Inference rule for Concurrent Execution

An incorrect attempt

A naive approach is to say that the concurrent execution of statements establishes postconditions of all the statements. We might try the following inference rule:

\[
\begin{align*}
\vdash P & \Rightarrow P_0 \land P_1 \\
\vdash \{P_0\} & S \{Q_0\} \\
\vdash \{P_1\} & T \{Q_1\} \\
\vdash Q_0 \land Q_1 & \Rightarrow Q
\end{align*}
\]

\[\vdash \{P\} \text{ co } S / / T \text{ oc } \{Q\} \] (Co) [Incorrect!]

It allows us to prove correct programs correct. For example:

\[
\{x = X \land y = Y\}
\]

**co**

\[
\{x = X\} \langle x := x + 1; \rangle \{x = X + 1\}
\]

//

\[
\{y = Y\} \langle y := y + 1; \rangle \{y = Y + 1\}
\]

**oc**

\[
\{x = X + 1 \land y = Y + 1\}
\]
But it also allows us to prove incorrect programs are correct!

\[\{ x = X \land y = Y \}\]

**CO**

\[\{ y = Y \}\ \langle x := y + 1; \rangle \ \{ x = Y + 1 \}\]

//

\[\{ x = X \}\ \langle y := x + 1; \rangle \ \{ y = X + 1 \}\]

**OC**

\[\{ x = Y + 1 \land y = X + 1 \}\]

**Why?** Consider the following interleaving

<table>
<thead>
<tr>
<th>0</th>
<th>{ y = Y }</th>
<th>{ x = X }</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\langle x := y + 1; \rangle</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{ x = Y + 1 }</td>
<td></td>
</tr>
</tbody>
</table>

At point 2, the precondition \( x = X \) no longer true.

The assignment \( x := y + 1 \) **interferes with** the assertion \( x = X \).

Thus the inference rule above **is not sound**.
The solution

Instead of Hoare triples, we use proof outlines. A proof outline is a triple \( \{P\}S\{Q\} \) where statement \( S \) is annotated by internal assertions. Each substatement of \( \{P\}S\{Q\} \) is preceded by an assertion. The precondition of each statement \( S \) is denoted \( \text{pre}(S) \).

Redo the logic using proof outlines instead of Hoare triples.

Assignment is as before.

\[
\begin{align*}
\frac{\vdash P \Rightarrow Q_{x \leftarrow E}}{\vdash \{P\} \ x := E \ \{Q\}} & \text{(Assign)} \\
\vdash \{P\} \ x := E \ \{Q\} & \\
\frac{\vdash \{Q\} \ T \ \{R\}}{\vdash \{P\} \ S \ \{Q\} \ T \ \{R\}} & \text{(Seq)}
\end{align*}
\]

Sequential composition requires an internal assertion

\[
\begin{align*}
\vdash \{Q\} \ S \ \{P\} & \\
\vdash \{P \land \neg E\} \Rightarrow R & \\
\vdash \{P\} \ \text{while}(E) \ \{Q\} \ S \ \{R\} & \text{(While)}
\end{align*}
\]
Now we can state a rule for concurrent composition

\[
\begin{align*}
\vdash P &\Rightarrow P_0 \land P_1 \\
\vdash \{P_0\} S \{Q_0\} \\
\vdash \{P_1\} T \{Q_1\} \\
\vdash Q_0 \land Q_1 \Rightarrow Q \\
S \text{ does not interfere with } &\{P_1\} T\{Q_1\} \\
T \text{ does not interfere with } &\{P_0\} S\{Q_0\}
\end{align*}
\]

\[
\{P\} \text{ co } \{P_0\} S\{Q_0\} /\ / \{P_1\} T\{Q_1\} \text{ oc } \{Q\} \quad \text{(Co)}
\]

**Interference**

An atomic action \( a \) interferes with an assertion \( P \) if it could cause \( P \) to change from true to false.

But \( a \) will only be executed from a state where \( \text{pre}(a) \) is true, so we may assume \( \text{pre}(a) \) is initially true.

So \( a \) **does not interfere with** \( P \) iff

\[
\vdash \{P \land \text{pre}(a)\} a \{P\}
\]

A critical assertion of \( \{P_0\} T \{Q_0\} \) is an assertion not inside an await statement.

\( S \) **does not interfere with** \( \{P_0\} T \{Q_0\} \) iff no atomic action in \( S \) interferes with any critical assertion in \( \{P_0\} T \{Q_0\} \).
Techniques for avoiding interference

Disjoint variables

if no variable in an assertion is in the write set of the action, there is no interference
[[ Need example ]]

Weakened assertions

Consider

```plaintext
## x = 0
co
## x = 0
⟨ x := x + 1; ⟩
## x = 1
/
## x = 0
⟨ x := x + 2; ⟩
## x = 2
oc
## x = 1 ∧ x = 2
```

There is interference:
\[ \forall \{ x = 0 \} \ x := x + 2; \ \{ x = 0 \} \]

We can use a weaker precondition to start each process

\[
\begin{align*}
\text{## } & x = 0 \\
\text{CO} & \\
\text{## } & x = 0 \lor x = 2 \\
\langle & x := x + 1; \rangle \\
\text{## } &? \\
\text{//} & \\
\text{## } & x = 0 \lor x = 1 \\
\langle & x := x + 2; \rangle \\
\text{## } &? \\
\text{OC} & \\
\text{## } &?
\end{align*}
\]

No interference, so far:
\[ \vdash \{ (x = 0 \lor x = 2) \land (x = 0 \lor x = 1) \} \\
\ x := x + 2 \\
\ \{ x = 0 \lor x = 2 \} \]
and

\[
\{ (x = 0 \lor x = 1) \land (x = 0 \lor x = 2) \} \\
x := x + 1 \\
\{ x = 0 \lor x = 1 \}
\]

Now complete the outline with the strongest possible postconditions, & check for interference.

---

## $x = 0$

**co**

### $x = 0 \lor x = 2$

\( x := x + 1; \)

### $x = 1 \lor x = 3$

//

### $x = 0 \lor x = 1$

\( x := x + 2; \)

### $x = 2 \lor x = 3$

**oc**

### $x = 3$

---
Global invariants

Global invariants are implied by the over-all precondition, and preserved by all atomic actions.

If $G$ is a global invariant we write

$$\begin{align*}
## P & \quad \text{for} \quad ## P \\
## \text{Global invariant } G & \\
\text{co} & \\
## L_0 & \quad a_0 \quad ## G & \quad L_0 \quad a_0 \\
## L_1 & \quad a_1 \quad ## G & \quad L_1 \\
## L_2 & \quad \text{co} \quad G & \quad L_2 \\
\text{oc} & \\
## Q & \quad \text{oc} \quad Q \\
\end{align*}$$

$\text{co}$

$$\begin{align*}
## G \land L_0 & \quad a_0 \\
## G \land L_1 & \quad a_1 \\
## G \land L_2 & \quad \text{co} \\
## G \land M_0 & \quad b_0 \\
## G \land M_1 & \quad b_1 \\
## G \land M_2 & \quad \text{oc} \\
\end{align*}$$
Now we need to check

- **Global invariance:** that $G$ is implied by $P$ and preserved by each action.

\[
P \Rightarrow G
\]
\[
\{G \land L_i\} a_i \{G\}
\]
\[
\{G \land M_i\} b_i \{G\}
\]

- **Remaining Noninterference:** the remaining parts of non-interference

\[
\{L_i \land G \land M_j\} b_j \{L_i\}
\]
\[
\{M_i \land G \land L_j\} a_i \{M_i\}
\]

- **Remaining Local Correctness:** the remaining parts of local correctness

\[
P \Rightarrow L_0 \land M_0
\]
\[
\{L_i \land G\} a_i \{L_{i+1}\}
\]
\[
\{M_i \land G\} b_i \{M_{i+1}\}
\]
\[
G \land L_2 \land M_2 \Rightarrow Q
\]

When all the local assertions $L_i$ and $M_i$ use only variables not changed by the other process, the second step is not needed (by disjoint variables): global invariance implies freedom from interference.
Example: Synchronizing loops (barrier synchronization)

Assume that $A_0$ and $A_1$ are independent of $\{c_0, c_1, s_0, s_1\}$

\[\text{## } P : c_0 = c_1 = s_0 = s_1;\]
\[\text{## global inv. } G_0 : s_0 \leq c_1 + 1\]
\[\text{## global inv. } G_1 : s_1 \leq c_0 + 1\]

\[\text{## } P_0 : s_0 = c_0 \leq c_1 \text{ while( true ) } \{\]
\[\text{## } P_0 : s_0 = c_0 \leq c_1\]
\[s_0 += 1;\]
\[\text{## } Q_0 : s_0 = c_0 + 1\]
\[A_0\]
\[\text{## } Q_0 : s_0 = c_0 + 1\]
\[c_0 += 1;\]
\[\text{## } R_0 : s_0 = c_0\]
\[\langle \text{await}(c_0 \leq c_1) \rangle\]
\[\}\]

\[\text{## } P_1 : s_1 = c_1 \leq c_0 \text{ while( true ) } \{\]
\[\text{## } P_1 : s_1 = c_1 \leq c_0\]
\[s_1 += 1;\]
\[\text{## } Q_1 : s_1 = c_1 + 1\]
\[A_1\]
\[\text{## } Q_1 : s_1 = c_1 + 1\]
\[c_1 += 1;\]
\[\text{## } R_1 : s_1 = c_1\]
\[\langle \text{await}(c_1 \leq c_0) \rangle\]
\[\}\]

Let $G = G_0 \land G_1$. What we need to show is:
Global invariance (dv means the proof is by disjoint variables)

- \( P \Rightarrow G0 \)
- \( \{ G \land P0 \} \quad s0 += 1; \quad \{ G0 \} \)
- \( \{ G \land Q0 \} \quad c0 += 1; \quad \{ G0 \} \) (dv)
- \( \{ G \land P1 \} \quad s1 += 1; \quad \{ G0 \} \) (dv)
- \( \{ G \land Q1 \} \quad c1 += 1; \quad \{ G0 \} \)

Remaining Noninterference

- \( \{ P0 \land G \land P1 \} \quad s1 += 1; \quad \{ P0 \} \) (dv)
- \( \{ P0 \land G \land Q1 \} \quad c1 += 1; \quad \{ P0 \} \)
- \( \{ Q0 \land G \land P1 \} \quad s1 += 1; \quad \{ Q0 \} \) (dv)
- \( \{ Q0 \land G \land Q1 \} \quad c1 += 1; \quad \{ Q0 \} \) (dv)
- \( \{ R0 \land G \land P1 \} \quad s1 += 1; \quad \{ R0 \} \) (dv)
- \( \{ R0 \land G \land Q1 \} \quad c1 += 1; \quad \{ R0 \} \) (dv)

Remaining local correctness

- \( P \Rightarrow P0 \)
- \( \{ G \land P0 \} \quad s0 += 1; \quad \{ Q0 \} \)
- \( \{ G \land Q0 \} \quad c1 += 1; \quad \{ R0 \} \)
- \( \{ G \land R0 \} \quad (\text{await} (c0 \leq c1)) \quad \{ P0 \} \)

And symmetrically for postconditions \( G1, P1, Q1, R1 \).
Ghost Variables

Ghost variables (aka thought variables, dummy variables, and auxiliary variables) are variables that are used for the purpose of proof, but do not need to be implemented.

Consider

```co
## x = 0

co
## x = 0
⟨ x := x + 1; ⟩
## x = 1

//
## x = 0
⟨ x := x + 1; ⟩
## x = 1

oc
## x = 1
```

Again there is interference. Note that weakening preconditions to \( \{x = 0 \lor x = 1\} \) is to no avail.
Introduce integer ghost variables $a$ and $b$, initially 0.

```plaintext
## int a := 0, b := 0 ;
## x = 0 ∧ a = 0 ∧ b = 0
## Global Inv: a + b = x

co
## a = 0
⟨ x := x + 1; a := a + 1; ⟩
## a = 1

//
## b = 0
⟨ x := x + 1; b := b + 1; ⟩
## b = 1

oc
## x = 2
```

Since $a$ and $b$ are each only in the write set of one process, there is no interference.

That $x = 2$ finally, follows from the global invariant, together with $a = 1 ∧ b = 1$. 
Await statements

Await statements force a delay until an assertion is true before proceeding.

\[
\vdash \{ P \wedge E \} \mathcal{S} \{ Q \} \\
\vdash \{ P \} \langle \text{await}(E) \mathcal{S} \rangle \{ Q \}
\]

(Await)

Two techniques:
1. ‘Hide’ assertions via mutual exclusion.
2. Strengthen the precondition via conditional synchronization.

Hide assertions

Derived inference rule

\[
\vdash \{ P \} \mathcal{S} \{ Q \} \\
\vdash \{ P \} \langle \mathcal{S} \rangle \{ Q \}
\]

(Mutual Exclusion)
On the left the global invariant is interfered with.

```plaintext
int size := 0;
int front := 0;
int back := 0;
## Global Inv:
## size = back - front
c
...  
  ⟨front := front +1;⟩
  ⟨size := size - 1 ;⟩
...  
oc
```

On the right the intermediate state is hidden in the atomic action.

```plaintext
int size := 0;
int front := 0;
int back := 0;
## Global Inv:
## size = back - front
c
...  
  ⟨front := front +1;⟩
  ## size = back - front - 1
  size := size - 1 ;⟩
...  
oc
```
Use conditional synchronization

Derived inference rule

\[
\frac{P \land E \Rightarrow Q}{\{P\} \langle \text{await}(E) \rangle \{Q\}} \quad \text{(Conditional synchronization)}
\]

Example: On the left, \( s := s - 1 \) does not respect the global invariant.

\[ \neg \{s \geq 0\} \quad s := s - 1; \quad \{s \geq 0\} \]

\[
\begin{align*}
\text{int} &\ s := 0; \\
\text{## Global Inv:} &\ s \geq 0 \\
\text{co} &\ ... \langle s := s - 1; \rangle ... \\
// &\ ... \langle s := s + 1; \rangle ...
\end{align*}
\]

\[
\begin{align*}
\text{oc} &\ \cdots
\end{align*}
\]

Solution. Use conditional synchronization to strengthen the precondition to \( s > 0 \).
**Data Refinement**

*Introducing one set of variable to represent another.*

We do data refinement in three steps:

- **Augment**: Add new variables and an invariant establishing their relationship to the preexisting variables.
- **Transform**: Change the algorithm to use the new variables rather than certain preexisting variables.
- **Diminish**: Demote any variables no longer needed to the status of ghosts.
Example of data refinement
Recall the producer and consumer with a shared buffer.

```plaintext
int buf
int p := 0; # The number of things produced.
int c := 0; # The number of things consumed.
## Global inv: 0 ≤ c ≤ p ≤ c + 1
```

```plaintext
process Producer {
  while (true) {
    ⟨await(p = c)⟩
    ## p = c
    buf := next value ;
    p := p + 1; }
}

process Consumer {
  while (true) {
    ⟨await(p > c)⟩
    ## p = c + 1
    use buf
    c := c + 1; }
}
```
Augment with a boolean $b$.

$b$ says whether $p = c$ or $p = c + 1$

---

```plaintext
int buf;
bool b := true;
int p := 0, c := 0;
## Global inv: $0 \leq c \leq p \leq c + 1$
## Global inv: $b = (p = c)$
```

---

```plaintext
process Producer {
    while (true) {
        ⟨await($p = c$)⟩
        ## $(p = c) \land b$
        buf := next value;
        ⟨$p, b := p + 1, false;$⟩
    }
}
```

```plaintext
process Consumer {
    while (true) {
        ⟨await($p > c$)⟩
        ## $(p = c + 1) \land b$
        use buf
        ⟨$c, b := c + 1, true;$⟩
    }
}
```
Transform
Rewrite so that \( p \) and \( c \) are no longer needed to compute the result.

```c
int buf;
bool b := true ;
int p := 0, c := 0;
## Global inv: 0 \leq c \leq p \leq c + 1
## Global inv: b = (p = c)
```
Diminish
Demote $p$ and $c$ to the status of ghost variables.

```
int buf;
bool b := true;
## int p := 0, c := 0;
## Global inv: $0 \leq c \leq p \leq c + 1$
## Global inv: $b = (p = c)$
```

```
process Producer {
    while (true) {
        ⟨await(b);⟩
        ## $(p = c) \land b$
        buf := next value ;
        ⟨p, b := p + 1, false;⟩
    }
}
```

```
process Consumer {
    while (true) {
        ⟨await( not b);⟩
        ## $(p = c + 1) \land \neg b$
        use buf
        ⟨c, b := c + 1, true;⟩
    }
}
```

Now $p$ and $c$ are used only in the reasoning.
Safety Properties

A property characterizes a set of executions. A program satisfies a property if every possible execution (history) of the program is in the set characterized by the property.

Safety property: Something must always be true (set of executions in which no undesirable states, or sequences of states, occur).

- partial correctness — program never enters a state that is both terminated and not described by the postcondition.
- absence of deadlock (doesn’t reach a deadlock state)
- mutual exclusion

- finitely refutable: if a safety property does not hold, there is a finite history that demonstrates this.
- characterized by negation of ‘bad’ things
Proving Safety

Let $B$ characterize undesirable states

- Show that for any critical assertion $C$, $C \Rightarrow \neg B$, or
- Show that $\neg B$ is a global invariant.
  - $\neg B$ is true initially,
  - $\{pre(S) \land \neg B\}$ $S$ $\{\neg B\}$ is valid for all program statements $S$

Special Case: Exclusion of configurations

```
co # process 1
    ... { P } ⟨a⟩ ...
// # process 2
    ... { Q } ⟨b⟩ ...
oc
```

If
- $P$ and $Q$ are not interfered with, and
- $P \land Q \equiv false$ (i.e. $\neg P \lor \neg Q$)

then statements $a$ and $b$ can never both be about to be executed.
Liveness Properties

Something must eventually become true.

- e.g.,
  - termination: process must eventually stop
  - absence of starvation (processes must eventually get serviced)

- not finitely refutable: any execution can be extended to satisfy the property.
Fairness

Fairness assumptions are assumptions about the nature of the scheduler.
Often some fairness assumption is required in order for (liveness) properties to be provable.
An atomic action is eligible if it could be executed next
scheduling policy — determines which eligible action will be executed next.

```plaintext
bool continue = true;
co
    while (continue) skip
//
    continue := false;
oc
```

Degrees of fairness:

**unconditional:** Every unconditional atomic action that is eligible is executed eventually.

**weak:** Unconditionally fair & every conditional atomic action for which the condition is continuously true (until it is executed), will eventually be executed.

**strong:** Unconditionally fair & every conditional atomic action for which the condition is true infinitely often, will eventually be executed.

```plaintext
bool continue := true, try := false ;

co

   while (continue) {
       try := true ;
       try := false ;
   }

//

   ⟨ await( try ) continue := false ; ⟩

oc
```

Under weak fairness, the above may not terminate.
Under strong fairness, it must terminate eventually.

Use fairness:

Often to show liveness properties, one must make use of fairness assumptions.