# Programs, Semantics and Effective Atomicity

Shankar

April 3, 2014

# Outline

# Program structure

- program *name* ( *params* )
  ia { *pred* }
  *main*
  *functions*
  *input functions*
  atomicity assumption {...}
  progress assumption {...}

- Read-only variables
  - mysid: "this" sid
  - mytid: "this" tid
- Parameters read-only

- input *rtype* mysid.*name* ( *params* ) // input function
  ia { *pred* }
  *body*

- *rval* ← *sid.fname* ( *params* ); // call to environment
  ia { *pred* }

# Systems and Threads

- startSystem(*P*(*params*))
  - instantiates program *P*
  - basic system is created with a unique system id (sid)
  - instantiating thread executes main and returns
  - system remains
- Aggregate system *x*: basic system *x* and its descended systems
- Composite system: arbitrary collection of systems
- startThread(*F*(*params*))
  - creates thread executing *local non-input* function *F*
  - returns a unique thread id (abbr tid)
  - thread ends when it reaches end of *F*

# System termination

- Platform eventually terminates a system if
  - a thread in system has executed endSystem()
  - system is continuously in a *endable* state
- System is endable
  - no guest threads in the system
  - no local thread of the system is in another system

  Ensures that a thread is not left in limbo.
- At termination, platform
  - terminates all local threads
  - cleans up system's state

# Outline

# Service program structure



- A service program is essentially a state machine organized into "input" and "output" functions

```
service prog name(params) {
  ic {predicate in params}
  <main>                        // define and initialize variables
  <input functions>
  <output functions>
  <atomicity and progress assumptions>
}
```

- Does not create any other system
  - so only one basic system, even for a distributed service
- Creates threads only to execute output functions (if any)
- Maximal atomicity: every atomic step does input or output

# Input function



- Consists of
  - input part: executed atomically when function is called
  - output part: executed atomically when function returns
- Input part consists of
  - input condition: predicate in vars and params, no side-effect
  - body: non-blocking deterministic update to main's vars

  Body is executed if input condition holds, o/w fault
- Output part consists of
  - output condition and body, as in input part

  Body is executed only if output condition holds, o/w block
- Note: input function never calls the environment

# Input function: general case

```
input retType sid.fname(param)
  ic {predicate}
  body
  output(extParam, internalParam)
    oc {pred}
    body
    return rval;
```

input part

output part

- output(.): introduces additional parameters for output part
  - *extParam*: return value; allows external nondeterminism
  - *internalParam*: allows internal nondeterminism
  - parameters can have any value allowed by oc's *pred*
  - parameters not updated in output body

# Output function



- Output function: "reverse" of an input function
  - output part followed by input part
- Output part: output condition and body
  - body ends in call to environment, say *sid.fn*(*param*)
  - atomically create thread and execute body (including call) only if output condition holds, o/w block
- Input part: input condition and body
  - body starts with the call's return value (if any)
  - upon return, atomically execute body and terminate thread if input condition holds, o/w fault
- Never called by environment.
- Program has no other call to *sid.fn*(.)
  - so all its *sid.fn*(.) calls are caputed by the output condition

# Output function: general case



- output *fname*( *extParam*, *intParam*) {
  oc {*oc predicate*}
  *output body*
  rval ← *sid*.*fn*(*args*);
  ic {*ic predicate*}
  *input body*
  }

| | |
|---|---|
| output part, ends at *sid.fn(.)* | |
| input part, begins at *rval* | |

- *extParam*: *sid* and *args* of the call
- *intParam*: internal parameters, allows internal nondeterminism

# Atomicity and progress assumptions



- Atomicity assumption
  - main, input parts, output parts

- Progress assumption
  - predicate with terms replaced by leads-to assertions, e.g.
    - $P \Rightarrow Q \quad \longrightarrow \quad (A \ \mathit{leads\text{-}to} \ B) \Rightarrow (C \ \mathit{leads\text{-}to} \ D)$
    - `forsome(j:` $P$`)` $\longrightarrow$ `forsome(j: (`$A \ \mathit{leads\text{-}to} \ B$`) )`
  - "thread-location" expressions are restricted to
    - "`thread t in s.f`" and its negation where `s.f` is an input function or output call of the service
  - locally realizable: w/o requiring inputs from environment

# Fault-Freedom and Usability



- Service program must be fault-free
- Service program with internal parameters must be usable
  - for any input $e$,
    for any finite evolutions $x$ and $y$ st $ioseq(x) = ioseq(y)$,
    $e$ accepted at end of $x$ iff $e$ accepted at end of $y$
- Otherwise the service program is useless as a standard

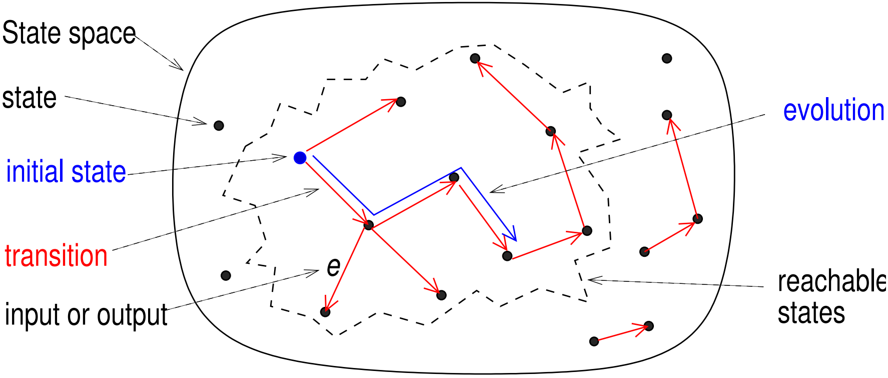
# Outline

# State transition model



- To reason about a program, need a mathematical model of its evolutions
- We use a state transition model
  - state: value assignment of vars, params, thread locations
  - transition: state change due to execution of an atomic step
  - evolution: sequence of transitions starting from initial state
- Provide state transition model for a composite system $M$
  - $M$ can be the aggregate system of a program

# State transition model




- First transition creates the system
  - initial state: system not yet created
  - next state: system exists

# States of $M$



- State of a basic system
  - value assignment of vars, params, thread locations
- State of composite system $M$ with multiple basic systems
  - collection of states of the basic systems in $M$
  - for a state $s$ and a component system $P$
    - $s.P$: $P$'s component of $s$

# Transitions of $M$



- Transition: $\langle s, t \rangle$ or $\langle s, e, t \rangle$ // atomic step execution
  - $s$: start state; fault-free
  - $e$: input or output, if present
  - $t$: end state; fault-free or `fault`
  - atomicity can be effective or platform-provided
- Types of transitions
  - basic internal: no io; internal to a basic system
  - input: input $e$ from environment
  - output: output $e$ to environment
  - composite internal: io $e$ between two basic systems of $M$
- For non-faulty transitions
  - basic internal, input, output: affect only one basic system of $M$
  - composite internal: affects two basic systems of $M$

# Evolutions

- Evolution of $M$: path in the state transition model
  - starts from initial state
  - has at least one transition // creates first basic system of $M$
  - finite (can end in `fault`), or infinite (no `fault`)
- Complete evolution: one that satisfies progress assumption of $M$
- Allowed evolution: one where every input is allowed
  - i.e., every input satisfies its input assumption
- Set of allowed evolutions determine $M$'s correctness properties
- $M$ is fault-free iff every allowed evolution is fault-free
  - an allowed evolution can be faulty

# Outline

# Predicates



- Predicates: express properties of system states
  - fault state does not satisfy any predicate
- Predicate: boolean-valued construct in
  - boolean-valued terms: usually involving system quantities
  - propositional operators: not, and, or, $\Rightarrow$, $\Leftrightarrow$, *OR*
  - quantifiers: `forall`, `forsome`, `forone`
- Bound variable: variable defined in the scope of a quantifier
- Free variable: variable that is not bound
- If free variable x of predicate P is not a system quantity, then P holds at a state iff `forall`$(x : P)$ holds at the state

# Assertions



- Assertions: express properties of system evolutions
  - faulty evolution does not satisfy any assertion
- Two kinds of properties: safety and progress
- Safety: nothing "bad" happens
  - if a finite sequence $x$ does not satisfy it, no extension of $x$ will satisfy it
- Progress: something "good" eventually happens
  - if a finite sequence $x$ does not satisfy it, there is an extension of $x$ that will satisfy it

# Safety assertions

- Invant assertion: $\mathit{Inv}\ P$ // $P$ predicate
  - holds for evolution $x$ if every non-initial state of $x$ satisfies $P$

- Unless assertion: $P\ \mathit{unless}\ Q$ // $P$, $Q$ predicates
  - holds for evolution $x$ if
    for every non-initial state $s$ of $x$ satisfying $P$ and not $Q$,
    $s$ is the last state of $x$ or the next state satisfies $P$ or $Q$

- Safety assertion: predicate [terms $\rightarrow$ invariant/unless assertions]
  - e.g., `forall(int n: (`$\mathit{Inv}$ `P)` $\Rightarrow$ `(Q` $\mathit{unless}$ `R))`
  - holds for evolution $x$ if predicate holds after evaluating its component assertions on $x$

- Safety assertion holds for a system if it holds for all its allowed evolutions

# Progress assertions – 1

- Weak fairness for thread $t$
  - holds for evolution $x$ if
    - $x$ is finite and $t$ is blocked in last state of $x$, or
    - $x$ is infinite and $t$ executes infinitely often or is blocked infinitely often
- Strong fairness for thread $t$
  - holds for evolution $x$ if
    - $x$ is finite and $t$ is blocked in last state of $x$, or
    - $x$ is infinite and $t$ executes infinitely often if it is unblocked infinitely often
- Weak (strong) fairness for statement $S$
  - weak (strong) fairness for every thread on $S$

# Progress assertions – 2

- Leads-to assertion: $P$ *leads-to* $Q$ // $P$, $Q$ predicates
  - holds for evolution $x$ if
    for every non-initial state $s$ of $x$ satisfying $P$ and not $Q$,
    some later state satisfies $P$ or $Q$

- Progress assertion: pred [terms $\rightarrow$ fairness/leads-to assertions]
  - e.g., `forall(int n: (P` *leads-to* `Q)` $\Rightarrow$ `(R` *leads-to* `S))`
  - holds for evolution $x$ if predicate holds after evaluating its component assertions on $x$

- Progress assertion holds for a system if the system is fault-free and every complete allowed evolution satsfies assertion

# Outline

# Image definitions

- Let $C$ be a composite system.
- Let $P$ be the composite system of a subset of basic systems of $C$.
- For every state, transition, or evolution $x$ of $C$
  - $x$ has an image on $P$, denoted $x.P$
- For state $s$ of $C$: $s.P \; = \;$ part of $s$ concerning $P$
- For transition $t = \langle u, e, v \rangle$ of $C$:

$$t.P \; = \; \begin{cases} \langle u.P,\ e,\ v.P \rangle & \text{if } t \text{ involves } P \\ \langle\ \rangle & \text{o/w} \end{cases}$$

- For evolution $x$ of $C$:

$x.P \; = \; x$ with every transition $t$ replaced by $t.P$

# Splitting

- Let $C$ be a composite system.
- Let $P$ be the composite system of a subset of $C$
- Let $x$ be a fault-free evolution of $C$ st $x.P$ is not null

- Theorem
  - $x.p$ is a fault-free evolution of $P$
  - for any assertion $\beta$ not involving $C - P$:
    $x$ satisfies $\beta$ iff $x.P$ satisfies $\beta$
  - if $x$ is a complete evolution of $C$:
    $x.P$ is a complete evolution of $P$

- Proof
  - easy; by induction on # transitions in $x$; see text

# Stitching Theorem



- Let $P_1, \cdots, P_N$ be disjoint composite systems
- Let $C$ be union of $P_1, \cdots, P_N$
- Let $x_1, \cdots, x_N$ be fault-free evolutions of $P_1, \cdots, P_N$
- Definition: $x_1, \cdots, x_N$ are signature-compatible if
  - there is a merge $y$ of $io(x_1), \cdots, io(x_N)$ such that $y_K$ is output $e$ of $P_i$ to $P_j \;\Rightarrow\; y_{K+1}$ is input $e$ of $io(x_j)$
- Theorem
  - there is a fault-free evolution $z$ of $C$ st $z.P_i = x_i$ for all $i$ iff $x_1, \cdots, x_N$ are signature-compatible
  - for any assertion $\beta$ not involving $C - P_i$:
    $z$ satisfies $\beta$ iff $x_i$ satisfies $\beta$
  - $z$ is a complete evolution of $C$ iff
    $x_i$ is complete evolution of $P_i$ for all $i$

# Outline

# Auxiliary variables



- Record information about a program's behavior without influencing its evolutions
- Auxiliary variable condition
  - aux vars do not appear in output conditions
  - aux var value not used in updating a non-aux var
  - any statement involving aux vars is fault-free
    - treat as atomic with an adjacent "non-aux" statement
- Theorem: Let $Q$ be program $P$ extended with auxiliary vars
  - for any $Q$-evolution $x$ (faulty or not): $x.P$ is a $P$-evolution
  - for any $P$-evolution $y$: there is a $Q$-evolution $x$ st $x.P = y$
  - for any assertion $\beta$ of $Q$: $P$ satisfies $\beta$ iff $Q$ satisfies $\beta$

# Outline

# Effective atomicity definition



- Let $S$ be a code chunk in a program $X$
- $S$-run: an execution of $S$ by a thread in an evolution
  - sequence of transitions $\langle t_1, t_2, \cdots t_n \rangle$
  - may not be contiguous: eg, $t_i$ end state $\neq t_{i+1}$ start state
  - may be whole or partial
- $S$-run is atomic if contiguous and whole
- $S$ is effectively atomic if:
  for every evolution $w$, there is an evolution $w'$ st
  - every $S$-run in $w'$ is atomic
  - $ioseq(w)$ equals $ioseq(w')$
  - if $w$ is complete then $w'$ is complete

# Effective atomicity theorem



- Theorem:
  - let $S$ be effectively atomic in system $X$
  - let $X'$ be $X$ with $S$ (platform-provided) atomic
  - let $\beta$ be a correctness property concerning only $\mathit{ioseqs}(X)$
  - then $X$ satisfies $\beta$ iff $X'$ satisfies $\beta$

# Effective atomicity for arbitrary properties



- Let $\mathcal{Z}(.)$ be a function on evolutions // e.g., *ioseq*(.)
- $S$ in program $X$ is effectively atomic wrt $\mathcal{Z}$ if: for every evolution $w$, there is an evolution $w'$ st
  - every $S$-run in $w'$ is atomic
  - $\mathcal{Z}(w)$ equals $\mathcal{Z}(w')$
  - if $w$ is complete then $w'$ is complete
- Let $\beta$ be a correctness property concerning only $\mathcal{Z}$
  - i.e., $\beta(w) = \beta(w')$ if $\mathcal{Z}(w) = \mathcal{Z}(w')$
- Theorem:
  - let $S$ and $\beta$ be as above
  - let $X'$ be $X$ with $S$ (platform-provided) atomic
  - then $X$ satisfies $\beta$ iff $X'$ satisfies $\beta$

# Outline

# Commuting transitions

- Commutativity is an incremental technique for $w \;\rightarrow\; w'$
- Let $\mathcal{Z}(.)$ be a function on sequences of transitions
- A sequence of transitions $x$ is massageable wrt $\mathcal{Z}$ if modifying only the states in $x$ yields an evolution $x'$ s.t.$\mathcal{Z}(x) = \mathcal{Z}(x')$
- A contiguous transition pair (ctp) is a pair of transitions $\langle t_1, t_2 \rangle$ s.t. $t_1$'s end state equals $t_2$'s start state
- Ctp $\langle t_1, t_2 \rangle$ commutes wrt $\mathcal{Z}$ if in every evolution $w$ with the cpt, replacing it by $\langle t_2, t_1 \rangle$ yields a sequence that is massageable wrt $\mathcal{Z}$

Typically if $t_1 = \langle a, F, b \rangle$ and $t_2 = \langle b, G, c \rangle$, only $b$ changes

- if $\quad w = [\cdots, \langle a, F, b \rangle, \langle b, G, c \rangle, \cdots]$
  then $\quad w' = [\cdots, \langle a, G, d \rangle, \langle d, F, c \rangle, \cdots]$

## Commuting atomic statements



- Let $F$ and $G$ be atomic statements in program $X$
- $\langle F, G \rangle$ commutes wrt $\mathcal{Z}$ if every ctp $\langle t_1, t_2 \rangle$ s.t.
  - $t_1$ is an $F$-transition
  - $t_2$ is a $G$-transition by another thread

  commutes wrt $\mathcal{Z}$

- Let $S$ be a code chunk in program $X$.
- For every atomic $F$ in $S$ and atomic $G$ in $X$
  - if $\langle F, G \rangle$ commutes wrt $\mathcal{Z}$ then every $S$-run can be coalesced
  - if $\langle F, G \rangle$ commutes wrt $\mathcal{Z}$ then every $S$-run can be coalesced

- What remains is to handle partial $S$-runs

# Handling partial runs



- Let $S$ be a code chunk in program $X$.
- An atomic $F$ in $S$ is tail-droppable wrt $\mathcal{Z}$ if for every evolution $w$ with a partial $S$-run ending at $F$, deleting the $F$-transition yields a sequence massageable wrt $\mathcal{Z}$
- An atomic $F$ in $S$ is tail-appendable wrt $\mathcal{Z}$ if for every evolution $w$ with a $S$-run ending just before $F$, inserting the $F$-transition at the end of the $S$-run yields a sequence massageable wrt $\mathcal{Z}$

# Effective atomicity theorem



- Let the following hold
  - $S$ is a code chunk in program $X$
  - $K$ is an atomic statement in $S$
  - for every atomic $F$ in $S$ before $K$ and every $G$ in $X$
    - $\langle F, G \rangle$ commutes wrt $\mathcal{Z}$
    - $F$ is tail-droppable wrt $\mathcal{Z}$
  - for every atomic $F$ in $S$ after $K$ and every $G$ in $X$
    - $\langle G, F \rangle$ commutes wrt $\mathcal{Z}$
    - $F$ is tail-appendable wrt $\mathcal{Z}$
- Then
  - $S$ is effectively atomic
  - $K$ is said to be the anchor of $S$

## Some simple sufficient conditions



- Below, $S$, $R$, $J$, $K$ are code chunks in program $X$
- $S$ and $R$ **interfere** if they conflict (over a variable) or both do io
- Theorem
  - let $S$ be blockable only at the start
  - let $S$ not interfere with any simultaneously executable $J$
  - then $S$ is effectively atomic.
- Theorem
  - let $S$ be blockable only at the start
  - let $K$ be atomic in $S$
  - let $S$ w/o $K$ not interfere with any simultaneously executable $J$
  - then $S$ is effectively atomic and $K$ is its anchor

# Outline

# Proof rules overview



- Assume a given program throughout this section
- Proof rule: template of requirements and concluding assertion
  - conclusion holds if requirements hold
  - requirements involve program/predicates/assertions
  - predicates/assertions need to be invented
  - requirements mechanically checkable
- Hoare-triples: properties of code in isolation
- Proof rules for safety assertions: $\mathit{Inv}\ P$; $P\ \mathit{unless}\ Q$
- Proof rules for progress assertions: $P\ \mathit{leads\text{-}to}\ Q$

- Hoare-triple $\{P\}\,S\,\{Q\}$ // code chunk $S$, predicates $P$, $Q$
  - $P$: precondition; $Q$: postcondition
- For $S$ nonblocking and non-input:
  - $\{P\}\,S\,\{Q\}$ means executing $S$ in isolation starting from *any* state satisfying $P$ always terminates with $Q$ holding
- For $S$ with blocking/input condition $B$ and action $C$:
  - $\{P\}\,S\,\{Q\}$ means $\{P \text{ and } B\}\,C\,\{Q\}$
- Terminology

  $\{P\}\,S\,\{Q\}$ aka "$S$ unconditionally establishes $Q$ from $P$"

  $\{true\}\,S\,\{Q\}$ aka "$S$ unconditionally establishes $Q$"

  $\{Q\}\,S\,\{Q\}$ aka "$S$ unconditionally preserves $Q$"

# Hoare-triple examples

- {true} if $x \neq y$ then $x \leftarrow y+1$ {$(x = y+1)$ or $(x = y)$} (valid)
- {$x = n$} for (i in 0..10) $x \leftarrow x+i$ {$x = n+55$} (valid)
- {$x = 3$} $x \leftarrow y+1$ {$x = 4$} (invalid; eg, if y is 1 at start)
- {$(x = 1)$ and $(y = 1)$} while $(x > 0)$ $x \leftarrow 2*x$ {$y = 1$}
  (invalid; does not terminate)
- {true} await $(x \neq y)$ $x \leftarrow y+1$ {$x=y+1$} (valid)
- {true} oc{$x \geq 1$} $y \leftarrow 1/(2-x)$ {$y=1/(2-x)$}
  (invalid; may divide by zero)

- Proof rules for Hoare-triples: see text

# Safety: invariance proof rule

- Invariance induction rule

  $\mathit{Inv}\ P$ holds if the following hold:

  1. for initial atomic step $f$: {true} $f$ $\{P\}$
  2. for every non-initial atomic step $e$: $\{P\}$ $e$ $\{P\}$

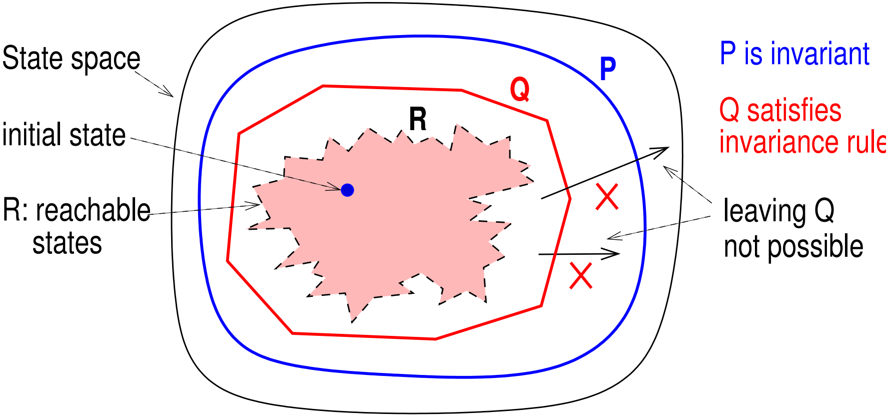
  Above aka "$P$ satisfies invariance rule"

- To exploit a previously-established $\mathit{Inv}\ R$

  1. {true}$f\{P\}$ $\longrightarrow$ {true} $f$ $\{R \Rightarrow P\}$
  2. $\{P\}e\{P\}$ $\longrightarrow$ $\{P$ and $R\}$ $e$ $\{R \Rightarrow P\}$

  Above aka "$P$ satisfies invariance rule assuming $\mathit{Inv}\ R$"

# Reachable vs invariant vs inv rule



# Safety: unless proof rule



- Unless rule $P$ *unless* $Q$ holds if
  - for every non-initial atomic step *e*:

    $\{P \text{ and not } Q\}\ e\ \{P \text{ or } Q\}$

  Above aka "$P$ satisfies unless rule"

- To exploit a previously-established *Inv* $R$
  - *pre* $\longrightarrow$ *pre* and $R$
  - *post* $\longrightarrow$ $R \Rightarrow$ *post*

  Above aka "$P$ satisfies unless rule assuming *Inv* $R$"

# Safety: closure proof rules

- Closure rules: requirements do not involve program
- $\mathit{Inv}\ P$ holds if $P$ holds
- $\mathit{Inv}\ P$ holds if $\mathit{Inv}\ Q$ and $\mathit{Inv}\ Q \Rightarrow P$ hold
- $P\ \mathit{unless}\ Q$ holds if $\mathit{Inv}\ P \Rightarrow Q$ holds
- $P\ \mathit{unless}\ Q$ holds if following hold:
  - $U\ \mathit{unless}\ V$
  - $\mathit{Inv}\ P \Rightarrow U$
  - $\mathit{Inv}\ V \Rightarrow Q$

We say "assertion holds via closure of <assertions>"

# Progress: leads-to weak-fair rule

- $e$.enabled, for atomic step $e$
  - (thread at $e$) — if $e$ is non-blocking
  - (thread at $e$) and $B$ — if $e$ has blocking condition $B$

- Leads-to weak-fair rule

  $P$ *leads-to* $Q$ holds if following hold:
  - $e$ is a weak-fair atomic step
  - ($P$ and not $Q$) $\Rightarrow$ $e$.enabled
  - $\{P$ and not $Q\}\, e\, \{Q\}$
  - for every non-initial atomic step $f$:

    $\{P$ and not $Q\}\, f\, \{P$ or $Q\}$

  Above aka "$P$ leads-to $Q$ via wfair $e$

# Progress: leads-to strong-fair rule

- Leads-to strong-fair rule
  $P$ *leads-to* $Q$ holds if following hold:
  - $e$ is a strong-fair atomic step
  - ($P$ and not $Q$ and not $e$.enabled) *leads-to*
    ($Q$ or $e$.enabled)
  - $\{P$ and not $Q\}\ e\ \{Q\}$
  - for every non-initial atomic step $f$:
    $\{P$ and not $Q\}\ f\ \{P$ or $Q\}$

  Above aka "$P$ leads-to $Q$ via sfair $e$

# Progress: closure proof rules



- $P$ *leads-to* ($Q_1$ or $Q_2$) holds if following hold:
  - $P$ *leads-to* ($P_1$ or $Q_2$)
  - $P_1$ *leads-to* $Q_1$
- $P$ *leads-to* $Q$ holds if following hold for some $R$:
  - *Inv* $R$
  - ($P$ and $R$) *leads-to* ($R \Rightarrow Q$)
- ($P_1$ and $P_2$) *leads-to* $Q_2$ holds if following hold for some $Q_1$:
  - $P_1$ *leads-to* $Q_1$
  - $P_2$ *unless* $Q_2$
  - *Inv* ($Q_1 \Rightarrow$ not $P2$)
- $P$ *leads-to* $Q$ holds if following hold for some $R$, $S$:
  - $P$ *unless* $Q$
  - *Inv* ($P \Rightarrow R$)
  - $R$ *leads-to* $S$
  - *Inv* ($S \Rightarrow$ not $R$)

# Progress: well-founded closure rule

- $P$ *leads-to* $Q$ holds if the following hold:
  - $F$ is a function on a lower-bounded partial order $(Z, \prec)$
  - $P$ *leads-to* (Q or forsome($x$ in $Z$ : $F(x)$))
  - forall($x$ in $Z$:
    $F(x)$ *leads-to* (Q or forsome($w$ in $Z$, $w \prec x$ : $F(w)$))

We say "assertion holds via closure of <assertions>"