composition of programs:

$\{\mathit{precondition}\}$

Prog_A		Prog_B
$\ldots$	$\|\|$	$\rightarrow \ldots$
$\rightarrow \ldots$		$\ldots$

$\{\mathit{postcondition}\}$

2 kinds of properties:

Safety:
"something bad does not happen"
(no bad state can be reached)
e.g. $\{x = 0\}$

Liveness:
"something good must happen"
(specific states must be reached)
e.g. the program terminates

With concurrency: much harder!
(set of reachable states much bigger)

With concurrency: new problems!
(dead-locks, live-locks...)

Goal

We want to be able to reason about parallel composition of programs:

$$\{\mathit{precondition}\}$$

$$\boxed{\begin{array}{l}\text{Prog}_A \\ \qquad \ldots \\ \quad \rightarrow \ldots\end{array}} \quad \| \quad \boxed{\begin{array}{l}\text{Prog}_B \\ \quad \rightarrow \ldots \\ \qquad \ldots\end{array}}$$

$$\{\mathit{postcondition}\}$$

Here:

- ➜ We focus on **safety** properties: postcondition holds **if reached**

Goal

We want to be able to reason about parallel composition of programs:

$\{\textit{precondition}\}$

Prog_A		Prog_B
$\dots$	$\|\|$	$\rightarrow \dots$
$\rightarrow \dots$		$\dots$

$\{\textit{postcondition}\}$

Here:

- ➜ We focus on **safety** properties: postcondition holds **if reached**
- ➜ We will define parallel composition ($||$) as non-deterministic interleaving

Goal

We want to be able to reason about parallel composition of programs:

$\{precondition\}$

Prog_A		Prog_B
$\dots$	$\|\|$	$\rightarrow \dots$
$\rightarrow \dots$		$\dots$

$\{postcondition\}$

Here:

- ➜ We focus on **safety** properties: postcondition holds **if reached**
- ➜ We will define parallel composition ($||$) as non-deterministic interleaving
- ➜ We go back to our minimal IMP language (forget about C and monads)

```
datatype com  =   SKIP
              |   Assign vname aexp       (_ := _)
              |   Semi com com            (_; _)
              |   Cond bexp com com       (IF _ THEN _ ELSE _)
              |   While bexp com          (WHILE _ DO _ OD)
```

Program verification so far

If the following true?

$$\{x = 0\}$$
$$y := x;$$
$$x := x + 1;$$
$$\{x = 1 \wedge y = 0\}$$

YES!

Program verification with concurrency

Is it still true?

$$\{x = 0\}$$
$$y := x; \qquad || \quad x := 4$$
$$x := x + 1;$$
$$\{x = 1 \wedge y = 0\}$$

NO!

Program verification with concurrency

Is it still true?

$$\{x = 0\}$$
$$y := x; \qquad\qquad || \quad x := 4$$
$$x := x + 1;$$
$$\{x = 1 \land y = 0\}$$

NO!

What is going wrong?
What do we need to change?

- ➜ to make sure we don't prove wrong statements!
- ➜ to allow us to prove true statements about concurrent programs

Program verification so far

How would we have proved this?

$\{x = 0\}$
$y := x;$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Program verification so far

How would we have proved this? Using Hoare logic rules!

$\{x = 0\}$
$y := x;$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Program verification so far

How would we have proved this?

$\{x = 0\}$
$y := x;$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Using Hoare logic rules!

$$\frac{\vdash \ \{P\}\ c_1\ \{R\} \quad \vdash \ \{R\}\ c_2\ \{Q\}}{\vdash \ \{P\} \quad c_1; c_2 \quad \{Q\}}$$

$$\frac{}{\vdash \ \{P[x \mapsto e]\} \quad x := e \quad \{P\}}$$

Program verification so far

How would we have proved this?

$\{x = 0\}$
$y := x;\ \{x + 1 = 1 \wedge y = 0\}$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Using Hoare logic rules!

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1; c_2\ \{Q\}}$$

$$\frac{}{\vdash \{P[x \mapsto e]\}\ x := e\ \{P\}}$$

Program verification so far

How would we have proved this?

$\{x = 0\} \implies \{x + 1 = 1 \wedge x = 0\}$
$y := x;\ \{x + 1 = 1 \wedge y = 0\}$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Using Hoare logic rules!

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ \ c_1; c_2\ \ \{Q\}}$$

$$\frac{}{\vdash \{P[x \mapsto e]\}\ \ x := e\ \ \{P\}}$$

Program verification so far

How would we have proved this?

$\{x = 0\} \implies \{x + 1 = 1 \wedge x = 0\}$
$y := x;$ $\{x + 1 = 1 \wedge y = 0\}$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Using Hoare logic rules!

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1; c_2\ \{Q\}}$$

$$\frac{}{\vdash \{P[x \mapsto e]\}\ \ x := e\ \ \{P\}}$$

Why does this make it true? What does it mean that it's true?

It means:

If the program "$y := x;\ x := x + 1$" is executed from a state satisfying $\{x = 0\}$ then, if it terminates, the resulting state satisfied $\{x = 1 \wedge y = 0\}$

Program verification so far

How would we have proved this?

$\{x = 0\} \quad \Longrightarrow \{x + 1 = 1 \wedge x = 0\}$
$y := x;\ \{x + 1 = 1 \wedge y = 0\}$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Using Hoare logic rules!

$$\frac{\vdash\ \{P\}\ c_1\ \{R\} \quad \vdash\ \{R\}\ c_2\ \{Q\}}{\vdash\ \{P\}\quad c_1; c_2 \quad \{Q\}}$$

$$\frac{}{\vdash\ \{P[x \mapsto e]\}\quad x := e \quad \{P\}}$$

Why does this make it true? What does it mean that it's true?

It means:

$$\langle y := x;\ x := x + 1, \sigma\rangle \rightarrow \sigma' \quad \wedge \quad x\ \sigma = 0 \quad \longrightarrow \quad x\ \sigma' = 1\ \wedge\ y\ \sigma' = 0$$

Program verification so far

How would we have proved this?

$\{x = 0\} \implies \{x + 1 = 1 \wedge x = 0\}$
$y := x;\ \{x + 1 = 1 \wedge y = 0\}$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Using Hoare logic rules!

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\quad c_1; c_2 \quad \{Q\}}$$

$$\frac{}{\vdash \{P[x \mapsto e]\}\quad x := e \quad \{P\}}$$

Why does this make it true? What does it mean that it's true?

It means:

$$\langle y := x;\ x := x + 1, \sigma\rangle \rightarrow \sigma' \quad \wedge \quad x\ \sigma = 0 \quad \longrightarrow \quad x\ \sigma' = 1\ \wedge\ y\ \sigma' = 0$$

Where:

$$\frac{\langle c_1, \sigma\rangle \rightarrow \sigma' \quad \langle c_2, \sigma'\rangle \rightarrow \sigma''}{\langle c_1; c_2, \sigma\rangle \rightarrow \sigma''} \qquad \frac{e\ \sigma = v}{\langle \mathrm{x} := \mathrm{e}, \sigma\rangle \rightarrow \sigma[x \mapsto v]}$$

Program verification so far

How would we have proved this?

$\{x = 0\} \Longrightarrow \{x + 1 = 1 \wedge x = 0\}$
$y := x;\ \{x + 1 = 1 \wedge y = 0\}$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Using Hoare logic rules!

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1; c_2\ \{Q\}}$$

$$\frac{}{\vdash \{P[x \mapsto e]\}\ x := e\ \{P\}}$$

Why does this make it true? What does it mean that it's true?

It means:

$$\langle y := x;\ x := x + 1, \sigma\rangle \to \sigma' \ \wedge\ x\ \sigma = 0 \ \longrightarrow\ x\ \sigma' = 1 \ \wedge\ y\ \sigma' = 0$$

Where:

$$\frac{\langle c_1, \sigma\rangle \to \sigma' \quad \langle c_2, \sigma'\rangle \to \sigma''}{\langle c_1; c_2, \sigma\rangle \to \sigma''} \qquad \frac{e\ \sigma = v}{\langle \mathrm{x} := \mathrm{e}, \sigma\rangle \to \sigma[x \mapsto v]}$$

Soundness: $\vdash \{P\}\ c\ \{Q\} \Longrightarrow \forall \sigma\ \sigma'.\ \langle c, \sigma\rangle \to \sigma' \wedge P\ \sigma \longrightarrow Q\ \sigma'$

Program verification so far

How would we have proved this?

$\{x = 0\} \implies \{x + 1 = 1 \wedge x = 0\}$
$y := x;\ \{x + 1 = 1 \wedge y = 0\}$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

Using Hoare logic rules!

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1; c_2\ \{Q\}}$$

$$\frac{}{\vdash \{P[x \mapsto e]\}\ x := e\ \{P\}}$$

Why does this make it true? What does it mean that it's true?

It means:

$$\langle y := x;\ x := x + 1, \sigma\rangle \to \sigma' \ \wedge\ x\ \sigma = 0 \ \longrightarrow\ x\ \sigma' = 1 \ \wedge\ y\ \sigma' = 0$$

Where:

$$\frac{\langle c_1, \sigma\rangle \to \sigma' \quad \langle c_2, \sigma'\rangle \to \sigma''}{\langle c_1; c_2, \sigma\rangle \to \sigma''} \qquad \frac{e\ \sigma = v}{\langle \mathrm{x} := \mathrm{e}, \sigma\rangle \to \sigma[x \mapsto v]}$$

Soundness: $\vdash \{P\}\ c\ \{Q\} \implies \forall \sigma\ \sigma'.\ \langle c, \sigma\rangle \to \sigma' \wedge P\ \sigma \longrightarrow Q\ \sigma'$

What changes when we have another program running in parallel?

Program verification with concurrency

$$\{x = 0\}$$
$$y := x;\ \cancel{\{x + 1 = 1 \wedge y = 0\}} \quad || \quad x := 4$$
$$x := x + 1;$$
$$\{x = 1 \wedge y = 0\}$$

Program verification with concurrency

$$\{x = 0\}$$
$$y := x;\ \cancel{\{x + 1 = 1 \wedge y = 0\}} \quad || \quad x := 4$$
$$x := x + 1;$$
$$\{x = 1 \wedge y = 0\}$$

➜ Execution is interleaved; Intermediate assertions can be interferred with

Program verification with concurrency

$$
\begin{array}{lcl}
\{x = 0\} & & \\
y := x;\ \cancel{\{x + 1 = 1 \wedge y = 0\}} & \quad || \quad & x := 4 \\
x := x + 1; & & \\
\{x = 1 \wedge y = 0\} & &
\end{array}
$$

➜ Execution is interleaved; Intermediate assertions can be interferred with

➜ Need a new reasoning framework!

Program verification with concurrency

$\{x = 0\}$
$y := x;\ \cancel{\{x + 1 = 1 \wedge y = 0\}} \quad || \quad x := 4$
$x := x + 1;$
$\{x = 1 \wedge y = 0\}$

➜ Execution is interleaved; Intermediate assertions can be interferred with

➜ Need a new reasoning framework!

➜ New syntax, new semantics,
new proof rules (proved sound w.r.t semantics), new VCG

Program verification with concurrency

$$\{x = 0\}$$

$$y := x;\ \cancel{\{x + 1 = 1 \wedge y = 0\}} \quad || \quad x := 4$$

$$x := x + 1;$$

$$\{x = 1 \wedge y = 0\}$$

➜ Execution is interleaved; Intermediate assertions can be interferred with

➜ Need a new reasoning framework!

➜ New syntax, new semantics,
new proof rules (proved sound w.r.t semantics), new VCG

➜ (1969: Hoare Logic (Tony Hoare))

➜ 1976: Owicki-Gries (Susan Owicki and David Gries)

➜ 1981: Rely-Guarantee (Cliff Jones)

➜ ...

Program verification with concurrency

$$\{x = 0\}$$
$$y := x;\ \cancel{\{x + 1 = 1 \wedge y = 0\}} \quad || \quad x := 4$$
$$x := x + 1;$$
$$\{x = 1 \wedge y = 0\}$$

- ➜ Execution is interleaved; Intermediate assertions can be interferred with
- ➜ Need a new reasoning framework!
- ➜ New syntax, new semantics,
  new proof rules (proved sound w.r.t semantics), new VCG
- ➜ (1969: Hoare Logic (Tony Hoare))
- ➜ 1976: Owicki-Gries (Susan Owicki and David Gries)
- ➜ 1981: Rely-Guarantee (Cliff Jones)
- ➜ ...

OG+RG formalised in Isabelle/HOL by Leonor Prensa Nieto, 2002

Owicki-Gries framework

Intuition:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:
  - $P \mid\mid Q$: pick one program and execute its current instruction
  - $AWAIT\ b\ DO\ c\ OD$: if guard is true execute c atomically

Owicki-Gries framework

Intuition:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:
  - $P \mid\mid Q$: pick one program and execute its current instruction
  - $AWAIT\ b\ DO\ c\ OD$: if guard is true execute c atomically
- Proof rules:
  - you prove *local correctness* (as before)
  - your prove *interference-freedom* (assertions not interfered with)

Owicki-Gries framework

Intuition:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:
  - $P \mid\mid Q$: pick one program and execute its current instruction
  - *AWAIT b DO c OD*: if guard is true execute c atomically
- Proof rules:
  - you prove *local correctness* (as before)
  - your prove *interference-freedom* (assertions not interfered with)

$$
\begin{array}{l|l}
\{is\_even\ x\} & \\
x := x + 1; & x := x + 2 \\
x := x + 1; & \\
\{is\_even\ x\} &
\end{array}
$$

Owicki-Gries framework

Intuition:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:
  - $P \,||\, Q$: pick one program and execute its current instruction
  - *AWAIT b DO c OD*: if guard is true execute c atomically
- Proof rules:
  - you prove *local correctness* (as before)
  - your prove *interference-freedom* (assertions not interfered with)

$$\begin{array}{l} \{\mathit{is\_even}\ x\} \\ x := x + 1;\ \{\mathit{is\_even}\ x + 1\} \\ x := x + 1; \\ \{\mathit{is\_even}\ x\} \end{array} \;\Bigg\|\; x := x + 2$$

Owicki-Gries framework

Intuition:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:
  - $P \mid\mid Q$: pick one program and execute its current instruction
  - $AWAIT\ b\ DO\ c\ OD$: if guard is true execute c atomically
- Proof rules:
  - you prove *local correctness* (as before)
  - your prove *interference-freedom* (assertions not interfered with)

$$\begin{array}{l} \{is\_even\ x\} \\ x := x + 1;\ \{is\_even\ x + 1\} \\ x := x + 1; \\ \{is\_even\ x\} \end{array} \quad \Bigg\| \quad x := x + 2$$

➜ **Needs a fully annotated program!**

➜ **Needs a "small-step semantics"** $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$
(before big-step: $\langle c, \sigma \rangle \rightarrow \sigma'$)

Owicki-Gries framework

Formally:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:

$$\frac{\langle c_1, \sigma \rangle \rightarrow \langle c_1', \sigma' \rangle}{\langle c_1 || c_2, \sigma \rangle \rightarrow \langle c_1' || c_2, \sigma' \rangle} \quad \frac{\langle c_2, \sigma \rangle \rightarrow \langle c_2', \sigma' \rangle}{\langle c_1 || c_2, \sigma \rangle \rightarrow \langle c_1 || c_2', \sigma' \rangle}$$

Owicki-Gries framework

Formally:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:

$$\frac{\langle c_1, \sigma\rangle \rightarrow \langle c_1', \sigma'\rangle}{\langle c_1 || c_2, \sigma\rangle \rightarrow \langle c_1' || c_2, \sigma'\rangle} \quad \frac{\langle c_2, \sigma\rangle \rightarrow \langle c_2', \sigma'\rangle}{\langle c_1 || c_2, \sigma\rangle \rightarrow \langle c_1 || c_2', \sigma'\rangle}$$

- Hoare rules:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \quad \{P_2\}\ c_2\ \{Q_2\} \quad \mathit{interfree}\ c_1\ c_2 \quad \mathit{interfree}\ c_2\ c_1}{\{P_1 \wedge P_2\}\ c_1 || c_2\ \{Q_1 \wedge Q_2\}}$$

Owicki-Gries framework

Formally:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:

$$\frac{\langle c_1, \sigma \rangle \rightarrow \langle c_1', \sigma' \rangle}{\langle c_1 || c_2, \sigma \rangle \rightarrow \langle c_1' || c_2, \sigma' \rangle} \quad \frac{\langle c_2, \sigma \rangle \rightarrow \langle c_2', \sigma' \rangle}{\langle c_1 || c_2, \sigma \rangle \rightarrow \langle c_1 || c_2', \sigma' \rangle}$$

- Hoare rules:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \quad \{P_2\}\ c_2\ \{Q_2\} \quad \mathit{interfree}\ c_1\ c_2 \quad \mathit{interfree}\ c_2\ c_1}{\{P_1 \wedge P_2\}\ c_1 || c_2\ \{Q_1 \wedge Q_2\}}$$

Where

$\mathit{interfree}\ c_1\ c_2 \equiv$

$\forall p \in (\mathit{assertions}\ c_1).\ \forall (a, c) \in (\mathit{atomics}\ c_2).\ \{p \wedge a\} c \{p\}$

Owicki-Gries framework

➜ **Quadratic explosion of proof obligations! (verification conditions)**
➜ **Not compositional**
➜ **Not complete:** sometimes need auxilliary/ghost variables

Owicki-Gries framework

- ➜ **Quadratic explosion of proof obligations! (verification conditions)**
- ➜ **Not compositional**
- ➜ **Not complete:** sometimes need auxilliary/ghost variables

$$\{x = 0\}$$
$$x := x + 1; \parallel x := x + 1$$
$$\{x = 2\}$$

Owicki-Gries framework

➜ **Quadratic explosion of proof obligations! (verification conditions)**
➜ **Not compositional**
➜ **Not complete:** sometimes need auxilliary/ghost variables

$$\{x = 0\}$$
$$x := x + 1; \parallel x := x + 1$$
$$\{x = 2\}$$

$$\{x = 0 \wedge a_1 = 0 \wedge a_2 = 0\}$$

$$< x := x + 1; a_1 := 1 > \parallel < x := x + 1; a_2 := 1 >$$

$$\{x = 2\}$$

Owicki-Gries framework

- ➜ **Quadratic explosion of proof obligations! (verification conditions)**
- ➜ **Not compositional**
- ➜ **Not complete:** sometimes need auxilliary/ghost variables

$$\{x = 0\}$$
$$x := x + 1; \parallel x := x + 1$$
$$\{x = 2\}$$

$$\{x = 0 \wedge a_1 = 0 \wedge a_2 = 0\}$$

$$\begin{array}{r|l} \{a_2 = 0 \wedge x = 0 \ \vee \ a_2 = 1 \wedge x = 1\} & \{a_1 = 0 \wedge x = 0 \ \vee \ a_1 = 1 \wedge x = 1\} \\ < x := x + 1; a_1 := 1 > & < x := x + 1; a_2 := 1 > \\ \{a_2 = 0 \wedge x = 1 \ \vee \ a_2 = 1 \wedge x = 2\} & \{a_1 = 0 \wedge x = 1 \ \vee \ a_1 = 1 \wedge x = 2\} \end{array}$$

$$\{x = 2\}$$

Owicki-Gries framework

- ➜ **Quadratic explosion of proof obligations! (verification conditions)**
- ➜ **Not compositional**
- ➜ **Not complete:** sometimes need auxilliary/ghost variables

$$\{x = 0\}$$
$$x := x + 1; \parallel x := x + 1$$
$$\{x = 2\}$$

$$\{x = 0 \wedge a_1 = 0 \wedge a_2 = 0\}$$

$$\begin{array}{r|l}
\wedge a_1 = 0 \qquad\qquad & \qquad\qquad \wedge a_2 = 0 \\
\{a_2 = 0 \wedge x = 0 \ \vee \ a_2 = 1 \wedge x = 1\} & \{a_1 = 0 \wedge x = 0 \ \vee \ a_1 = 1 \wedge x = 1\} \\
< x := x + 1; a_1 := 1 > & < x := x + 1; a_2 := 1 > \\
\{a_2 = 0 \wedge x = 1 \ \vee \ a_2 = 1 \wedge x = 2\} & \{a_1 = 0 \wedge x = 1 \ \vee \ a_1 = 1 \wedge x = 2\} \\
\wedge a_1 = 1 \qquad\qquad & \qquad\qquad \wedge a_2 = 1
\end{array}$$

$$\{x = 2\}$$

Demo

Rely-Guarantee?

Intuition:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules:
  - each program is specified in isolation, **assuming a behavior of the "environment"** (other programs in parallel)

Rely-Guarantee?

Intuition:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules:
  - each program is specified in isolation, **assuming a behavior of the "environment"** (other programs in parallel)
  - each program has: precondition, postcondition, **rely and guarantee**

$$\overset{\text{c}}{\{P, R, G, Q\}} \;\Big\|\; \overset{\text{c'}}{\{P', R', G', Q'\}}$$

Rely-Guarantee?

Intuition:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules:
  - each program is specified in isolation, **assuming a behavior of the "environment"** (other programs in parallel)
  - each program has: precondition, postcondition, **rely and guarantee**
  - rely and guarantee are **relations between 2 states**

$$\underset{\{P, R, G, Q\}}{\text{c}} \;\Big\|\; \underset{\{P', R', G', Q'\}}{\text{c'}}$$

Rely-Guarantee?

Intuition:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules:
  - each program is specified in isolation, **assuming a behavior of the "environment"** (other programs in parallel)
  - each program has: precondition, postcondition, **rely and guarantee**
  - rely and guarantee are **relations between 2 states**
  - **rely** expresses the maximum behavior of the environment (the interference that the program can tolerate)

$$\underset{\{P, R, G, Q\}}{\text{c}} \quad \Big\| \quad \underset{\{P', R', G', Q'\}}{\text{c'}}$$

Rely-Guarantee?

Intuition:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules:
  - each program is specified in isolation, **assuming a behavior of the “environment”** (other programs in parallel)
  - each program has: precondition, postcondition, **rely and guarantee**
  - rely and guarantee are **relations between 2 states**
  - **rely** expresses the maximum behavior of the environment (the interference that the program can tolerate)
  - **guarantee** expresses a maximum behavior promised to the environment

$$\underset{\{P, R, G, Q\}}{c} \;\Big\|\; \underset{\{P', R', G', Q'\}}{c'}$$

Rely-Guarantee?

Intuition:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules:
  - each program is specified in isolation, **assuming a behavior of the “environment”** (other programs in parallel)
  - each program has: precondition, postcondition, **rely and guarantee**
  - rely and guarantee are **relations between 2 states**
  - **rely** expresses the maximum behavior of the environment (the interference that the program can tolerate)
  - **guarantee** expresses a maximum behavior promised to the environment

$$\underset{\{P, R, G, Q\}}{\text{c}} \;\Big\|\; \underset{\{P', R', G', Q'\}}{\text{c'}}$$

$$\underset{P}{\sigma_0} \overset{c}{\rightarrow} \sigma_1 \overset{c}{\rightarrow} \sigma_2 \underset{R}{\overset{c'}{\rightarrow}} \sigma_3 \overset{c}{\rightarrow} \sigma_4 \underset{R}{\overset{c'}{\rightarrow}} \sigma_5 \underset{R}{\overset{c'}{\rightarrow}} \sigma_6 \overset{c}{\rightarrow} \underset{Q}{\sigma_7}$$

Rely-Guarantee?

Formally:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules (examples):

Rely-Guarantee?

Formally:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules (examples):

$$\frac{P \subseteq \{s.\ f\ s \in Q\} \quad \{(s,t).\ P\ s \wedge (t = f\ s \vee t = s)\} \subseteq G \quad \mathit{stable}\ P\ R \quad \mathit{stable}\ Q\ R}{\mathit{Basic}\ f\{P, R, G, Q\}}$$

Where $\mathit{stable}\ P\ R\ =\ \forall\ \sigma\ \sigma'.\ (P\sigma \wedge R(\sigma, \sigma')) \rightarrow P\sigma'$
(*doing an environment step before or after P should not make P invalid*)

Rely-Guarantee?

Formally:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules (examples):

$$\frac{P \subseteq \{s.\ f\ s \in Q\} \quad \{(s,t).\ P\ s \wedge (t = f\ s \vee t = s)\} \subseteq G \quad \mathit{stable}\ P\ R \quad \mathit{stable}\ Q\ R}{\mathit{Basic}\ f\{P, R, G, Q\}}$$

$$\frac{c_1\{P_1, R_1, G_1, Q_1\} \quad c_2\{P_2, R_2, G_2, Q_2\} \quad G_1 \subseteq R_2 \quad G_2 \subseteq R_1}{c_1 || c_2\{P_1 \cap P_2, R_1 \cap R_2, G_1 \cup G_2, Q_1 \cap Q_2\}}$$

Where $\mathit{stable}\ P\ R\ =\ \forall\ \sigma\ \sigma'.\ (P\sigma \wedge R(\sigma, \sigma')) \rightarrow P\sigma'$
(*doing an environment step before or after P should not make P invalid*)

Intuition: the guarantee of one program is the rely of the other program

Demo

We have seen today ...

- Need for new reasoning framework for parallel/concurrent programs
- Owicki-Gries
- Rely-Guarantee