Logic for Verification 1b

Nisansala Yatapanage
ANU Logic Summer School

Concurrent Programs

The programs we've seen so far are **sequential** programs.

This means the statements are executed one after the other.

Now we're going to look at **concurrent** programs.

A concurrent program consists of a *set* of sequential processes all executing together.

Why does this make things more complicated?

Concurrent Programs

Concurrent vs. Parallel

Parallel systems – several processes (threads) each running on separate processors

- useful to make the computation faster.

Concurrent systems – several processes (threads) with interleaving statements, often sharing the same resource

- not necessarily running in parallel, e.g. it could be running on a single processor
- much more interesting behaviour
- difficult to reason about.

Concurrent Programs

Concurrency models use *interleaving* statements as if they all execute on a single processor.

thread 1 does line 1	thread 1 does line 1	thread 2 does line 1
thread 2 does line 1	thread 1 does line 2	thread 1 does line 1
thread 1 does line 2	thread 2 does line 1	thread 1 does line 2

We need to verify that the program is correct under all interleavings. Just running it again wouldn't necessarily mean that the same sequence would run.

Concurrent Programs

A *process* can have multiple *threads*. In Java, there is a concept of *threads*.

Reasoning about concurrent programs depends on the level of abstraction of the model.

An **atomic** statement cannot be interrupted by another process. What is considered atomic depends on the abstraction level of the model, e.g. is n = n + 1 atomic or not?

Sequential Example

Consider the following **sequential** program:

Initialisation	`int n = 0;`
p1	`n = 1;`
p2	`n = 2;`

What is the final value of n?

Sequential Example

Consider the following **sequential** program:

```
Initialisation   int n = 0;

p1               n = 1;
p2               n = 2;
```

There is only one possibility:
p1 executes, n = 1
p2 executes, n = 2
Final value of n is 2.

Sequential Example

The state transition diagram for the program:

$$\begin{tikzcd} \boxed{n = 0} \arrow[r, "p1"] & \boxed{n = 1} \arrow[r, "p2"] & \boxed{n = 2} \end{tikzcd}$$

Note: The states are not showing the program statements – it just looks like it in this case!

The states are showing the current values of the variables.

The program statements are what happens in between.

Sequential Example

What if there was more than one variable?

```
init1   int n = 0;
init2   int m = 1;
   p1   n = 1;
   p2   n = 2;
```

This is a state transition diagram starting at the state just after init2.

$$\begin{tikzcd} \boxed{\begin{array}{l} n = 0 \\ m = 1 \end{array}} \arrow[r, "p1"] & \boxed{\begin{array}{l} n = 1 \\ m = 1 \end{array}} \arrow[r, "p2"] & \boxed{\begin{array}{l} n = 2 \\ m = 1 \end{array}} \end{tikzcd}$$

Concurrent Example

Consider the following **concurrent** program:

```
int n = 0;
```

Process p		Process q	
p1	`n = 1;`	q1	`n = 2;`

What is the final value of n? Is it 1 or is it 2?

Concurrent Example

It could be p1 first or q1 first. We don’t know!

Trace 1:	Trace 2:
p1: n = 1	q1: n = 2
q1: n = 2	p1: n = 1
Final value for n is 2	Final value for n is 1

There’s no way to know which one it will be.

Sequential Example

Consider the following **sequential** program:

Initialisation	int n = 0;
p1	n = n + 1;
p2	n = n + 1;

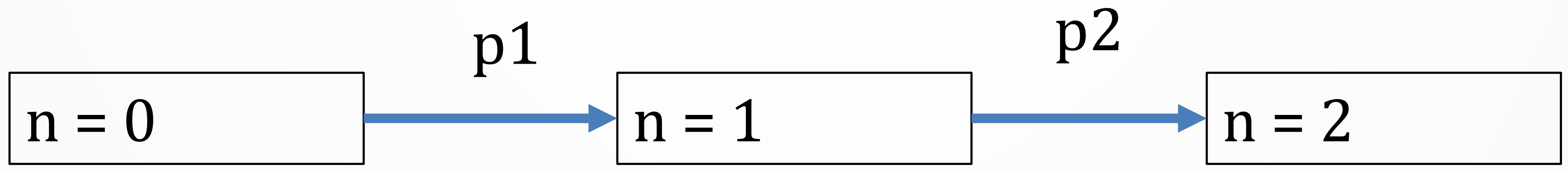
What is the final value of n?

Trace 1:
p1: n = 1
p2: n = 2

This is the only possible trace, so the final value is 2.

Sequential Example

The state transition diagram for the program:


$$\begin{tikzcd}\boxed{n = 0} \arrow[r, "p1"] & \boxed{n = 1} \arrow[r, "p2"] & \boxed{n = 2}\end{tikzcd}$$

The final answer for n is clear: n = 2.

Concurrent Example

Consider the following **concurrent** program:

```
int n = 0;
```

Process p

p1 `n = n + 1;`

Process q

q1 `n = n + 1;`

What is the final value for n?   Is it 2?

Concurrent Example

Let's assume that each statement happens in one go.

Trace 1:	Trace 2:
p1: n = 1	q1: n = 1
q1: n = 2	p1: n = 2
Final value for n = 2	Final value for n = 2

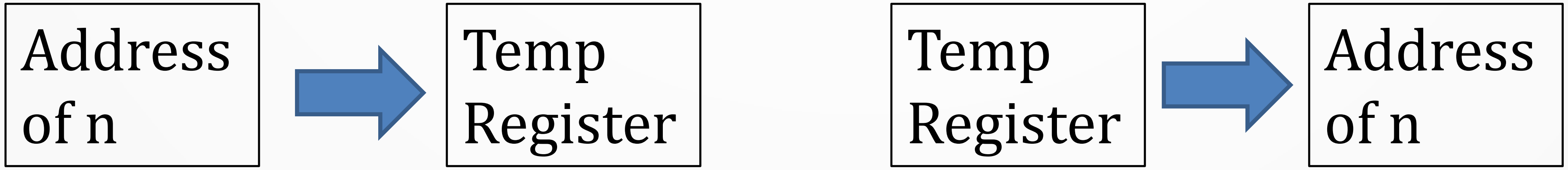
Great – both traces end up with n = 2.

Can we really assume the statement happens in one go?

Concurrent Example

When n = n + 1 executes, there are three steps:

- reading the value of n into a register (load),
- incrementing it and
- storing the result into n (store).



Concurrent Example

Here is the code at the load/store level:

```
int n = 0;
```

Process p

```
p1   load (read) n into register reg1
p2   increment register
p3   store (write) register value into n
```

Process q

```
q1   load n into reg2
q2   increment reg2
q3   store reg2 value into n
```

Let's work out the possibilities…

Concurrent Example

Trace 1:
p1: reg1 = 0, reg2 = null, n = 0
p2: reg1 = 1, reg2 = null, n = 0
p3: reg1 = 1, reg2 = null, n = 1
q1: reg1 = 1, reg2 = 1, n = 1
q2: reg1 = 1, reg2 = 2, n = 1
q3: reg1 = 1, reg2 = 2, n = 2

The final value of n is 2.

Trace 2:
p1: reg1 = 0, reg2 = null, n = 0
q1: reg1 = ?, reg2 = ?, n = ?
p2: reg1 = ?, reg2 = ?, n = ?
p3: reg1 = ?, reg2 = ?, n = ?
q2: reg1 = ?, reg2 = ?, n = ?
q3: reg1 = ?, reg2 = ?, n = ?

What is the final value?

What other traces are possible?

Concurrent Example

Another example:

```
int n = 0;
```

Process p

```
p1  int x = 2;
p2  n = x;
```

Process q

```
q1  int y = n;
q2  n = y + 1;
```

x and y are **local** variables while n is a **shared** variable.

What possible values could n have when it finishes?

Concurrent Example

Now try this example:

```
int n = 0;
```

Process p

```
p1  for (int i = 0; i < 5; i++){
p2          n = n + 1;
    }
```

Process q

```
q1  for (int i = 0; i < 5; i++){
q2          n = n + 1;
    }
```

What possible values could n have when it finishes? Just 10 like for a sequential program?   Is it possible to end up with n = 2?

This problem is taken from Ben-Ari’s concurrency book.

No bread Problem

A and B live together. A comes home and finds no bread and goes to buy bread. B comes home and finds no bread and goes to buy some. Both come home and have too much bread.

A and B want the following:

- Only one person buys bread when there is no bread.
- Someone always buys bread when there is no bread.

Problem taken from: Taubenfield G. *Synchronisation Algorithms and Concurrent Programming*, Pearson, 2006.

No bread Problem

Solution 1: Process A

```
if (no note){
    if (no bread){
        leave note;
        buy bread;
        remove note;
    }
}
```

Process B

```
if (no note){
    if (no bread){
        leave note;
        buy bread;
        remove note;
    }
}
```

Problem: They might still both buy bread.

No bread Problem

Solution 2: Process A

```
leave note A
if (no note B){
        if (no bread){
                buy bread;
        }
}
remove note A;
```

Process B

```
leave note B
if (no note A){
        if (no bread){
                buy bread;
        }
}
remove note B;
```

Problem: They might end up with no bread.

No bread Problem

Solution 3: Process A

```
leave note A
if (no note B){
        if (no bread){
                buy bread;
        }
}
remove note A;
```

Process B

```
leave note B
while(note A){
        skip;
}
if (no bread){
        buy bread;
}
remove note B;
```

Problem: They might end up with no bread.

No bread Problem

Solution 4: Use 4 notes: A1, A2, B1, B2

If A finds that there is no B1, then A buys bread.
Otherwise, if both A1 and B1 are there, the decision is made according to A2 and B2.
- If both A2 and B2 are there or neither then B buys bread; otherwise A buys bread.

No bread Problem

Solution 4:

A: Leaves A1. If B2 is there, leaves A2; otherwise removes A2. Then A checks for notes over and over as long as B1 is there and either both A2 and B2 are there or neither.
When either of the conditions fails, A checks if there is bread. If not, A buys bread and removes A1.

B: Leaves B1. If **not** A2, leaves B2; otherwise removes B2. Then B keeps checking for notes as long as A1 is there and either A2 or B2 but not both.
When either of the conditions fails, B checks if there is bread. If there is no bread, B buys bread and removes A1.

No bread Problem

	Fridge door: Empty
A comes home and leaves A1.	Fridge door: A1
A checks for B2. It’s not there so removes A2.	Fridge door: A1
A checks for B1. It’s not there so A doesn’t wait.	Fridge door: A1
A goes to buy bread.	Fridge door: A1
Meanwhile B comes home.	Fridge door: A1

No bread Problem

B leaves B1.	Fridge door: A1, B1
B checks for A2 and doesn't finds it, so leaves B2	Fridge door: A1, B1, B2
B keeps checking while there is A1 and either A2 or B2 but not both.	Fridge door: A1, B1, B2
A comes home and removes A1.	Fridge door: B1,B2
B checks if there is bread and there is, so removes B1.	Fridge door: B2

No bread Problem

The next day, B comes home and leaves B1.	Fridge door: B1, B2
B checks for "not A2". It's not there so leaves B2.	Fridge door: B1, B2
B doesn't wait because A1 is not there.	Fridge door: B1, B2
B checks if there is bread and goes to buy it.	Fridge door: B1, B2
A comes home and leaves A1.	Fridge door: A1, B1, B2

No bread Problem

A checks for B2. It’s there so A leaves A2.	Fridge door: A1, B1, B2, A2
A waits while B1 is there and both A2 and B2.	Fridge door: A1, B1, B2, A2
B comes home and removes B1	Fridge door: A1, A2, B2
A checks for bread. It’s there so removes A1.	Fridge door: A2, B2

No bread Problem

Solution 4: Process A

```
leave note A1;
if (B2){
        leave note A2;
}else{
        remove note A2;
}
while(B1 and ((A2 and B2) or
                (not A2 and not B2))
){
        skip;
}
if (no bread){
        buy bread;
}
remove note A1;
```

Process B

```
leave note B1;
if (no A2){
        leave note B2;
}else{
        remove note B2;
}
while(A1 and ((A2 and not B2) or
                (not A2 and B2)) ){
        skip;
}
if (no bread){
        buy bread;
}
remove note B1;
```

Rely/Guarantee

What is interference?

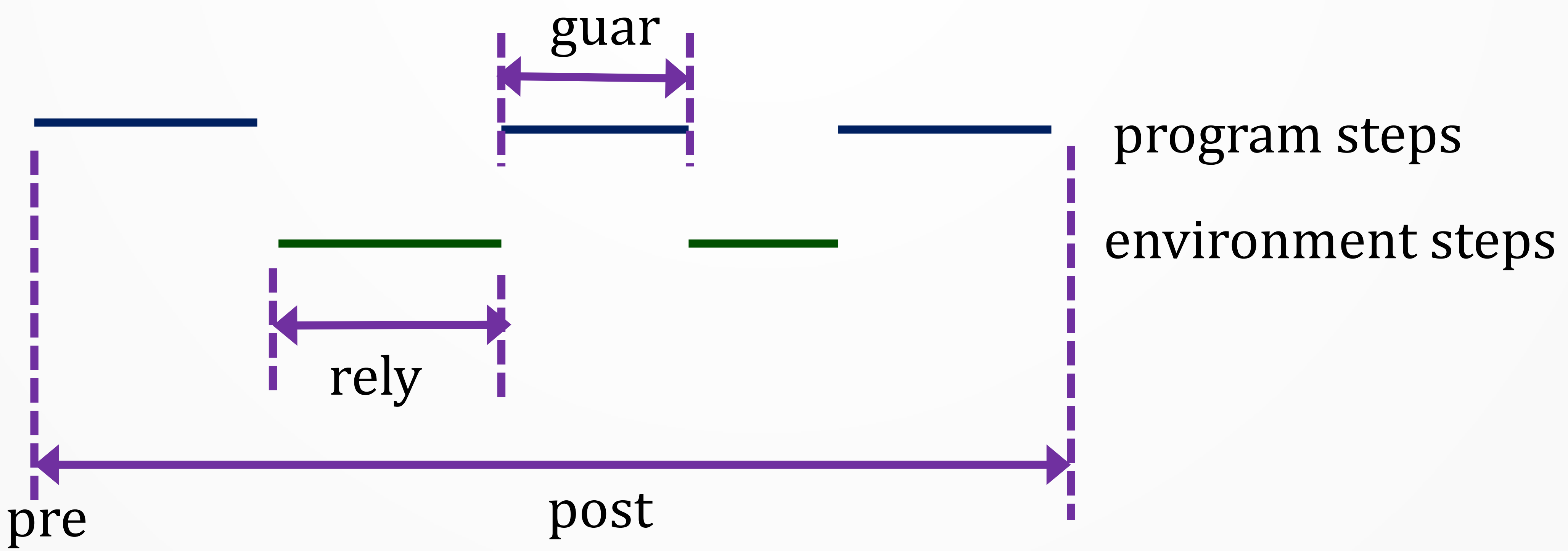
Interference occurs when one process writes to a variable at the same time as another one, or when one process reads a variable while another is writing to it.

If A is reading variable x, while B is writing a new value to it, which value will A see? The old one or the new one?

We don’t know – that’s interference.

Concurrency: Separation vs. Interference

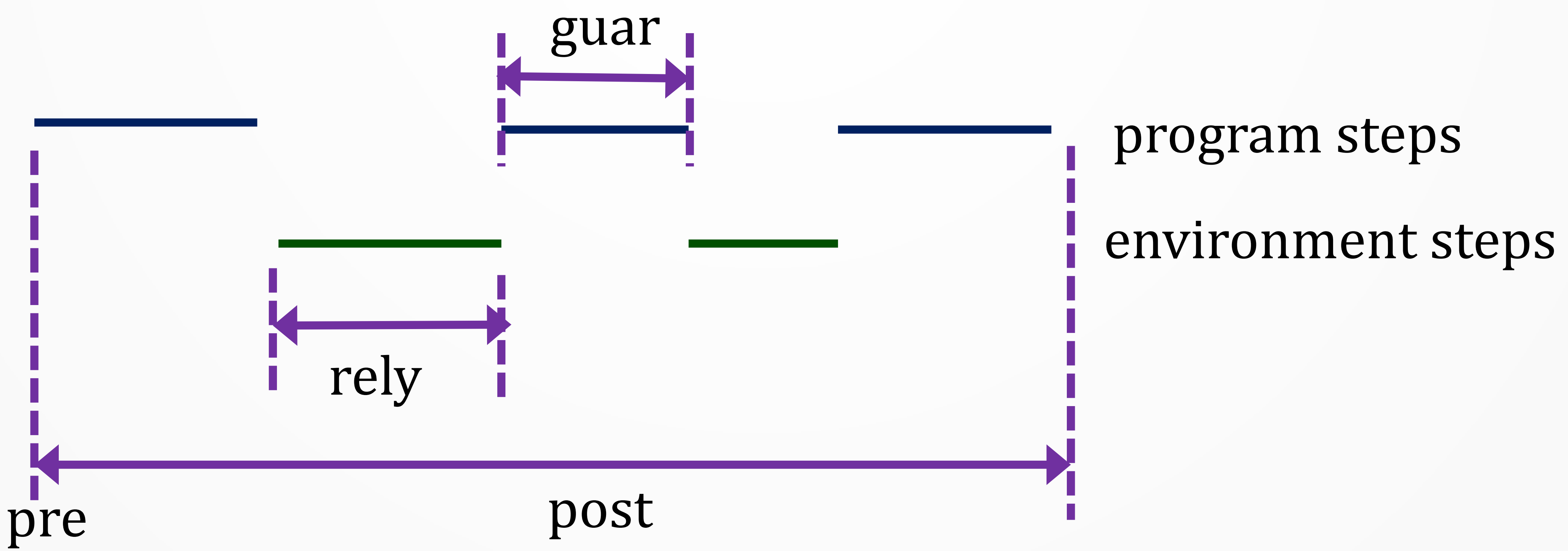
A rely/guarantee spec consists of: pre, post, rely and guar conditions.



Note that post conditions are now binary conditions, unlike Hoare logic.
x’ > x means that the new value of x is greater than the old value.

Concurrency: Separation vs. Interference

A rely/guarantee spec consists of: pre, post, rely and guar conditions.



Rely – specifies the interference that this progam can tolerate.

Guar – specifies the interference this program inflicts on the environment.

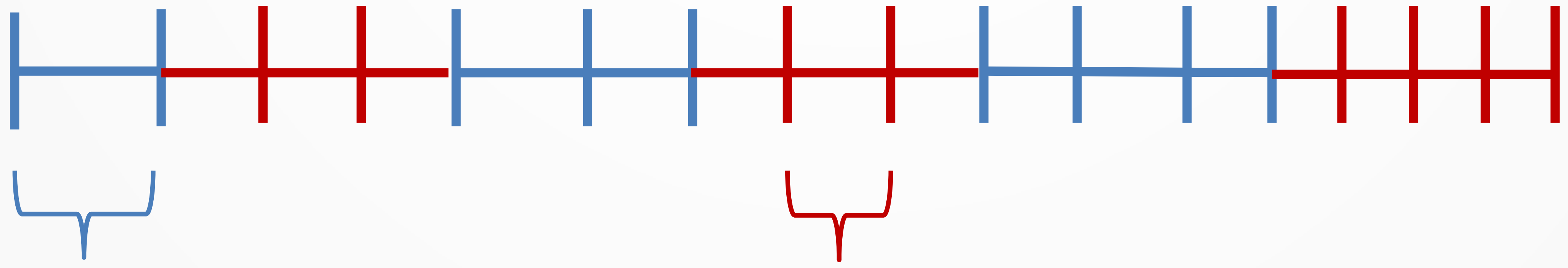
Rely/Guarantee

What's it saying exactly?

Let's look at the program on the left on the last slide.

The rely states what will happen on the red steps (the environment).

The guar states what will happen on the blue steps (this program).



The blue ones are steps made by this program.

The red ones are steps made by the other program (called the environment).

Rely/Guarantee

Recap: A **rely** tells us what the **other** program should do.
A **guar** tells us what **this** program guarantees to do.

Can the two programs below run concurrently together?

$$\begin{tikzcd} \text{Program A has:} & \text{Program B has:} \\ \text{Rely: } x' < x & \text{Rely: } y' >= y \\ \text{Guar: } y' = y \arrow[ur] & \text{Guar: } x' < x \arrow[ul] \end{tikzcd}$$

Yes, because A’s guar satisfies B’s rely and B’s guar satisfies A’s rely.

Rely/Guarantee

Recap: A **rely** tells us what the **other** program should do.
A **guar** tells us what **this** program guarantees to do.

What about these two?

$$\begin{tikzcd}
\text{Program A has:} & \text{Program B has:} \\
\text{Rely: } x' < x & \text{Rely: } y' = y \\
\text{Guar: } y' >= y \arrow[ur] & \text{Guar: } x' < x \arrow[ul]
\end{tikzcd}$$

No, because A is saying that it will either keep y the same or increase it, but B requires y to stay the same only. Therefore they can't work together!

Rely/Guarantee - History

Cliff B. Jones invented rely/guarantee in the early 80’s.

Previous approaches include the Owicki-Gries method.

The disadvantage with Owicki-Gries is that if one component changes, all the proofs have to be re-done.

Rely/guarantee is compositional. The interference is expressed as a boolean predicate.

There are many variants, e.g. assume-guarantee, deny-guarantee, variants used with separation logic, etc.

The original version is based on VDM. An algebraic representation has been developed by Ian Hayes, et al.

Rely/Guarantee - Rules

Now, instead of {P} S {Q}, we have:
{P, R} S {G, Q}

R is the rely. G is the guarantee.

R and G are both binary relations.

P and R are what this program needs from the environment.
G and Q are what this program promises to do.

The rely and guarantee conditions must be transitive.

Rely/Guarantee - Rules

The parallel rule:

$$\frac{\begin{array}{l} \{p, r_l\}\ s_l\ \{g_l, q_l\} \\ \{p, r_r\}\ s_r\ \{g_l, q_r\} \\ r \vee g_r \Rightarrow r_l \\ r \vee g_l \Rightarrow r_r \\ g_l \vee g_r \Rightarrow g \\ p \wedge q_l \wedge q_r \wedge (r \vee g_l \vee g_r)^* \Rightarrow q \end{array}}{\{p, r\}\ s_l \parallel s_r\ \{g, q\}}$$