

Hoare Logic (I): Axiomatic Semantics and Program Correctness

(Based on [Apt and Olderog 1991; Gries 1981;
Hoare 1969; Kleymann 1999; Sethi 199])

Yih-Kuen Tsay

Dept. of Information Management
National Taiwan University



Prelude: the Coffee Can Problem

- 🌍 **Initially:** a coffee can contains some black beans and some white beans.
- 🌍 **Action:** the following steps are repeated as many times as possible.
 1. Pick any two beans from the can.
 2. If they have the same color, put another black bean in. (Assume there is a sufficient supply of additional black beans.)
 3. Otherwise, put the white bean back in and throw the black one away.
- 🌍 **Finally:** only one bean remains in the can.
- 🌍 **Question:** what can be said about the color of the last remaining bean?



The Coffee Can Problem as a Program

$B, W := m, n; \ // \ m > 0 \wedge n > 0$

do $B \geq 0 \wedge W \geq 2 \rightarrow B, W := B + 1, W - 2 \ // \ \text{both white}$

$\parallel \ B \geq 2 \wedge W \geq 0 \rightarrow B, W := B - 1, W \ // \ \text{both black}$

$\parallel \ B \geq 1 \wedge W \geq 1 \rightarrow B, W := B - 1, W \ // \ \text{different colors}$

od

- 🌐 What are the values of B and W , when the program terminates?
- 🌐 Will the program actually terminate?

Invariants and Rank Functions

- 🌐 An *invariant* captures something that is never changed by the program.
- 🌐 A *rank function* (or variant function) measures the progress made by the program.
- 🌐 For the Coffee Can problem,
 - ☀️ (Loop) Invariant: the parity of the number of white beans never changes, i.e., $W \equiv n \pmod{2}$. (in addition, $B + W \geq 1$)
 - ☀️ Rank Function: the total number of beans, i.e., $B + W$.
 - ☀️ The do loop decrements the rank function by one in each iteration and eventually terminates when $B + W = 1$ (i.e., $B = 0 \wedge W = 1$ or $B = 1 \wedge W = 0$).
 - ☀️ So, what is the color of the remaining bean?

An Axiomatic View of Programs

- 🌐 The properties of a program can, in principle, be found out from its text by means of purely *deductive* reasoning.
- 🌐 The deductive reasoning involves the application of valid *inference rules* to a set of valid *axioms*.
- 🌐 The choice of axioms will depend on the choice of programming languages.
- 🌐 We shall introduce such an axiomatic approach, called the *Hoare logic*, to program correctness.



Assertions

- 🌐 When executed, a program will evolve through different *states*, which are essentially a mapping of the program variables to values in their respective domains.
- 🌐 An *assertion* is a precise statement about the state of a program.
- 🌐 Most interesting assertions can be expressed in a *first-order* language.



Pre and Post-conditions

- 🌐 The behavior of a “structured” (single-entry/single-exit) program statement can be characterized by **attaching assertions at the entry and the exit of the statement.**
- 🌐 For a statement S , this is conveniently expressed as a so-called *Hoare triple*, denoted $\{P\} S \{Q\}$, where
 - ☀️ P is called the *pre-condition* and
 - ☀️ Q is called the *post-condition* of S .



Interpretations of a Hoare Triple

- 🌐 A Hoare triple $\{P\} S \{Q\}$ may be interpreted in two different ways:
 - ☀️ **Partial Correctness:** if the execution of S starts in a state satisfying P and terminates, then it results in a state satisfying Q .
 - ☀️ **Total Correctness:** if the execution of S starts in a state satisfying P , then it will terminate and result in a state satisfying Q .

Note: sometimes we write $\langle P \rangle S \langle Q \rangle$ when total correctness is intended.



Pre and Post-Conditions for Specification

- 🌐 Find an integer approximate to the square root of another integer n :

$$\{0 \leq n\} ? \{d^2 \leq n < (d + 1)^2\}$$

or slightly better

$$\{0 \leq n\} d := ? \{d^2 \leq n < (d + 1)^2\}$$

- 🌐 Find the index of value x in an array b :

- ☀️ $\{x \in b[0..n - 1]\} ? \{0 \leq i < n \wedge x = b[i]\}$

- ☀️ $\{0 \leq n\} ? \{(0 \leq i < n \wedge x = b[i]) \vee (i = n \wedge x \notin b[0..n - 1])\}$

Note: there are other ways to stipulate which variables are to be changed and which are not.

A Little Bit of History

The following seminal paper started it all:

C.A.R. Hoare. An axiomatic basis for computer programs. *CACM*, 12(8):576-580, 1969.


- 🌐 Original notation: $P \{S\} Q$ (vs. $\{P\} S \{Q\}$)
- 🌐 Interpretation: partial correctness
- 🌐 Provided axioms and proof rules


Note: R.W. Floyd did something similar for flowcharts earlier in 1967, which was also a precursor of “proof outline” (a program fully annotated with assertions).

The Assignment Statement


Syntax:

$$x := E$$

 Meaning: execution of the assignment $x := E$ (read as “ x becomes E ”) evaluates E and stores the result in variable x .

 We will assume that expression E in $x := E$ has *no side-effect* (i.e., does not change the value of any variable).

 Which of the following two Hoare triples is correct about the assignment $x := E$?


 $\{P\} x := E \{P[E/x]\}$


 $\{Q[E/x]\} x := E \{Q\}$


Note: E is essentially a first-order term.




Some Hoare Triples for Assignments

 $\{x > 0\} x := x - 1 \{x \geq 0\}$

 $\{x - 1 \geq 0\} x := x - 1 \{x \geq 0\}$

 $\{x + 1 > 5\} x := x + 1 \{x > 5\}$

 $\{5 \neq 5\} x := 5 \{x \neq 5\}$

Axiom of the Assignment Statement

$$\frac{}{\{Q[E/x]\} x := E \{Q\}} \text{ (Assignment)}$$

Why is this so?


- 🌐 Let s be the state before $x := E$ and s' the state after.
- 🌐 So, $s' = s[x := E]$ assuming E has no side-effect.
- 🌐 $Q[E/x]$ holds in s if and only if Q holds in s' , because
 - ☀ every variable, except x , in $Q[E/x]$ and Q has the same value in s and s' , and
 - ☀ $Q[E/x]$ has every x in Q replaced by E , while Q has every x evaluated to E in s' ($= s[x := E]$).

The Multiple Assignment Statement

Syntax:


$$x_1, x_2, \dots, x_n := E_1, E_2, \dots, E_n$$

where x_i 's are distinct variables.

 Meaning: execution of the multiple assignment evaluates all E_i 's and stores the results in the corresponding variables x_i 's.

Examples:

 $i, j := 0, 0$ (initialize i and j to 0)

 $x, y := y, x$ (swap x and y)

 $g, p := g + 1, p - 1$ (increment g by 1, while decrement p by 1)

 $i, x := i + 1, x + i$ (increment i by 1 and x by i)

Some Hoare Triples for Multi-assignments

Swapping two values

$$\{x < y\} x, y := y, x \{y < x\}$$

Number of games in a tournament

$$\{g + p = n\} g, p := g + 1, p - 1 \{g + p = n\}$$

Taking a sum

$$\{x + i = 1 + 2 + \dots + (i + 1 - 1)\}$$

$$i, x := i + 1, x + i$$

$$\{x = 1 + 2 + \dots + (i - 1)\}$$

Simultaneous Substitution

- 🌐 $P[E/x]$ can be naturally extended to allow E to be a list E_1, E_2, \dots, E_n and x to be x_1, x_2, \dots, x_n , all of which are distinct variables.
- 🌐 $P[E/x]$ is then the result of simultaneously replaying x_1, x_2, \dots, x_n with the corresponding expressions E_1, E_2, \dots, E_n ; enclose E_i 's in parentheses if necessary.
- 🌐 **Examples:**
 - ☀️ $(x < y)[y, x/x, y] = (y < x)$
 - ☀️ $(g + p = n)[g + 1, p - 1/g, p] = ((g + 1) + (p - 1) = n) = (g + p = n)$
 - ☀️ $(x = 1 + 2 + \dots + (i - 1))[i + 1, x + i/i, x]$
 $= ((x + i) = 1 + 2 + \dots + ((i + 1) - 1))$
 $= (x + i = 1 + 2 + \dots + ((i + 1) - 1))$

Axiom of the Multiple Assignment

🌐 Syntax:

$$x_1, x_2, \dots, x_n := E_1, E_2, \dots, E_n$$

where x_i 's are distinct variables.

🌐 Axiom:

$$\frac{\{Q[E_1, \dots, E_n/x_1, \dots, x_n]\} \quad x_1, \dots, x_n := E_1, \dots, E_n \quad \{Q\}}{\text{(Assign.)}}$$



Assignment to an Array Entry

🌐 Syntax:

$$b[i] := E$$

🌐 Notation for an altered array: $(b; i : E)$ denotes the array that is identical to b , except that entry i stores the value of E .

$$(b; i : E)[j] = \begin{cases} E & \text{if } i = j \\ b[j] & \text{if } i \neq j \end{cases}$$

🌐 Axiom:

$$\frac{}{\{Q[(b; i : E)/b]\} b[i] := E \{Q\}} \text{ (Assignment)}$$



Pre and Post-condition of a Loop

- 🌐 A precondition just **before** a loop can capture the conditions for executing the loop.
- 🌐 An assertion just **within** a loop body can capture the conditions for staying in the loop.
- 🌐 A postcondition just **after** a loop can capture the conditions upon leaving the loop.



A Simple Example

$\{x \geq 0 \wedge y > 0\}$

while $x \geq y$ **do**

$\{x \geq 0 \wedge y > 0 \wedge x \geq y\}$

$x := x - y$

od

$\{x \geq 0 \wedge y > 0 \wedge x \not\geq y\}$

// or

$\{x \geq 0 \wedge y > 0 \wedge x < y\}$



More about the Example

We can say more about the program.

```
// may assume  $x, y := m, n$  here for some  $m \geq 0$  and  $n > 0$   
 $\{x \geq 0 \wedge y > 0 \wedge (x \equiv m \pmod{y})\}$   
while  $x \geq y$  do  
     $x := x - y$   
od  
 $\{x \geq 0 \wedge y > 0 \wedge (x \equiv m \pmod{y}) \wedge x < y\}$ 
```

Note: repeated subtraction is a way to implement the integer division. So, the program is taking the residue of x divided by y .



A Simple Programming Language

- 🌐 To study inference rules of Hoare logic, we consider a simple programming language with the following syntax for statements:

$$\begin{array}{l} S ::= \text{skip} \\ | x := E \\ | S_1; S_2 \\ | \text{if } B \text{ then } S \text{ fi} \\ | \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \\ | \text{while } B \text{ do } S \text{ od} \end{array}$$


Proof Rules

$$\frac{}{\{Q[E/x]\} x := E \{Q\}}$$

(Assignment)

$$\frac{}{\{P\} \text{skip} \{P\}}$$

(Skip)

$$\frac{\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}}{\{P\} S_1; S_2 \{R\}}$$

(Sequence)

$$\frac{\{P \wedge B\} S_1 \{Q\} \quad \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{Q\}}$$

(Conditional)

“if B then S fi” can be treated as “if B then S else skip fi” or directly with the following rule:

$$\frac{\{P \wedge B\} S \{Q\} \quad P \wedge \neg B \rightarrow Q}{\{P\} \text{if } B \text{ then } S \text{ fi } \{Q\}}$$

(If-Then)



Proof Rules (cont.)

$$\frac{\{P \wedge B\} S \{P\}}{\{P\} \text{ while } B \text{ do } S \text{ od } \{P \wedge \neg B\}} \quad \text{(while)}$$
$$\frac{P \rightarrow P' \quad \{P'\} S \{Q'\} \quad Q' \rightarrow Q}{\{P\} S \{Q\}} \quad \text{(Consequence)}$$

Note: with a suitable notion of validity, the set of proof rules up to now can be shown to be sound and (relatively) complete for programs that use only the considered constructs.

Some Auxiliary Rules

$$\frac{P \rightarrow P' \quad \{P'\} S \{Q\}}{\{P\} S \{Q\}}$$

(Strengthening Precondition)

$$\frac{\{P\} S \{Q'\} \quad Q' \rightarrow Q}{\{P\} S \{Q\}}$$

(Weakening Postcondition)

$$\frac{\{P_1\} S \{Q_1\} \quad \{P_2\} S \{Q_2\}}{\{P_1 \wedge P_2\} S \{Q_1 \wedge Q_2\}}$$

(Conjunction)

$$\frac{\{P_1\} S \{Q_1\} \quad \{P_2\} S \{Q_2\}}{\{P_1 \vee P_2\} S \{Q_1 \vee Q_2\}}$$

(Disjunction)



Invariants

- 🌐 An *invariant* at some point of a program is an assertion that holds whenever execution of the program reaches that point.
- 🌐 Assertion P in the rule for a while loop is called a *loop invariant* of the while loop.
- 🌐 An assertion is called an *invariant of an operation* (a segment of code) if, assumed true before execution of the operation, the assertion remains true after execution of the operation.
- 🌐 Invariants are a bridge between the **static text** of a program and its **dynamic computation**.



Program Annotation

- Inserting assertions/invariants in a program as comments helps understanding of the program.

$$\{x \geq 0 \wedge y > 0 \wedge (x \equiv m \pmod{y})\}$$

while $x \geq y$ **do**

$$\{x \geq 0 \wedge y > 0 \wedge x \geq y \wedge (x \equiv m \pmod{y})\}$$
$$x := x - y$$
$$\{y > 0 \wedge x \geq 0 \wedge (x \equiv m \pmod{y})\}$$

od

$$\{x \geq 0 \wedge y > 0 \wedge (x \equiv m \pmod{y}) \wedge x < y\}$$

- A correct annotation of a program can be seen as a partial proof outline for the program.
- Boolean assertions can also be used as an aid to program testing.



An Annotated Program

$\{x \geq 0 \wedge y \geq 0 \wedge \gcd(x, y) = \gcd(m, n)\}$

while $x \neq 0$ **and** $y \neq 0$ **do**

$\{x \geq 0 \wedge y \geq 0 \wedge \gcd(x, y) = \gcd(m, n)\}$

if $x < y$ **then** $x, y := y, x$ **fi**;

$\{x \geq y \wedge y \geq 0 \wedge \gcd(x, y) = \gcd(m, n)\}$

$x := x - y$

$\{x \geq 0 \wedge y \geq 0 \wedge \gcd(x, y) = \gcd(m, n)\}$

od

$\{(x = 0 \wedge y \geq 0 \wedge y = \gcd(x, y) = \gcd(m, n)) \vee$

$(x \geq 0 \wedge y = 0 \wedge x = \gcd(x, y) = \gcd(m, n))\}$

Note: m and n are two arbitrary non-negative integers, at least one of which is nonzero.



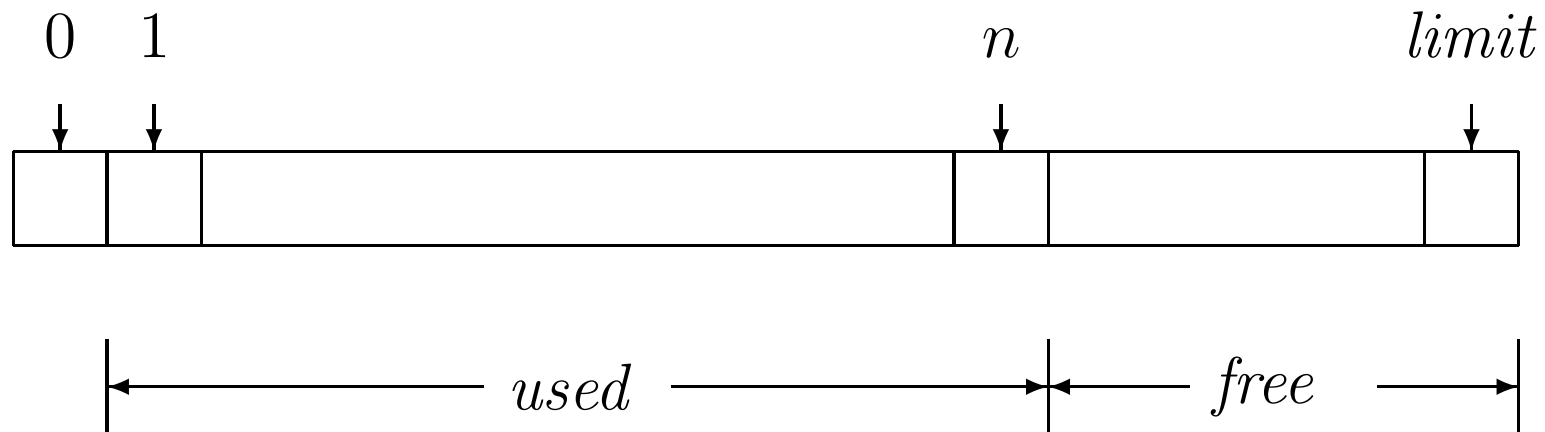
Programming with Invariants

- 🌐 Think about invariants at the beginning.
- 🌐 Invariants capture the dynamic computation that we intend to realize by the static text of a program.
- 🌐 They can guide us through the program development.



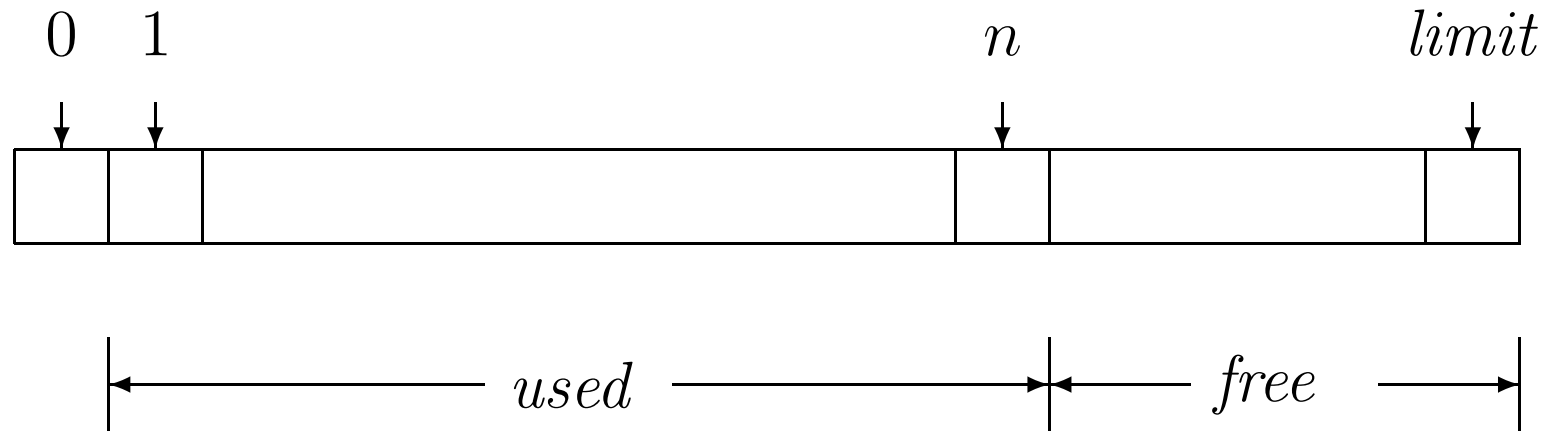
Developing a Search Program

- Consider a table that supports two operations: $insert(x)$ and $find(x)$.
- Elements are inserted from left to right, starting at position 1.



- An invariant for the table:
The elements of the table are stored in the sub-array $A[1..n]$, for $0 \leq n$, and $0 \leq n \leq limit$.

Developing a Search Program (cont.)



- 🌐 Operation $find(x)$ returns 0 if x is not in the table; otherwise, it returns the position in the table at which x was inserted most recently.

Developing a Search Program (cont.)

Initial code sketch:

initialization;

do the search;

*{ (x is not in the table, i.e., not in $A[1..n]$) or
(the most recent x is $A[i]$ and $0 < i \leq n$) }*

if *x is not in the table* **then**

found := 0

else

found := i

fi



Developing a Search Program (cont.)

Simplified computation of the result, with the help of a *sentinel* (an x stored in $A[0]$ before the search):

initialization; // set the sentinel here

do the search;

$\{ x \text{ equals } A[i] \text{ and } x \text{ is not in } A[i + 1..n] \text{ and } 0 \leq i \leq n \}$

$found := i$



Developing a Search Program (cont.)

Rewrite the assertion in a more formal way:

initialization; // set the sentinel here

do the search;

$\{x = A[i] \wedge x \notin A[i + 1..n] \wedge 0 \leq i \leq n\}$

found := *i*



Developing a Search Program (cont.)

Making the sentinel explicit:

$A[0] := x;$

further initialization;

$\{x = A[0] \wedge x \notin A[i + 1..n] \wedge 0 \leq i \leq n\}$

do the search;

$\{x = A[0] \wedge x \notin A[i + 1..n] \wedge 0 \leq i \leq n \wedge x = A[i]\}$

$found := i$



Developing a Search Program (cont.)

Refining the search step:

$A[0] := x;$

further initialization;

$\{x = A[0] \wedge x \notin A[i + 1..n] \wedge 0 \leq i \leq n\}$

while $x \neq A[i]$ **do**

$i := i - 1$

od;

$\{x = A[0] \wedge x \notin A[i + 1..n] \wedge 0 \leq i \leq n \wedge x = A[i]\}$

$found := i$



Developing a Search Program (cont.)

Final developed program fragment:

$A[0] := x;$

$i := n;$

$\{x = A[0] \wedge x \notin A[i + 1..n] \wedge 0 \leq i \leq n\}$

while $x \neq A[i]$ **do**

$i := i - 1$

od;

$\{x = A[0] \wedge x \notin A[i + 1..n] \wedge 0 \leq i \leq n \wedge x = A[i]\}$

$found := i$



Total Correctness: Termination

- 🌐 All inference rules introduced so far, except the **while** rule, work for total correctness.
- 🌐 Below is a rule for the total correctness of the **while** statement:

$$\frac{\{P \wedge B\} S \{P\} \quad \{P \wedge B \wedge t = Z\} S \{t < Z\} \quad P \rightarrow (t \geq 0)}{\{P\} \text{ while } B \text{ do } S \text{ od } \{P \wedge \neg B\}}$$

where t is an integer-valued expression (state function) and Z is a “rigid” variable that does not occur in P , B , t , or S .

- 🌐 The above function t is called a *rank* (or variant) function.

Termination of a Simple Program

```
 $g, p := 0, n; \quad // \ n \geq 1$   
while  $p \geq 2$  do  
     $g, p := g + 1, p - 1$   
od
```

- 🌐 Loop Invariant: $(g + p = n) \wedge (p \geq 1)$
- 🌐 Rank (Variant) Function: p
- 🌐 The loop terminates when $p = 1$ ($p \geq 1 \wedge p \not\geq 2$).



Well-Founded Sets

- 🌐 A binary relation $\preceq \subseteq A \times A$ is a **partial order** if it is
 - ☀️ reflexive: $\forall x \in A(x \preceq x)$,
 - ☀️ transitive: $\forall x, y, z \in A((x \preceq y \wedge y \preceq z) \rightarrow x \preceq z)$, and
 - ☀️ antisymmetric: $\forall x, y \in A((x \preceq y \wedge y \preceq x) \rightarrow x = y)$.
- 🌐 A partially ordered set (W, \preceq) is **well-founded** if there is no infinite decreasing chain $x_1 \succ x_2 \succ x_3 \succ \dots$ of elements from W . (Note: “ $x \succ y$ ” means “ $y \preceq x \wedge y \neq x$ ”.)
- 🌐 Examples:
 - ☀️ $(Z_{\geq 0}, \leq)$
 - ☀️ $(Z_{\geq 0} \times Z_{\geq 0}, \leq)$,
where $(x_1, y_1) \leq (x_2, y_2)$ if $(x_1 < y_1) \vee (x_1 = y_1 \wedge x_2 \leq y_2)$

Termination by Well-Founded Induction

Below is a more general rule for the total correctness of the **while** statement:

$$\frac{\{P \wedge B\} S \{P\} \quad \{P \wedge B \wedge \delta = D\} S \{\delta \prec D\} \quad P \rightarrow (\delta \in W)}{\{P\} \text{ while } B \text{ do } S \text{ od } \{P \wedge \neg B\}}$$

where (W, \preceq) is a well-founded set, δ is a state function, and D is a “rigid” variable ranged over W that does not occur in P , B , δ , or S .



Nondeterminism

🌐 Syntax of the Alternative Statement:

if $B_1 \rightarrow S_1$
 $\parallel B_2 \rightarrow S_2$
 ...
 $\parallel B_n \rightarrow S_n$
fi

Each of the “ $B_i \rightarrow S_i$ ”s is called a guarded command, where B_i is the guard of the command and S_i the body.

- 🌐 Semantic: one of the guarded commands, whose guard evaluates to true, is nondeterministically selected and its body executed. If none of the guards evaluates to true, then the execution aborts.

Rule for the Alternative Statement

The Alternative Statement:

if $B_1 \rightarrow S_1$
 $\parallel B_2 \rightarrow S_2$
 ...
 $\parallel B_n \rightarrow S_n$
fi

Inference rule:

$$\frac{P \rightarrow B_1 \vee \dots \vee B_n \quad \{P \wedge B_i\} S_i \{Q\}, \text{ for } 1 \leq i \leq n}{\{P\} \text{ if } B_1 \rightarrow S_1 \parallel \dots \parallel B_n \rightarrow S_n \text{ fi } \{Q\}}$$

References

- 🌐 [Apt and Olderog 1991] *Verification of Sequential and Concurrent Programs*, K.R. Apt and E.-R. Olderog, Springer-Verlag, 1991.
- 🌐 [Gries 1981] *The Science of Programming*, D. Gries, Springer-Verlag, 1981.
- 🌐 [Hoare 1969] “An axiomatic basis for computer program”, C.A.R. Hoare, *CACM*, 12(10):576–583, 1969”,
- 🌐 [Kleymann 1999] “Hoare logic and auxiliary variables”, T. Kleymann, *Formal Aspects of Computing*, 11:541–566, 1999.
- 🌐 [Sethi 199] *Programming Languages: Concepts and Constructs, 2nd Ed.*, R. Sethi, Addison-Wesley, 1996.

