Bilateral Proofs of Concurrent Programs

Jayadev Misra

Department of Computer Science
University of Texas at Austin

WG 2.3, Jan 2016
Pasadena
Agonizing Reappraisal

- Is it realistic to prove concurrent programs in practice?
- Need to prove only tightly-coupled programs?
  Can they be handled through model-checking?
- Could loose-coupled concurrency become the norm, say through mobile computing?
Agonizing Reappraisal

- Is it realistic to prove concurrent programs in practice?
- Need to prove only tightly-coupled programs?
  Can they be handled through model-checking?
- Could loose-coupled concurrency become the norm, say through mobile computing?
Agonizing Reappraisal

- Is it realistic to prove concurrent programs in practice?

- Need to prove only tightly-coupled programs?
  Can they be handled through model-checking?

- Could loose-coupled concurrency become the norm, say through mobile computing?
Status of Program Design and Verification in Four Decades

- Astounding gains for sequential programming.

- Vast improvement in understanding of concurrent programming.

  - Theory and practice lag considerably for the latter, compared to the former.

  - Very small concurrent programs proved manually, occasionally.

  - Larger concurrent programs proved using model checking. Only bright spot.
Status of Program Design and Verification in Four Decades

- Astounding gains for sequential programming.

- Vast improvement in understanding of concurrent programming.

- Theory and practice lag considerably for the latter, compared to the former.

- Very small concurrent programs proved manually, occasionally.

- Larger concurrent programs proved using model checking. Only bright spot.
Why sequential Programs are more amenable

- Hoare’s Proof Theory: Program specification by pre- and postcondition.
  
- Permits verification of sequential program code for a given specification.
  
- **Proof rules**: permit composition of the component specifications, for hierarchical construction.
  
- Specification used in program construction, instead of source code.
  
- Concurrent programming lacks a theory of composable specification. Pre- and postcondition do not compose for concurrent programs.
  
- Needed: a scalable theory of composable specification of concurrent programs.
Why sequential Programs are more amenable

- Hoare’s Proof Theory: Program specification by **pre- and postcondition**.

- Permits verification of sequential program code for a given specification.

- **Proof rules**: permit composition of the component specifications, for hierarchical construction.

- Specification used in program construction, instead of source code.

- Concurrent programming lacks a theory of **composable specification**. Pre- and postcondition do not compose for concurrent programs.

- Needed: a scalable theory of composable specification of concurrent programs.
Motivation for the current work:
Commutative, Associative Fold of a bag

- Bag \( u \).
  Commutative, associative binary operator \( \oplus \)
  Write fold of \( u \) as \( \Sigma u \).

- Problem: Replace all elements of \( u \) by \( \Sigma u \).

- Strategy: Define \( f_k \) that transforms \( u \)
  - reduces the size of \( u \) by \( k \), and
  - the resulting bag has the same fold as the original bag.
An Orc Program

\[ f_1 = \text{get}(x); \text{get}(y); \text{put}(x \oplus y) \]

\[ f_k = f_1 \mathbin{\square} f_{k-1}, \quad k > 1 \]

Given that \( u \) has \( n \) items initially, \( n > 1 \), apply \( f_{n-1} \).

- Safety: Finally \( u \) has one item, the fold of the original items. Easy.
- Progress: Program terminates. Hard.

The result does not hold for \( f_n \). There is deadlock.

- No known proof technique for this program.
Observations about the problem

- Desired: Respect the recursive program structure in proof.
- Note interplay between sequential and concurrent aspects.
- Entire code is not available.
Another very difficult program to prove

\[
x = 0
\]

\[
x := x + 1 \quad x := x + 2
\]

\[
x = 3
\]
Owicki’s Thesis

• Construct annotation of each sequential component.

\[
\begin{align*}
\{ x = 0 \} \\
( \{ x = 0 \lor x = 2 \} & \ x := x + 1 \ \{ x = 1 \lor x = 3 \} ) \\
\Box \ \{ x = 0 \lor x = 1 \} \ x := x + 2 \ \{ x = 2 \lor x = 3 \} ) \\
\{ ( x = 1 \lor x = 3 ) \land ( x = 2 \lor x = 3 ) \} \\
\{ x = 3 \}
\end{align*}
\]

• Show: **proofs** don’t interfere, e.g.,
given assertions valid in concurrent execution

\[
\begin{align*}
\{ ( x = 0 \lor x = 2 ) \land ( x = 0 \lor x = 1 ) \} \ x := x + 2 \ \{ x = 0 \lor x = 2 \} \\
\{ ( x = 0 \lor x = 1 ) \land ( x = 0 \lor x = 2 ) \} \ x := x + 1 \ \{ x = 0 \lor x = 1 \} \\
\ldots
\end{align*}
\]
Owicki’s Thesis

• Construct annotation of each sequential component.

\[ \{ x = 0 \} \]

\[
\begin{align*}
& \quad \{ x = 0 \lor x = 2 \} \ x := x + 1 \ \{ x = 1 \lor x = 3 \} \\
& \quad [] \ \{ x = 0 \lor x = 1 \} \ x := x + 2 \ \{ x = 2 \lor x = 3 \} \\
& \quad \{ (x = 1 \lor x = 3) \land (x = 2 \lor x = 3) \} \\
& \quad \{ x = 3 \}
\end{align*}
\]

• Show: proofs don’t interfere, e.g.,
given assertions valid in concurrent execution

\[
\begin{align*}
& \quad \{ (x = 0 \lor x = 2) \land (x = 0 \lor x = 1) \} \ x := x + 2 \ \{ x = 0 \lor x = 2 \} \\
& \quad \{ (x = 0 \lor x = 1) \land (x = 0 \lor x = 2) \} \ x := x + 1 \ \{ x = 0 \lor x = 1 \} \\
& \quad \ldots
\end{align*}
\]
Assessment

- First real proof technique for concurrent programs.
- Works well for small tightly-coupled components.
- Not scalable.
- Needs program code.
- No notion of a specification.
Rely-Guarantee of Cliff Jones

- Replace non-interference proofs by checks against stable predicates.
- Hoare-like proof rule.
- Limited to safety properties.
Unity by Chandy and Misra

- Simplify program structure: \( \text{loop } \langle g \rightarrow s \rangle \ [\] \text{loop } \langle g' \rightarrow s' \rangle \ [\] \cdots \)

- Each \( \langle g \rightarrow s \rangle \) is a guarded action.

- Prove program properties, not assertions at program points:
  - A resource is never granted unless requested.
  - A request for a resource is eventually granted.

- Specification is a set of properties.

- Composition rules for specification are given.

The guard holds as a precondition in concurrent execution.
Limitations of the Unity approach

- Does not support traditional program structure.
- Auxiliary variables needed to capture program control points.
Current Theory: Specification

- **Terminal** property: postcondition of a program for a given precondition.

- **Perpetual** property: holds throughout every program execution. Similar to invariant.
  
  - (Safety) once it requests a resource the thread waits until the resource is granted,
  
  - (Progress) once the resource is granted the thread will eventually release it.

- **Specification**: Terminal and Perpetual properties.
Summary of the approach

- Create program annotation as before, but with restrictions.

- Annotations are valid even under concurrent execution. As in UNITY.

- **Bilateral**: Derive terminal and perpetual properties from annotations. And conversely.

- Composition rules for specifications.
Program Model

- **command**: Uninterruptible, terminating code, e.g.: $x := x - 1$, *put* on a channel.

- **action**: Guarded command, $b \rightarrow \alpha$, e.g.: $x > 0 \rightarrow x := x - 1$, or
  get from a channel, where the guard is implicit.

- $f, g :: \text{component}$: action $| f [] g | \text{seq} (f_0, f_1, \cdots f_n)$

- **program**: component executing alone.
Programming Constructs

• seq: Any sequential programming construct that has a proof rule, e.g.:
  \[ s; t \]
  \[ \text{if } b \text{ then } s \text{ else } t \]
  \[ \text{while } b \text{ do } s \]

• Join: \( f \parallel g \) is commutative, associative.

• Arbitrary hierarchy of sequential and concurrent constructs:
  \( (f \parallel g); (f' \parallel g') \)
Program Execution

- Sequential components follow their execution rules.
- Join: starts all components simultaneously. Terminates when they all do.
- Program control may reside at multiple program points simultaneously.
- At any moment the action at some control point is executed.
- Every control point is chosen eventually for execution.
Action Execution

- Execution of $b \rightarrow \alpha$ always terminates, either **effectively** or **ineffectively**.

- **Effective** execution:
  - $b$ is true and $\alpha$ is executed to completion.
  - Program control moves past the action.

- **Ineffective** execution:
  - $b$ is **false**.
  - Program control remains before the action.

- Evaluation of $b$ is uninterruptible in all cases.

- If $b$ is true: $\alpha$ is executed immediately.
Example: Distributed counter

Program \( f = \bigcup_j f_j \) implements counter \( ctr \).

Initially \( ctr = 0 \)

\( f_j :: \)

Initially \( old_j, new_j = 0, 0 \)

Loop

\( new_j := old_j + 1; \)

If \( \left[ \begin{array}{c}
ctr = old_j \rightarrow ctr := new_j \\
ctr \neq old_j \rightarrow old_j := ctr
\end{array} \right] \)

Forever

Show:

Safety: \( ctr \) is changed only by incrementation (increased by 1).

Progress: \( ctr \) is changed eventually.
Example: Distributed counter

Program $f = \prod_j f_j$ implements counter $ctr$.

**initially** $ctr = 0$

$f_j ::$

**initially** $old_j, new_j = 0, 0$

**loop**

$\quad new_j := old_j + 1;$

$\quad$ if $\left[ ctr = old_j \rightarrow ctr := new_j \\
\quad \mid ctr \neq old_j \rightarrow old_j := ctr \right]$  

**forever**

Show:

Safety: $ctr$ is changed only by incrementation (increased by 1).

Progress: $ctr$ is changed eventually.
Inviolable preconditions of actions

- Find precondition $p$ of each action so that $p$ remains true as long as control remains at the action.

\[
\left( \{ x = 0 \lor x = 2 \} \ x := x + 1 \ \{ x = 1 \lor x = 3 \} \right.
\]
\[
\left[ \{ x = 0 \lor x = 1 \} \ x := x + 2 \ \{ x = 2 \lor x = 3 \} \right)
\]
\[
\{ (x = 1 \lor x = 3) \land (x = 2 \lor x = 3) \}
\]

- Owicki: Check that precondition can not be violated by any concurrent action.

- Unity: Programmer specifies guards for each action.

- In the current theory:
  Unknown concurrent environment.
  General programs: Guards are usually too weak.
  Control flow carries additional information.
Inviolable preconditions of actions

- Find precondition $p$ of each action so that $p$ remains true as long as control remains at the action.

\[
( \{ x = 0 \lor x = 2 \} \ x := x + 1 \ \{ x = 1 \lor x = 3 \} \\
\Box \ \{ x = 0 \lor x = 1 \} \ x := x + 2 \ \{ x = 2 \lor x = 3 \} ) \\
\{ (x = 1 \lor x = 3) \land (x = 2 \lor x = 3) \}
\]

- Owicki: Check that precondition can not be violated by any concurrent action.

- Unity: Programmer specifies guards for each action.

- In the current theory:
  Unknown concurrent environment.
  General programs: Guards are usually too weak.
  Control flow carries additional information.
Inviolable preconditions of actions

- Find precondition $p$ of each action so that $p$ remains true as long as control remains at the action.

\[
\begin{align*}
( \{ x = 0 \lor x = 2 \} & \ x := x + 1 \ \{ x = 1 \lor x = 3 \} \\
\mathbf{[]} \ \{ x = 0 \lor x = 1 \} & \ x := x + 2 \ \{ x = 2 \lor x = 3 \} \\
\{ (x = 1 \lor x = 3) \land (x = 2 \lor x = 3) \}
\end{align*}
\]

- Owicki: Check that precondition can not be violated by any concurrent action.

- Unity: Programmer specifies guards for each action.

- In the current theory:
  Unknown concurrent environment.
  General programs: Guards are usually too weak.
  Control flow carries additional information.
Inviolable preconditions of actions

- Find precondition \( p \) of each action so that \( p \) remains true as long as control remains at the action.

\[
\begin{align*}
( \{ x = 0 \lor x = 2 \} & \ x := x + 1 \ {\{ x = 1 \lor x = 3 \} } \\
[] \{ x = 0 \lor x = 1 \} & \ x := x + 2 \ {\{ x = 2 \lor x = 3 \} } \\
& \{(x = 1 \lor x = 3) \land (x = 2 \lor x = 3)\}
\end{align*}
\]

- Owicki: Check that precondition can not be violated by any concurrent action.

- Unity: Programmer specifies guards for each action.

In the current theory:
- Unknown concurrent environment.
- General programs: Guards are usually too weak.
- Control flow carries additional information.
Access rights to variables

- $x$ local to $f$: $f$ has exclusive write-access to $x$ during any execution.

- $p$ local predicate of $f$: every variable in $p$ is local to $f$. 
Local Annotation

- Annotation of a program in which all predicates are local to the component in which they appear.

- Assert: Given local annotation in which \( \{p\} \ b \rightarrow \alpha \), \( p \) holds whenever \( b \rightarrow \alpha \) is executed.

- Construct local annotation using Hoare-proof rules for seq construct.

- For join, use:

\[
\begin{align*}
\{r\} f \{s\} \\
\{r'\} g \{s'\}
\end{align*}
\]

\[
\frac{r \land r'}{f \mid g \{s \land s'\}}
\]
Local Annotation: Distributed Counter

\[ f_j :: \]

- **initially** \( old_j, \ new_j = 0, 0 \)
- \( \{true\} \)

**loop**

- \( \{true\} \)
  - \( \alpha_j :: \ new_j := old_j + 1; \)
  - \( \{new_j = old_j + 1\} \)
  - **if** [ \( \beta_j :: \ new_j = old_j + 1 \) \( ctr = old_j \rightarrow ctr := new_j \) \{true\}]
  - \( \gamma_j :: \ new_j = old_j + 1 \) \( ctr \neq old_j \rightarrow old_j := ctr \) \{true\}

- \( \{true\} \)

**forever**
Safety Property  $\text{co}$

- $p \ co \ q$ in component $f$:  
  Effective execution of any action of $f$ in a $p$-state achieves a $q$-state.

- In program $f$: once $p$ holds it continues to hold until $q$ is established.

- As a composition rule: 
  $p \ co \ q$ holds in $f$ if it holds in every component of $f$. 
Formal definition of co

\[
\{r\} f \{s\}
\]

For every action \( b \rightarrow \alpha \) with precondition \( \text{pre} \) in the annotation of \( f \) :

\[
\{\text{pre} \land b \land p\} \alpha \{q\}
\]

\[
\{r\} f \{p \textbf{co} q \mid s\}
\]
Special cases of \textit{co}

- **stable** $p$: Once $p$ holds, it continues to hold:
  
  $p \text{ co } p$

- **constant** $e$: Value of expression $e$ never changes:
  
  $(\forall c :: \text{stable } e = c)$

- **invariant** $p$: $p$ always holds:
  
  initially $p$ and stable $p$
Distributed Counter, contd.

\[ f_j :: \]

\textbf{initially} \quad \text{old}_j, \text{new}_j = 0, 0

\{true\}

\textbf{loop}

\{true\}

\[ \alpha_j :: \quad \text{new}_j := \text{old}_j + 1; \]

\{new\_j = old\_j + 1\}

\textbf{if}\ [ \quad \beta_j :: \quad \{\text{new}_j = old\_j + 1\} \quad ctr = old\_j \rightarrow ctr := new\_j \quad \{true\} \]

\textbf{if}\ [ \quad \gamma_j :: \quad \{\text{new}_j = old\_j + 1\} \quad ctr \neq old\_j \rightarrow old\_j := ctr \quad \{true\}] \]

\{true\}

\textbf{forever}
Safety: $ctr$’s value is only incremented

- Show: $ctr = m$ co $ctr = m \lor ctr = m + 1$ in $f$
  prove: $ctr = m$ co $ctr = m \lor ctr = m + 1$ holds in all $f_j$.

- For each action $b \rightarrow \alpha$ with precondition $pre$, show:
  \[
  \{\text{pre} \land b \land ctr = m\} \alpha \{ctr = m \lor ctr = m + 1\}
  \]

- Only $\beta_j$ may change the value of $ctr$. So, prove:
  \[
  \{ctr = m \land new_j = old_j + 1 \land ctr = old_j\}
  
  ctr := new_j
  
  \{ctr = m \lor ctr = m + 1\} \]
Progress Properties

  \[
  \text{transient } p: \ p \text{ is false eventually. } \square \diamond \neg p.
  \]

- Ensures: \( p \text{ en } q \)
  Once \( p \) holds, it continues to hold until \( q \) holds; and \( q \) holds eventually.
  More useful in practice.
  Defined using transient.

- Leads-to: \( p \leftrightarrow q \)
  once \( p \) holds, \( q \) holds eventually.
  Typical property in a specification.
  Defined using ensures.
Progress Properties

  
  \text{transient } p: \ p \text{ is false eventually. } \Box \Diamond \neg p.

- Ensures: \text{p en } q
  Once \ p \text{ holds, it continues to hold until } q \text{ holds; and } q \text{ holds eventually.}
  More useful in practice.
  Defined using transient.

- Leads-to: \text{p } \Rightarrow q
  once \ p \text{ holds, } q \text{ holds eventually.}
  Typical property in a specification.
  Defined using ensures.
Progress Properties

• Transient: Fundamental property. Compositional.
  
  \textbf{transient } p: \ p \text{ is false eventually. } \square \Diamond \neg p.

• Ensures: \ p \text{ en } q
  Once \ p \text{ holds, it continues to hold until } q \text{ holds; and } q \text{ holds eventually.}
  More useful in practice.
  Defined using transient.

• Leads-to: \ p \leftrightarrow q
  once \ p \text{ holds, } q \text{ holds eventually.}
  Typical property in a specification.
  Defined using ensures.
Simplistic Definition of \textbf{transient} $p$ in $f$:

$p$ is false eventually in $f$

- Each action of $f$ is effectively executed if $p$ is a precondition, and
- its execution establishes $\neg p$.

For every action $b \rightarrow \alpha$ of $f$ with precondition $pre$:

\[
\begin{align*}
pre \land p & \Rightarrow b \\
\{pre \land p\} & \alpha \{\neg p\} \\
\{\} f & \{ \text{transient } p \mid \}
\end{align*}
\]
Stronger Rules for \( p \)

- \( f ; g \): either \( f \) terminates or \( p \) transient in \( f \) AND \( p \) transient in \( g \).
  
  Sufficient: \( f \) terminates AND \( p \) transient in \( g \).

- \( f \parallel g \): \( p \) transient in \( f \) or \( g \).

- Inheritance: If \( p \) transient in ALL components of \( f \), \( p \) transient in \( f \).
Stronger Rules for transient $p$

- $f; g$: either $f$ terminates or $p$ transient in $f$ AND $p$ transient in $g$.
  Sufficient: $f$ terminates AND $p$ transient in $g$.

- $f □ g$: $p$ transient in $f$ or $g$.

- Inheritance: If $p$ transient in ALL components of $f$, $p$ transient in $f$. 
Stronger Rules for **transient** $p$

- $f; g$: either $f$ terminates or $p$ transient in $f$ AND $p$ transient in $g$.
  
  Sufficient: $f$ terminates AND $p$ transient in $g$.

- $f \parallel g$: $p$ transient in $f$ or $g$.

- Inheritance: If $p$ transient in ALL components of $f$, $p$ transient in $f$. 
Ensures: $p \text{ en } q$

Once $p$ holds, it continues to hold until $q$ holds; and $q$ holds eventually.

- $p \land \neg q \text{ co } p \lor q$
- transient $p \land \neg q$
Distributed Counter

• Prove: \( ctr \) increases eventually.

• Can not be proved as an ensures property.

• Prove:

  In every step, either \( ctr \) increases, or
  the number of \( \text{old}_j \) that differ from \( ctr \) decreases.

• \( nb \): number of \( \text{old}_j \) such that \( ctr \neq \text{old}_j \).

\[
ctr = m \land nb = N \quad \text{en} \quad nb < N \lor ctr > m \quad \text{in } f \\
\text{(E)}
\]
Proof strategy

\[
\text{ctr} = m \land \text{nb} = N \quad \text{en} \quad \text{nb} < N \lor \text{ctr} > m \text{ in } f
\] (E)

- To prove (E) in \([j \cdot f_j]\): Prove (E) in each \(f_j\).

- To prove (E) in initialization; loop \(body_j\) forever: Since initialization terminates, show (E) in: loop \(body_j\) forever.

- To prove (E) in loop \(body_j\) forever: using inheritance prove (E) in \(body_j\),

- To prove (E) in \(body_j\), i.e., \(\text{new}_j := \text{old}_j + 1; \text{if } [\beta_j \mid \gamma_j]\): Since \(\text{new}_j := \text{old}_j + 1\) terminates, prove (E) in if \([\beta_j \mid \gamma_j]\),

- To prove (E) in if \([\beta_j \mid \gamma_j]\): prove (E) in \(\beta_j\) and \(\gamma_j\), i.e., Effective executions of \(\beta_j\) and \(\gamma_j\) establish the postcondition of (E) given its pre-condition.
Proof strategy

\[ ctr = m \land nb = N \quad \text{en} \quad nb < N \lor ctr > m \quad \text{in} \quad f \]  

(E)

- To prove (E) in \( []_j f_j \): Prove (E) in each \( f_j \).

- To prove (E) in \textit{initialization}; loop \textit{body}_j \; \text{forever}:
  Since \textit{initialization} terminates, show (E) in: \quad \text{loop} \; \textit{body}_j \; \text{forever}.

- To prove (E) in \text{loop} \; \textit{body}_j \; \text{forever}:
  using inheritance prove (E) in \textit{body}_j, .

- To prove (E) in \textit{body}_j, i.e., \textit{new}_j := \textit{old}_j + 1; \texttt{if} \ [\beta_j | \gamma_j]::
  Since \textit{new}_j := \textit{old}_j + 1 terminates, prove (E) in \texttt{if} \ [\beta_j | \gamma_j],

- To prove (E) in \texttt{if} \ [\beta_j | \gamma_j]: prove (E) in \beta_j \; \text{and} \; \gamma_j, \ i.e.,
  Effective executions of \beta_j \; \text{and} \; \gamma_j \; \text{establish the postcondition of (E)}
  given its pre-condition.
Proof strategy

\[ \text{ctr} = m \land \text{nb} = N \quad \text{en} \quad \text{nb} < N \lor \text{ctr} > m \text{ in } f \quad \] (E)

- To prove (E) in \[ \text{[]}_j f_j \]: Prove (E) in each \( f_j \).

- To prove (E) in \text{initialization}; \text{loop } body_j \text{ forever}:
  
  Since \text{initialization} terminates, show (E) in: \text{loop } body_j \text{ forever}.

- To prove (E) in \text{loop } body_j \text{ forever}:
  
  using inheritance prove (E) in \text{body}_j, .

- To prove (E) in \text{body}_j, i.e., \text{new}_j := \text{old}_j + 1; \text{if } [\beta_j \mid \gamma_j]:
  
  Since \text{new}_j := \text{old}_j + 1 terminates, prove (E) in \text{if } [\beta_j \mid \gamma_j],

- To prove (E) in \text{if } [\beta_j \mid \gamma_j]: \text{prove (E) in } \beta_j \text{ and } \gamma_j, \text{ i.e.,}
  
  Effective executions of \beta_j \text{ and } \gamma_j establish the postcondition of (E) given its pre-condition.
Proof strategy

\[ ctr = m \land nb = N \quad \text{en} \quad nb < N \lor ctr > m \quad \text{in} \quad f \quad \text{(E)} \]

- To prove (E) in \[\{j.f_j\] : Prove (E) in each \(f_j\).

- To prove (E) in \textit{initialization}; loop \(body_j\) forever:
  Since \textit{initialization} terminates, show (E) in: \(\text{loop } body_j\) forever.

- To prove (E) in \textbf{loop} \(body_j\) \textbf{forever}:
  using inheritance prove (E) in \(body_j\).

- To prove (E) in \(body_j\), i.e., \(new_j := old_j + 1; \textbf{if} \ [\beta_j | \gamma_j]\):
  Since \(new_j := old_j + 1\) terminates, prove (E) in \(\textbf{if} \ [\beta_j | \gamma_j]\),

- To prove (E) in \(\textbf{if} \ [\beta_j | \gamma_j]\): prove (E) in \(\beta_j\) and \(\gamma_j\), i.e.,
  Effective executions of \(\beta_j\) and \(\gamma_j\) establish the postcondition of (E) given its pre-condition.
Proof strategy

\[ ctr = m \land nb = N \quad \text{en} \quad nb < N \lor ctr > m \text{ in } f \]  

(E)

- To prove \((E)\) in \([;] f_j\): Prove \((E)\) in each \(f_j\).

- To prove \((E)\) in \textit{initialization}; \textit{loop } body_j \textit{ forever}:
  Since \textit{initialization} terminates, show \((E)\) in: \textit{loop } body_j \textit{ forever}.

- To prove \((E)\) in \textit{loop } body_j \textit{ forever}:
  using inheritance prove \((E)\) in \textit{body}_j, .

- To prove \((E)\) in \textit{body}_j, i.e., \textit{new}_j := \textit{old}_j + 1; \textit{if } \textit{[} \beta_j \mid \gamma_j \textit{]}:
  Since \textit{new}_j := \textit{old}_j + 1 terminates, prove \((E)\) in \textit{if } \textit{[} \beta_j \mid \gamma_j \textit{]},

- To prove \((E)\) in \textit{if } \textit{[} \beta_j \mid \gamma_j \textit{]}: prove \((E)\) in \textit{β}_j and \textit{γ}_j, i.e.,
  Effective executions of \textit{β}_j and \textit{γ}_j establish the postcondition of \((E)\) given its pre-condition.
Proof Obligations

Relevant Annotation of $f_j$:

\[
\begin{align*}
\text{if} \quad & \beta_j :: \{new_j = old_j + 1\} \rightarrow ctr = old_j \rightarrow ctr := new_j \quad \{\text{true}\} \\
\text{\quad |} \quad & \gamma_j :: \{new_j = old_j + 1\} \rightarrow ctr \neq old_j \rightarrow old_j := ctr \quad \{\text{true}\} \\
\{\text{true}\}
\end{align*}
\]

Proof Obligations:

\[
\begin{align*}
\beta_j :: \{ctr = m \land nb = N \land new_j = old_j + 1 \land ctr = old_j\} \\
\quad & ctr := new_j \\
\quad & \{nb < N \lor ctr > m\}
\end{align*}
\]

\[
\begin{align*}
\gamma_j :: \{ctr = m \land nb = N \land new_j = old_j + 1 \land ctr \neq old_j\} \\
\quad & old_j := ctr \\
\quad & \{nb < N \lor ctr > m\}
\end{align*}
\]
Leads-to

$p \leadsto q$: once $p$ holds, $q$ holds eventually.

- (basis) $\dfrac{p \text{ en } q}{p \leadsto q}$

- (transitivity) $\dfrac{p \leadsto q, q \leadsto r}{p \leadsto r}$

- (disjunction) For any (finite or infinite) set of predicates $S$
  $\dfrac{\left( \forall p: p \in S : p \leadsto q \right)}{(\forall p : p \in S : p) \leadsto q}$
Derived Rules: What makes Proofs Practical. For $\text{co}$

\[
\begin{align*}
\text{false} & \text{ co } q \\
p & \text{ co } \text{true} \\
\frac{p \text{ co } q \text{, } p' \text{ co } q'}{p \land p' \text{ co } q \land q'} & \text{(CONJUNCTION)} \\
\frac{p \text{ co } q \text{, } p' \text{ co } q'}{p \lor p' \text{ co } q \lor q'} & \text{(DISJUNCTION)} \\
\frac{p \text{ co } q}{p \land p' \text{ co } q} & \text{(LHS STRENGTHENING)} \\
\frac{p \text{ co } q}{p \text{ co } q \lor q'} & \text{(RHS WEAKENING)}
\end{align*}
\]
Lightweight Derived Rules for $\leftrightarrow$

1. (implication) \[
\frac{p \Rightarrow q}{p \iff q}
\]

2. (lhs strengthening, rhs weakening) \[
\frac{p \iff q}{p' \land p \iff q} \quad \frac{p \iff q \lor q'}{p \iff q \lor q'}
\]

3. (cancellation) \[
\frac{p \iff q \lor r \quad r \iff s}{p \iff q \lor s}
\]
Heavyweight Derived Rules for \( \leftrightarrow \)

1. (PSP) \[
p \leftrightarrow q \\
\text{stable } p' \\
p \land p' \leftrightarrow q \land p'
\]

2. (induction) \[
M : \text{Program States} \rightarrow W. (W, \prec) \text{ well-founded.} \\
\forall m :: p \land M = m \leftrightarrow (p \land M \prec m) \lor q \\
p \leftrightarrow q
\]

3. (completion) \( p_i \) and \( q_i \) are predicates; \( i \) index over a finite set. \[
\forall i :: \\
p_i \leftrightarrow q_i \lor b \\
q_i \co q_i \lor b \\
\) \[
\forall i :: p_i \leftrightarrow (\forall i :: q_i) \lor b
\]
Heavyweight Derived Rules for $\mapsto$ 

1. (PSP)  
\[
p \mapsto q \\
\text{stable } p' \\
p \wedge p' \mapsto q \wedge p'
\]

2. (induction)  
\[
M : \text{Program States} \rightarrow W. \ (W, \prec) \text{ well-founded.}
\]
\[
(\forall m :: p \wedge M = m \mapsto (p \wedge M \prec m) \vee q) \\
p \mapsto q
\]

3. (completion)  
\[
p_i \text{ and } q_i \text{ are predicates; } i \text{ index over a finite set.}
\]
\[
(\forall i :: \\
p_i \mapsto q_i \lor b \\
q_i \text{ co } q_i \lor b \\
) \\
(\forall i :: p_i) \mapsto (\forall i :: q_i) \lor b
\]
Heavyweight Derived Rules for \( \iff \)

1. (PSP) \[
\begin{align*}
p & \iff q \\
\text{stable } p' \\
p \land p' & \iff q \land p'
\end{align*}
\]

2. (induction) \[
M : \text{Program States} \rightarrow W. \ (W, \prec) \text{ well-founded.}
\]
\[
(\forall m :: p \land M = m \iff (p \land M \prec m) \lor q) \\
p \iff q
\]

3. (completion) \( p_i \) and \( q_i \) are predicates; \( i \) index over a finite set.
\[
(\forall i :: \\
p_i \iff q_i \lor b \\
q_i \text{ co } q_i \lor b \\
) \\
(\forall i :: p_i) \iff (\forall i :: q_i) \lor b
\]
Heavyweight Derived Rules for \( \mapsto \mapsto \)

1. (PSP) \[
p \mapsto q \\
{\text{stable } p'} \\
p \land p' \mapsto q \land p'
\]

2. (induction) \[
M: \text{Program States } \rightarrow W. (W, \prec) \text{ well-founded.}
\]
\[
(\forall m :: p \land M = m \mapsto (p \land M \prec m) \lor q) \\
p \mapsto q
\]

3. (completion) \( p_i \) and \( q_i \) are predicates; \( i \) index over a finite set.
\[
(\forall i :: \\
p_i \mapsto q_i \lor b \\
q_i \co q_i \lor b
\]
\[
(\forall i :: p_i) \mapsto (\forall i :: q_i) \lor b
\]
Distributed Counter

• Prove in $f$: $ctr$ increases unboundedly:

  $true \implies ctr > C$, for any integer $C$

• Proved in $f$: $ctr = m \land nb = N \land en \land nb < N \lor ctr > m$

• Use definition of $\rightarrow$ and its derived rules for the proof.
Distributed Counter, Contd.

\[
ctr = m \land nb = N \quad \text{en} \quad nb < N \lor ctr > m
\]

proven

\[
ctr = m \land nb = N \quad \iff \quad nb < N \lor ctr > m
\]

basis rule of leads-to

\[
ctr = m \land nb = N \quad \iff \quad ctr = m \land nb < N \lor ctr > m
\]

PSP with \( ctr = m \) co \( ctr = m \lor ctr = m + 1 \)
Distributed Counter, Contd.

\[ ctr = m \land nb = N \quad \text{en} \quad nb < N \lor ctr > m \]
proven

\[ ctr = m \land nb = N \iff nb < N \lor ctr > m \]
basis rule of \textit{leads-to}

\[ ctr = m \land nb = N \iff ctr = m \land nb < N \lor ctr > m \]
PSP with \[ ctr = m \quad \text{co} \quad ctr = m \lor ctr = m + 1 \]
Distributed Counter, Contd.

\[ ctr = m \land nb = N \quad \text{en} \quad nb < N \lor ctr > m \]

proven

\[ ctr = m \land nb = N \quad \leftrightarrow \quad nb < N \lor ctr > m \]

basis rule of *leads-to*

\[ ctr = m \land nb = N \quad \leftrightarrow \quad ctr = m \land nb < N \lor ctr > m \]

PSP with  \( ctr = m \)  co  \( ctr = m \lor ctr = m + 1 \)
Apply Induction Rule

\[ ctr = m \land nb = N \iff ctr = m \land nb < N \lor ctr > m \]

Induction rule:

\[
(\forall m :: p \land M = m \iff (p \land M \prec m) \lor q) \\
p \iff q
\]

Use \( nb \) for \( M \) and \( \prec \) for \( \prec \) to conclude:

\[ ctr = m \iff ctr > m \]
Distributed Counter, Contd.

\[ ctr = m \land nb = N \implies \text{en } nb < N \lor ctr > m \]
proven

\[ ctr = m \land nb = N \iff nb < N \lor ctr > m \]
basis rule of \textit{leads-to}

\[ ctr = m \land nb = N \text{ en } ctr = m \land nb < N \lor ctr > m \]
PSP with \( ctr = m \) co \( ctr = m \lor ctr = m + 1 \)

\[ ctr = m \iff ctr > m \]
Induction rule; well-founded order \( \prec \) over natural numbers

\[ true \iff ctr > C, \text{ for any integer } C \]
Induction rule, well-founded order \( \prec \) over natural numbers.
Distributed Counter, Contd.

\[ ctr = m \land nb = N \quad \text{en} \quad nb < N \lor ctr > m \]
proven

\[ ctr = m \land nb = N \rightarrow nb < N \lor ctr > m \]
basis rule of \textit{leads-to}

\[ ctr = m \land nb = N \quad \text{en} \quad ctr = m \land nb < N \lor ctr > m \]
PSP with \( ctr = m \lor ctr = m + 1 \)

\[ ctr = m \rightarrow ctr > m \]
Induction rule; well-founded order \(<\) over natural numbers

\[ true \rightarrow ctr > C, \text{ for any integer } C \]
Induction rule, well-founded order \(<\) over natural numbers.