Bilateral Proofs of Concurrent Programs: A simple and neat solution to a complex problem

Jayadev Misra

Department of Computer Science
University of Texas at Austin

Nov 10 and 17, 2015
University of Texas
A Quote from H. L. Mencken, American Essayist, 1930s

For every complex problem there is a solution that is simple, neat and wrong.
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. Bright spot.
Distinction: Sequential and Concurrent Programs

• 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 theory of composable specification that scales up and be automated.
Distinction: Sequential and Concurrent Programs

- 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 theory of composable specification that scales up and be automated.
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

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

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

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

\[
\begin{align*}
  \{ x = 0 \} \\
  x &:= x + 1 \quad [] \quad x := x + 2 \\
  \{ x = 3 \}
\end{align*}
\]
Owicki’s Thesis

• Construct annotation of each sequential component.

\[
\{ x = 0 \}
\]

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

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

\[
\{( x = 1 \lor x = 3 ) \land ( x = 2 \lor x = 3 ) \}
\]

\[
\{ x = 3 \}
\]

• Show that the proofs don’t interfere, e.g.,

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

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

\[
\ldots
\]
Owicki’s Thesis

- Construct annotation of each sequential component.

\[
\{ x = 0 \}
\]

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

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

\[
\{ (x = 1 \lor x = 3) \land (x = 2 \lor x = 3) \}
\]

\[
\{ x = 3 \}
\]

- Show that the proofs don’t interfere, e.g.,

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

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

...
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: $\text{loop } \langle g \rightarrow s \rangle \cdot \text{loop } \langle g' \rightarrow s' \rangle \cdot \cdots$

- 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.
Implementations

- Some successes: Telephony, Control systems

- Model checkers:
  UV (Markus Kaltenbach, UT),
  Murφ (David Dill, Stanford),
  Siemens (Jorge Cuellar),
  SAL

- Implementations in other logics:
  Boyer-Moore prover, Larch, HOL, Coq, Isabelle/ZF
  DisCo (based on Unity) in PVS
  CommUNITY workbench
Limitations of the Unity approach

- Does not support traditional program structure.
- Auxiliary variables needed to capture program control points.
- Termination and deadlock equated.
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.

• Use the annotations to derive terminal and perpetual properties.

  Bilateral

• 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.: get from a channel.

- **$f, g :: component$**: action $| f \parallel g | seq (f_0, f_1, \cdots f_n)$

- **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: \( f \shove {\cdot} g \) is commutative, associative.

• A sequential construct may combine concurrent programs:
  \[
  (f \shove {\cdot} g); (f' \shove {\cdot} g')
  \]
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 $f = \square_j f_j$ implements counter $ctr$.

Initially $ctr = 0$

$f_j ::$

Initially $old_j, new_j = 0, 0$

Loop

$new_j := old_j + 1;$

If $[ ctr = old_j \rightarrow ctr := new_j$

$| ctr \neq old_j \rightarrow old_j := ctr ]$

Forever

Show:

Safety: $ctr$ is changed only by incrementation.

Progress: $ctr$ is changed eventually.
Example: Distributed counter

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

**initially** $ctr = 0$

$f_j ::$

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

**loop**

$new_j := old_j + 1$;

if $[ ctr = old_j \rightarrow ctr := new_j$

$\mid ctr \neq old_j \rightarrow old_j := ctr ]$

forever

Show:
Safety: $ctr$ is changed only by incrementation.

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.

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

- 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.

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

- 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.
  
  $$(\{x = 0 \lor x = 2\} x := x + 1 \{x = 1 \lor x = 3\})$$
  $$(\{x = 0 \lor x = 1\} x := x + 2 \{x = 2 \lor x = 3\})$$
  $$(\{x = 1 \lor x = 3\} \land (x = 2 \lor x = 3))$$

- 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.

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

• 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.

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

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

- For join, use:

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

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

\[ f_j :: \]

**initially** \( \text{old}_j, \text{new}_j = 0, 0 \)

\{true\}

loop

\{true\}

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

\{new_j = old_j + 1\}

**if** [\[ \beta_j :: \{new_j = old_j + 1\} \text{ctr} = old_j \rightarrow \text{ctr} := \text{new}_j \{true\} \]

\| \[ \gamma_j :: \{new_j = old_j + 1\} \text{ctr} \neq old_j \rightarrow \text{old}_j := \text{ctr} \{true\}\]

\{true\}

forever
Safety Property  \( co \)

- \( p \ co \ q \) 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 $\text{co}$

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

$$\{r\} f \{s\} \quad \{pre \land b \land p\} \alpha \{q\} \quad \{r\} f \{p \text{ co } q \mid s\}$$
Special cases of 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.

Prove: $\text{ctr} = m \land \text{ctr} = m \lor \text{ctr} = m + 1$.

$f_j ::$

initially $old_j, new_j = 0, 0$
\{true\}

loop
\{true\}

$\alpha_j :: new_j := old_j + 1;$
\{new_j = old_j + 1\}

if $[ β_j :: \{new_j = old_j + 1\} \text{ctr} = old_j \rightarrow \text{ctr} := new_j \ \{true\}$

$| γ_j :: \{new_j = old_j + 1\} \text{ctr} \neq old_j \rightarrow old_j := \text{ctr} \ \{true\}]$

\{true\}

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:
  \[ \{ pre \land b \land ctr = m \} \alpha \{ ctr = m \lor ctr = m + 1 \} \]

• 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

- **Transient**: Fundamental property. Compositional.
  
  \[ \text{transient } p: \text{ } p \text{ will be false eventually. } \square \lozenge \neg p. \]

  
  \[ p \text{ en } q: \]
  once \( p \) holds, it continues to hold until \( q \) holds; and \( q \) holds eventually.

  
  \[ p \mapsto q: \text{ once } p \text{ holds, } q \text{ holds eventually.} \]
Progress Properties

  \[\text{transient } p: \ p \text{ will be false eventually. } \square \Diamond \neg p.\]

  \[p \text{ en } q:\]
  once \( p \) holds, it continues to hold until \( q \) holds; and \( q \) holds eventually.

  \[p \leftrightarrow q:\] once \( p \) holds, \( q \) holds eventually.
Progress Properties

  
  \textbf{transient } p: p \text{ will be false eventually. } \square \diamond \neg p.

  
  \textbf{p \text{ en } q}: \text{ once } p \text{ holds, it continues to hold until } q \text{ holds; and } q \text{ holds eventually.}

  
  \textbf{p } \mapsto \text{ q}: \text{ once } p \text{ holds, } q \text{ holds eventually.}
Simplistic Definition of transient $p$ in $f$:
$p$ will be 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 $pre$:

$$pre \land p \Rightarrow b$$

$$\{pre \land p\} \alpha \{\neg p\}$$

$$\{ \} f \{ \text{transient } p \mid \}$$
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 \( f \).
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 $f$. 
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.

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

- Prove: $ctr$ increases eventually.

- This is not an ensures property.

- Prove:

  In every step, either $ctr$ increases, or the number of $old_j$ that differ from $ctr$ decreases.

- $nb$: number of $old_j$ such that $ctr \neq old_j$.

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

(E)
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 \([j \cdot f_j]\): 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: Prove (E) in \(body_j\), using inheritance.

- To prove (E) in \(body_j\), i.e., new\(j\) := old\(j\) + 1; if \([\beta_j \mid \gamma_j]\): Prove (E) in If\(j\), since new\(j\) := old\(j\) + 1 terminates.

- 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

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

(E)

- To prove (E) in \[[0] f_j\]: 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: Prove (E) in \( body_j \), using inheritance.

- To prove (E) in \( body_j \), i.e., \( new_j := old_j + 1; \text{ if } [\beta_j \mid \gamma_j] \):
  Prove (E) in \( If_j \), since \( new_j := old_j + 1 \) terminates.

- 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

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

(E)

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

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

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

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

- To prove (E) in 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

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

(E)

- To prove (E) in $\left[ j \right] f_j$: 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: Prove (E) in $body_j$, using inheritance.

- To prove (E) in $body_j$, i.e., $new_j := old_j + 1$; if $[\beta_j | \gamma_j]$:
  Prove (E) in $If_j$, since $new_j := old_j + 1$ terminates.

- To prove (E) in 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

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

- To prove (E) in \( \llbracket j \rrbracket f_j \) : Prove (E) in each \( f_j \).

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

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

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

- To prove (E) in \textit{if} \quad [\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 Obligations

Relevant Annotation of $f_j$:

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

Proof Obligations:

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

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

\[ p \leadsto q: \text{ once } p \text{ holds, } q \text{ holds eventually.} \]

- **(basis)** \[ \frac{p \en q}{p \leadsto q} \]

- **(transitivity)** \[ \frac{p \leadsto q, q \leadsto r}{p \leadsto r} \]

- **(disjunction)** For any (finite or infinite) set of predicates \( S \)

\[ (\forall p: p \in S: p \leadsto q) \]
\[ (\forall p: p \in S: p) \leadsto q \]
Derived Rules: What makes Proofs Practical. For $\text{co}$

- $false \text{ co } q$
- $p \text{ co } true$
- $\frac{p \text{ co } q, p' \text{ co } q'}{p \land p' \text{ co } q \land q'}$ (CONJUNCTION)
- $\frac{p \text{ co } q, p' \text{ co } q'}{p \lor p' \text{ co } q \lor q'}$ (DISJUNCTION)
- $\frac{p \text{ co } q}{p \land p' \text{ co } q}$ (LHS STRENGTHENING)
- $\frac{p \text{ co } q}{p \text{ co } q \lor q'}$ (RHS WEAKENING)
Lightweight Derived Rules for $\leftrightarrow$

1. (implication) $\quad \frac{p \Rightarrow q}{p \iff q}$

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

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

1. (PSP)

\[
p \rightarrow q \\
\text{stable } p' \\
p \land p' \rightarrow q \land p'
\]

2. (induction) $M : \text{Program States} \rightarrow W$. $(W, \prec)$ well-founded.

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

3. (completion) $p_i$ and $q_i$ are predicates; $i$ index over a finite set.

\[
(\forall i :: \\
p_i \rightarrow q_i \lor b \\
q_i \co q_i \lor b \\
) \\
(\forall i :: p_i) \rightarrow (\forall i :: q_i) \lor b
\]
Heavyweight Derived Rules for \( \implies \)

1. (PSP) \( p \implies q \)

\[
\begin{array}{c}
\text{stable } p' \\
\hline
p \land p' \implies q \land p'
\end{array}
\]

2. (induction) \( M : \) Program States \( \rightarrow \) \( W. \) \( (W, \prec) \) well-founded.

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

3. (completion) \( p_i \) and \( q_i \) are predicates; \( i \) index over a finite set.

\[
(\forall i :: \\
p_i \implies q_i \lor b \\
q_i \lor b)
\]

\[
(\forall i :: p_i) \implies (\forall i :: q_i) \lor b
\]
Heavyweight Derived Rules for $\Rightarrow$

1. **(PSP)**

   $p \Rightarrow q$

   \[
   \begin{array}{c}
   \text{stable} \ p' \\
   \hline
   p \land p' \Rightarrow q \land p'
   \end{array}
   \]

2. **(induction)**

   $M : \text{Program States} \rightarrow W$. $(W, \prec)$ well-founded.

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

3. **(completion)**

   $p_i$ and $q_i$ are predicates; $i$ index over a finite set.

   \[
   (\forall i :: p_i \Rightarrow q_i \lor b)
   \]

   \[
   q_i \text{ co } q_i \lor b
   \]

   \[
   (\forall i :: p_i) \Rightarrow (\forall i :: q_i) \lor b
   \]
Heavyweight Derived Rules for $\rightarrow$

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

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

3. (completion) $p_i$ and $q_i$ are predicates; $i$ index over a finite set.
\[
\left(\forall i ::
\begin{align*}
p_i & \rightarrow q_i \lor b \\
q_i & \text{co } q_i \lor b
\end{align*}
\right)
\]
\[
\left(\forall i :: p_i\right) \rightarrow \left(\forall i :: q_i\right) \lor b
\]
Distributed Counter

- Prove in $f$: $ctr$ increases unboundedly:
  \[\text{true} \rightarrow ctr > C, \text{ for any integer } C\]

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

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

\( ctr = m \land nb = N \)  \text{en}  \ nb < N \lor ctr > m \\
proven

\( ctr = m \land nb = N \iff nb < N \lor ctr > m \)  \\
basis rule of \ leads-to

\( ctr = m \land nb = N \iff ctr = m \land nb < N \lor ctr > m \)  \\
PSP with \( ctr = m \)  \text{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

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

\[ ctr = m \land nb = N \quad \leftrightarrow \quad ctr = m \land nb < N \lor ctr > m \]
PSP with \( ctr = m \) co \( ctr = m \lor ctr = m + 1 \)
Distributed Counter, Contd.

\[ ctr = m \land nb = N \] en \( nb < N \lor ctr > m \)
proven

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

\[ ctr = m \land nb = N \iff ctr = m \land nb < N \lor ctr > m \]
PSP with \( ctr = m \oco ctr = m \lor ctr = m + 1 \)
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.

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

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

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

\[ \text{ctr} = m \quad \Leftrightarrow \quad \text{ctr} > m \]
Induction rule; well-founded order \( < \) over natural numbers

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

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

\[
ctr = m \land nb = N \quad \leftrightarrow \quad nb < N \lor ctr > m
\]
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 \(ctr = m\) co \(ctr = m \lor ctr = m + 1\)

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

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