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 Σu.

- Problem: Replace all elements of u by Σu.

- 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

$$f_1 \quad = \quad get(x);\ get(y);\ put(x \oplus y)$$

$$f_k \quad = \quad f_1 \ [\!]\ f_{k-1}, \qquad k > 1$$

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

$$\{x = 0\}$$

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

$$\{x = 3\}$$

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.

$$\{x = 0\}$$

$$(\ \{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\}$$

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

$$\{(x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)\}$$

$$\{x = 3\}$$

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

$$\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\}\ \ x := x + 2\ \ \{x = 0 \vee x = 2\}$$
$$\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\}\ \ x := x + 1\ \ \{x = 0 \vee x = 1\}$$
$$\ldots$$

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: $loop\ \langle g \rightarrow s\rangle\ []\ loop\ \langle g' \rightarrow s'\rangle\ []\ \cdots$
- 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.

Implementations

- Some succeses: 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 $|$ $f \,[]\, g$ $|$ 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.:

  $s;\ t$
  if b **then** s **else** t
  while b **do** s

- Join: $f\ []\ g$ is commutative, associative.

- A sequential construct may combine concurrent programs:

  $(f\ []\ g);\ (f'\ []\ g')$

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 α 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: α is executed immediately.

Example: Distributed counter

Program $f = []_j\, f_j$ 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.

Example: Distributed counter

Program $f = []_j \, f_j$ 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.

Inviolable preconditions of actions

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

$$(\{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\}$$
$$[]\ \{x = 0 \vee x = 1\}\ x := x + 2\ \{x = 2 \vee x = 3\})$$
$$\{(x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)\}$$

- 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.

$$
\begin{array}{l}
(\ \{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\} \\
[]\ \ \{x = 0 \vee x = 1\}\ x := x + 2\ \{x = 2 \vee x = 3\}) \\
\{(x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)\}
\end{array}
$$

- 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.

$$
\begin{array}{c}
(\ \{x = 0 \lor x = 2\}\ x := x + 1\ \{x = 1 \lor x = 3\} \\
[]\ \ \{x = 0 \lor x = 1\}\ x := x + 2\ \{x = 2 \lor x = 3\}) \\
\{(x = 1 \lor x = 3) \land (x = 2 \lor x = 3)\}
\end{array}
$$

- 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.

$$
\begin{array}{c}
(\ \{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\} \\
[]\ \ \{x = 0 \vee x = 1\}\ x := x + 2\ \{x = 2 \vee x = 3\}) \\
\{(x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)\}
\end{array}
$$

- 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 $\{p\}\ b \rightarrow \alpha$, p holds whenever $b \rightarrow \alpha$ is executed.

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

- For join, use:

$$\frac{\{r\}\ f\ \{s\} \qquad \{r'\}\ g\ \{s'\}}{\{r \wedge r'\}\ f \mathbin{[]} g\ \{s \wedge s'\}}$$

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 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

$$\frac{\begin{array}{c}\{r\}\ f\ \{s\} \\ \text{For every action } b \to \alpha \text{ with precondition } pre \text{ in any annotation of } f: \\ \{pre \wedge b \wedge p\}\ \alpha\ \{q\}\end{array}}{\{r\}\ f\ \{p \ \mathbf{co}\ q \mid s\}}$$

Special cases of co

- stable p: Once p holds, it continues to hold:

  p 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: $ctr = m$ co $ctr = m \lor ctr = m + 1$.

f_j ::

initially $old_j,\ new_j = 0,\ 0$

$\{true\}$

loop

$\{true\}$

α_j :: $new_j := old_j + 1;$

$\{new_j = old_j + 1\}$

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

$\mid \gamma_j :: \ \{new_j = old_j + 1\}\ ctr \neq old_j \ \rightarrow\ old_j := ctr \ \ \{true\}]$

$\{true\}$

forever

Safety: *ctr*'s value is only incremented

- Show: $ctr = m \;\; \text{co} \;\; ctr = m \vee ctr = m+1$ in f
  prove: $ctr = m \;\; \text{co} \;\; ctr = m \vee ctr = m+1$ holds in all f_j.

- For each action $b \rightarrow \alpha$ with precondition pre, show:

$$\{pre \wedge b \wedge ctr = m\} \; \alpha \; \{ctr = m \vee ctr = m+1\}$$

- Only β_j may change the value of ctr. So, prove:

$$\begin{array}{c} \{ctr = m \wedge new_j = old_j + 1 \wedge ctr = old_j\} \\ ctr := new_j \\ \{ctr = m \vee ctr = m+1\} \end{array}$$

Progress Properties

- Transient: Fundamental property. Compositional.

  transient p: p will be false eventually. $\Box\Diamond\neg p$.

- Ensures: More useful in practice. Compositional.
  Defined using transient.

  p en q:
  once p holds, it continues to hold until q holds; and q holds eventually.

- Leads-to: Typical property in a specification. Non-compositional.
  Defined using ensures.

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

Progress Properties

- Transient: Fundamental property. Compositional.

  transient p: p will be false eventually. $\Box\Diamond\neg p$.

- Ensures: More useful in practice. Compositional.
  Defined using transient.

  p en q:
  once p holds, it continues to hold until q holds; and q holds eventually.

- Leads-to: Typical property in a specification. Non-compositional.
  Defined using ensures.

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

Progress Properties

- Transient: Fundamental property. Compositional.

  transient p: p will be false eventually. $\Box\Diamond\neg p$.

- Ensures: More useful in practice. Compositional.
  Defined using transient.

  p en q:
  once p holds, it continues to hold until q holds; and q holds eventually.

- Leads-to: Typical property in a specification. Non-compositional.
  Defined using ensures.

  $p \mapsto q$: once p holds, q 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* :

$$\frac{\begin{array}{c} pre \wedge p \Rightarrow b \\ \{pre \wedge p\}\ \alpha\ \{\neg p\} \end{array}}{\{\}\ f\ \{\ \text{transient}\ \ p \mid \ \}}$$

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.

- $f \,[]\, g$: 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.

- $f \; [] \; g$: p transient in f or g.

- Inheritance: If p transient in ALL components of f, p transient in f.

Ensures: p en q

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

- $p \wedge \neg q$ co $p \vee q$
- transient $p \wedge \neg q$

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 $ctr \neq old_j$.

  $ctr = m \wedge nb = N$ en $nb < N \vee ctr > m$ in f (E)

Proof strategy

$ctr = m \wedge nb = N$ **en** $nb < N \vee ctr > m$ in f (E)

- To prove (E) in $[]_j f_j$: 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 β_j and γ_j, i.e.,
  Effective executions of β_j and γ_j establish the postcondition of (E) given its pre-condition.

Proof strategy

$ctr = m \wedge nb = N$ **en** $nb < N \vee ctr > m$ in f (E)

- To prove (E) in $[]_j\, f_j$: 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 β_j and γ_j, i.e.,
  Effective executions of β_j and γ_j establish the postcondition of (E) given its pre-condition.

Proof strategy

$ctr = m \wedge nb = N$ **en** $nb < N \vee ctr > m$ in f (E)

- To prove (E) in $[]_j\, f_j$: 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 β_j and γ_j, i.e.,
  Effective executions of β_j and γ_j establish the postcondition of (E) given its pre-condition.

Proof strategy

$ctr = m \wedge nb = N$ **en** $nb < N \vee ctr > m$ in f (E)

- To prove (E) in $[]_j\, f_j$: 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 β_j and γ_j, i.e.,
  Effective executions of β_j and γ_j establish the postcondition of (E) given its pre-condition.

Proof strategy

$$ctr = m \wedge nb = N \ \ \mathbf{en} \ \ nb < N \ \vee \ ctr > m \text{ in } f \qquad \text{(E)}$$

- To prove (E) in $[]_j\, f_j$: 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 β_j and γ_j, i.e.,
  Effective executions of β_j and γ_j establish the postcondition of (E) given its pre-condition.

Proof Obligations

Relevant Annotation of f_j:

$$\begin{array}{ll} \textbf{if} & [\ \beta_j :: \ \{new_j = old_j + 1\}\ ctr = old_j \ \rightarrow \ ctr := new_j \ \ \{true\} \\ & \mid \gamma_j :: \ \{new_j = old_j + 1\}\ ctr \neq old_j \ \rightarrow \ old_j := ctr \ \ \{true\}] \end{array}$$
$$\{true\}$$

Proof Obligations:

$$\begin{array}{l} \beta_j :: \ \{ctr = m \wedge nb = N \wedge new_j = old_j + 1 \wedge ctr = old_j\} \\ \qquad ctr := new_j \\ \qquad \{nb < N \vee ctr > m\} \end{array}$$

$$\begin{array}{l} \gamma_j :: \ \{ctr = m \wedge nb = N \wedge new_j = old_j + 1 \wedge ctr \neq old_j\} \\ \qquad old_j := ctr \\ \qquad \{nb < N \vee ctr > m\} \end{array}$$

Leads-to

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

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

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

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

$$\frac{(\forall p : p \in S : p \mapsto q)}{(\vee p : p \in S : p) \mapsto q}$$

Derived Rules: What makes Proofs Practical. For co

$$\mathit{false}\ \ \mathbf{co}\ \ q$$

$$p\ \ \mathbf{co}\ \ \mathit{true}$$

$$\frac{p\ \mathbf{co}\ q\ ,\ p'\ \mathbf{co}\ q'}{p \wedge p'\ \mathbf{co}\ q \wedge q'} \qquad \text{(CONJUNCTION)}$$

$$\frac{p\ \mathbf{co}\ q\ ,\ p'\ \mathbf{co}\ q'}{p \vee p'\ \mathbf{co}\ q \vee q'} \qquad \text{(DISJUNCTION)}$$

$$\frac{p\ \mathbf{co}\ q}{p \wedge p'\ \mathbf{co}\ q} \qquad \text{(LHS STRENGTHENING)}$$

$$\frac{p\ \mathbf{co}\ q}{p\ \mathbf{co}\ q \vee q'} \qquad \text{(RHS WEAKENING)}$$

Lightweight Derived Rules for $\mapsto$

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

2. (lhs strengthening, rhs weakening)
$$\frac{p \mapsto q}{\begin{array}{c} p' \wedge p \mapsto q \\ p \mapsto q \vee q' \end{array}}$$

3. (cancellation) $$\frac{p \mapsto q \vee r \, , \; r \mapsto s}{p \mapsto q \vee s}$$

Heavyweight Derived Rules for $\mapsto$

1. (PSP) $$\frac{p \mapsto q \qquad \text{stable } p'}{p \wedge p' \mapsto q \wedge p'}$$

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

$$\frac{(\forall m :: p \wedge M = m \mapsto (p \wedge M \prec m) \vee q)}{p \mapsto q}$$

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

$$\frac{(\forall i :: \; p_i \mapsto q_i \vee b \quad q_i \text{ co } q_i \vee b)}{(\forall i :: p_i) \mapsto (\forall i :: q_i) \vee b}$$

Heavyweight Derived Rules for $\mapsto$

1. (PSP) $$\frac{p \mapsto q \quad \text{stable } p'}{p \wedge p' \mapsto q \wedge p'}$$

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

$$\frac{(\forall m :: p \wedge M = m \mapsto (p \wedge M \prec m) \vee q)}{p \mapsto q}$$

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

$$\frac{(\forall i :: \quad p_i \mapsto q_i \vee b \quad q_i \text{ co } q_i \vee b \quad)}{(\forall i :: p_i) \mapsto (\forall i :: q_i) \vee b}$$

Heavyweight Derived Rules for $\mapsto$

1. (PSP) $$\frac{\begin{array}{c} p \mapsto q \\ \text{stable } p' \end{array}}{p \wedge p' \mapsto q \wedge p'}$$

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

$$\frac{(\forall m :: \; p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M \prec m) \;\vee\; q)}{p \;\mapsto\; q}$$

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

$$\frac{\begin{array}{l} (\forall i :: \\ \quad p_i \mapsto q_i \vee b \\ \quad q_i \text{ co } q_i \vee b \\ ) \end{array}}{(\forall i :: \; p_i) \;\mapsto\; (\forall i :: \; q_i) \;\vee\; b}$$

Heavyweight Derived Rules for $\mapsto$

1. (PSP)
$$\frac{\begin{array}{c} p \mapsto q \\ \text{stable } p' \end{array}}{p \wedge p' \mapsto q \wedge p'}$$

2. (induction) M : Program States $\to W$. $(W, \prec)$ well-founded.
$$\frac{(\forall m :: p \wedge M = m \mapsto (p \wedge M \prec m) \vee q)}{p \mapsto q}$$

3. (completion) p_i and q_i are predicates; i index over a finite set.
$$\frac{\begin{array}{l} (\forall i :: \\ \quad p_i \mapsto q_i \vee b \\ \quad q_i \text{ co } q_i \vee b \\ ) \end{array}}{(\forall i :: p_i) \mapsto (\forall i :: q_i) \vee b}$$

Distributed Counter

- Prove in f: ctr increases unboundedly:

  $true \;\mapsto\; ctr > C$, for any integer C

- Proved in f: $ctr = m \wedge nb = N$ en $nb < N \vee ctr > m$

- Use definition of $\mapsto$ and its derived rules for the proof.

Distributed Counter, Contd.

$ctr = m \wedge nb = N$ en $nb < N \vee ctr > m$

proven

$ctr = m \wedge nb = N \mapsto nb < N \vee ctr > m$

basis rule of *leads-to*

$ctr = m \wedge nb = N \mapsto ctr = m \wedge nb < N \vee ctr > m$

PSP with $ctr = m$ co $ctr = m \vee ctr = m + 1$

Distributed Counter, Contd.

$ctr = m \wedge nb = N$ en $nb < N \vee ctr > m$

proven

$ctr = m \wedge nb = N \mapsto nb < N \vee ctr > m$

basis rule of *leads-to*

$ctr = m \wedge nb = N \mapsto ctr = m \wedge nb < N \vee ctr > m$

PSP with $ctr = m$ co $ctr = m \vee ctr = m + 1$

Distributed Counter, Contd.

$ctr = m \wedge nb = N$ en $nb < N \vee ctr > m$

proven

$ctr = m \wedge nb = N \mapsto nb < N \vee ctr > m$

basis rule of *leads-to*

$ctr = m \wedge nb = N \mapsto ctr = m \wedge nb < N \vee ctr > m$

PSP with $ctr = m$ co $ctr = m \vee ctr = m + 1$

Apply Induction Rule

$$ctr = m \wedge nb = N \;\mapsto\; ctr = m \wedge nb < N \;\vee\; ctr > m$$

Induction rule:

$$\frac{(\forall m :: \; p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M \prec m) \;\vee\; q)}{p \;\mapsto\; q}$$

Use nb for M and $<$ for $\prec$ to conclude:

$$ctr = m \;\mapsto\; ctr > m$$

Distributed Counter, Contd.

$ctr = m \wedge nb = N$ **en** $nb < N \vee ctr > m$

proven

$ctr = m \wedge nb = N \mapsto nb < N \vee ctr > m$

basis rule of *leads-to*

$ctr = m \wedge nb = N$ **en** $ctr = m \wedge nb < N \vee ctr > m$

PSP with $ctr = m$ co $ctr = m \vee ctr = m + 1$

$ctr = m \mapsto ctr > m$

Induction rule; well-founded order $<$ over natural numbers

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

Induction rule, well-founded order $<$ over natural numbers.

Distributed Counter, Contd.

$ctr = m \wedge nb = N \ \ \text{en} \ \ nb < N \vee ctr > m$

proven

$ctr = m \wedge nb = N \ \mapsto \ nb < N \vee ctr > m$

basis rule of *leads-to*

$ctr = m \wedge nb = N \ \ \text{en} \ \ ctr = m \wedge nb < N \vee ctr > m$

PSP with $ctr = m \ \text{co} \ ctr = m \vee ctr = m + 1$

$ctr = m \ \mapsto \ ctr > m$

Induction rule; well-founded order $<$ over natural numbers

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

Induction rule, well-founded order $<$ over natural numbers.