Methods and Logics for Proving Programs [1]

Patrick COUSOT

École Polytechnique, LIX
91128 Palaiseau Cedex (France)

Contents

---

[1] Cousot, P.
Methods and Logics for Proving Program
In ``*Handbook of Theoretical Computer Science*'', J. van Leeuwen (Ed.), vol. B `` *Formal Models and Semantics*'', Ch. 15, pp. 843--993, Elsevier, 1990.

1. Introduction

Formalizing ideas of FLOYD [1967a] and NAUR [1966] which, in essence, were already present in GOLDSTINE & VON NEUMANN [1947] and TURING [1949] (as recalled in MORRIS & JONES [1984]), C. A. R. Hoare introduced in October 1969 an axiomatic method for proving that a program is partially correct with respect to a specification (HOARE [1969], see the genesis and reprint of this paper in HOARE & JONES [1989, pp. 45-58]). This paper introduced or revealed a number of ideas which originated an evolution of programming from arts and crafts to a science. Hoare logic had a very significant impact on program verification and design methods. It was an essential step in the emergence of "structured programming" in the 1970's. It is also an important contribution to the development of formal semantics of programming languages. Understanding that programs can be a subject of mathematical investigations was also crucial in the development of a theory of programming. This is reflected in the fact that HOARE [1969] is one of the most widely cited papers in computing science (see the bibliography of more than 350 references).

1.1 A brief review of HOARE [1969]

The "*introduction*" of HOARE [1969] claims that "computer programming is an exact science" and calls for the development of a formal system for reasoning about programs.

The first part of HOARE [1969] is an attempt to axiomatize "*computer arithmetic*" in two stages : first axioms are given for arithmetic operations on natural numbers which are valid independently of their computer representation (such as "x + y = y + x", "x + 0 = x", etc) and then choices of supplementary axioms are proposed for characterizing various possible implementations. For example a finite representation of natural numbers ("∀ x. 0 ≤ x ≤ maxint") can lead to various possible interpretations of overflow such as "¬ ∃ x. (x = maxint + 1)" i.e. program execution should be stopped, "maxint + 1 = maxint" i.e. the result of an overflowing operation should be taken as the maximum value represented or "maxint + 1 = 0" i.e. arithmetic operations should be computed modulo this maximum value "maxint".

The second part of HOARE [1969] introduces an axiomatic definition of "*program execution*". It defines Hoare correctness formulae "{P} C {Q}" where C is a program and the precondition P and postcondition Q are logical formulae describing properties of data manipulated by program C. Such a formula "{P} C {Q}" means that if execution of C is started in any memory state satisfying precondition P and if this execution does terminate then postcondition Q will be true upon completion (Hoare correctness formulae were originally written "P {C} Q" but are now written as "{P} C {Q}" to emphasize the rôle of assertions P and Q as comments). The main contribution of HOARE [1969] is the elucidation of a set of axioms and rules of inference which can be used in correctness proofs (termed "partial" since termination is not involved). For example if X is a variable identifier, E is an expression, $P[X \leftarrow E]$ is obtained from P by substituting E for all occurrences of X then "$\{P[X \leftarrow E]\}\ X := E\ \{P\}$" is an axiom for the assignment command "X := E". Intuitively, it simply states that what is true of expression E before the assignment is true of X after assignment of E to X. Another example is the sequential composition "C_1; C_2" of commands C_1 and C_2. From "$\{P\}\ C_1\ \{Q\}$" and "$\{Q\}\ C_2\ \{R\}$", one can infer "$\{P\}\ C_1;\ C_2\ \{R\}$". Intuitively this rule of inference states that if Q is true after execution of C_1 starting with P true and if R is true after execution of C_2 starting with Q true then R is true after the sequential execution of C_1 and C_2 starting with P true. This second part of HOARE [1969] ends with the formal partial correctness proof of a program for computing the Euclidian division of nonnegative integers by successive subtractions.

The third part of HOARE [1969] is on "*general reservations*" about this paper. First side-effects are excluded. Then, and most importantly, termination is not considered. Finally a number of languages features (such as procedures and parallelism) are omitted.

Then HOARE [1969] discusses "*proofs of program correctness*". An axiomatic approach is indispensable for achieving program reliability. The practicability of program proving is advocated in view of the cost of programming errors and program testing. It is also useful for program documentation and modification and to achieve portability (machine dependent part being clearly identified by the use of implementation dependent axioms). The difficulties such as unreliable specifications or proof complexity are also foreseen: "program proving, certainly at present, will be difficult even for programmers of high calibre; and may be applicable only to quite simple program designs".

Finally HOARE [1969] discusses "*formal language definition*". The axioms and rules of inference can be understood as "the ultimate definitive specification of the meaning of the language". The approach is simple. It can cope with the problem of machine dependence by leaving certain aspects of the language undefined and serve as a guide for language design.

HOARE [1969] last words in the "*acknowledgements*" anticipated the enormous work on Hoare logic: "The formal material presented here has only an expository status and represents only a minute proportion of what remains to be done. It is hoped that many of the fascinating problems involved will be taken up by others".

1.2 Further work by C. A. R. Hoare on Hoare logic

A number of these problems were treated by C. A. R. Hoare himself. A famous and non trivial proof, that of the program "Find" (HOARE [1961]), was later given in HOARE

[1971a]. This paper shows a significant move in the use of Hoare logic from a program verification method (i.e. an a posteriori proof of a complete program) to a program design method: "A systematic technique is described for constructing the program proof during the process of coding it". The use of data abstractions to handle complex data structures was introduced in HOARE [1972a] and exemplified in HOARE [1972b]. Other programming language features were covered later: procedures and parameters (HOARE [1971b]), including recursion (FOLEY & HOARE [1971]), jumps and functions (CLINT & HOARE [1972], ASHCROFT, CLINT & HOARE [1976]), parallel programs (HOARE [1972c], HOARE [1975]). Hoare logic had an important influence over the design of modern programming languages, Pascal notably (WIRTH [1971], HOARE & WIRTH [1973]). However progresses were sometimes slower than anticipated by HOARE [1969]. In the case of parallelism for example, synchronization (HOARE [1974]) and communication primitives (HOARE [1978b], HOARE [1985b]) had to be better understood before introducing formal proof methods (HOARE [1981], ZHOU CHAO CHEN & HOARE [1981]).

1.3 Further work by C. A. R. Hoare on methods of reasoning about programs

With regard to program proofs, HOARE [1969] suffers from a number of weaknesses, some of which were corrected later (HOARE & HE JIFENG [1986] [1987], HOARE, HE JIFENG & SANDERS [1987], HOARE, HAYES, HE JIFENG, MORGAN, ROSCOE, SANDERS, SORENSEN & SUFRIN [1987]). First of all, termination is not involved. For example "{true} while true do skip {Q}" holds for all assertions Q. This can be considered as a regrettable omission (see the "Envoi" to HOARE & JONES [1989, p. 391]) or as an historically fruitful simplification (since, for example, total correctness was not that simple to understand in the presence of nondeterminism, HOARE [1978a]). Later work by C. A. R. Hoare insists upon total correctness (HOARE [1981]). Second, the postcondition Q in Hoare correctness formulae "{P} C {Q}" is a predicate of the final states alone. Adopting postconditions which are predicates of initial and final states makes it easier to specify relations (as latter in HOARE & HE, JIFENG [1986] [1987], HOARE, HE JIFENG & SANDERS [1987], HOARE, HAYES, HE JIFENG, MORGAN, ROSCOE, SANDERS, SORENSEN & SUFRIN [1987]). To achieve the same effect in Hoare logic, one has to use so-called logical auxiliary variables such as x in "{X = x} C {X = x}" to express that execution of C leaves the value of the programming variable X unchanged (assuming that x does not appear in C). Third, Hoare correctness formulae "{P} C {Q}" do not describe the course of computation of the program C, a flaw in the presence of parallelism. For example "{X = x} C {X = x}" does not mean that X is not modified during execution of C so that we do not know whether "{X = x}[C || X := 1]{X = 1}" holds or not when command C is executed in parallel with assignment "X := 1". Sequences of intermediate states (more precisely messages) were latter used in HOARE [1984]. Fourth

Hoare logic imposes a standard form of reasoning about programs which, for example, lacks the flexibility of choosing between positive or contrapositive arguments or of choosing the most adequate form of induction (the only one allowed being structural induction upon the syntactical structure of programs). Fifth, the logical treatment of Hoare correctness formulae "{P} C {Q}", predicates P, Q and programs C are quite separated. For example predicates can be conjuncted but not Hoare correctness formulae. The use of names for designating objects obeys quite different conventions in predicates and programs. Finally Hoare logic was a very important step in the understanding that "computers are mathematical machines", "computer programs are mathematical expressions", that "a programming language is a mathematical theory" and that "programming is a mathematical activity" (HOARE [1985a]). However, the main emphasis of HOARE [1969] is on the logical and formal aspects of this mathematical activity. It obliterates the informal and often more elegant mathematical proofs which should also have had their useful counterparts in programming. This point of view is similar to that of a mathematician adhering to rigorous and well-understood proof methods without using exclusively formal mathematical logic.

With regard to program semantics, HOARE [1969] was somewhat optimistic: the axiomatic semantics can no longer be considered as universal but as one of the complementary definitions of programming languages (HOARE & LAUER [1974]).

1.4 Surveys on Hoare logic

Over the past twenty years, Hoare logic has been considerably studied. Most results have been reported in numerous surveys (LANGMAACK & OLDEROG [1980], APT [1981a], OLDEROG [1983b], APT [1984], CLARKE [1984], BARRINGER [1985], DE ROEVER [1985a], APT [1988], HOOMAN & DE ROEVER [1989]) and books (MANNA [1971], DE BAKKER [1980], LŒCKX & SIEBER [1984]), to cite a few. This chapter is an elementary but rigorous introduction to Hoare logic. We just assume some elementary familiarity with naïve logical and set theoretic notations and some very rudimentary practice of the so-called intermediate assertions method of Floyd but nevertheless do the necessary recalls. We hope that this survey will be useful to readers willing to understand the abundant and often very technical literature on Hoare logic. Numerous references are suggested for further study. We apologize to all those researchers who have been misunderstood or not referenced.

1.5 Summary

In paragraph § 2 we fix up the logical, set and order theoretic notations which are used subsequently.

In paragraph § 3 we define the syntax of while-programs, their operational semantics (that is a set of finite or infinite sequences of configurations representing successive observable states of an abstract machine executing the program for each possible input data) and their relational semantics (directly defining the relationship between initial and final configurations of terminating program executions).

In paragraph § 4 we define the partial correctness “{ p }C{ q }” of a command C with respect to a precondition p and postcondition q to mean that any terminating execution of C starting from an initial state s satisfying p must end in some final state s' satisfying q.

Paragraph § 5 is a presentation of Floyd-Naur partial correctness proof method. After giving a simple introductory example, we formally derive Floyd-Naur's method from the operational semantics using an elementary stepwise induction principle and predicates attached to program points to express invariant properties of programs. This systematic construction of the verification conditions ensures that the method is semantically sound (i.e. correct) and complete (i.e. always applicable). Then we explain a compositional presentation of Floyd-Naur's method inspired by Hoare logic where proofs are given by induction on the syntactical structure of programs. These two approaches are shown to be equivalent in the strong sense that, up to a difference of presentation, they require to verify exactly the same conditions. Few other partial correctness proof methods are shortly reviewed and shown also to be variants of the basic Floyd-Naur's method.

In paragraph § 6, we study total correctness (that is the conjunction of partial correctness, absence of runtime errors, deadlock freedom and termination) and more generally liveness proof methods. After giving a short mathematical recall on well founded relations, well orderings and ordinals, we review Floyd's well-founded set method and Burstall's intermittent assertion method.

Paragraph § 7 is devoted to Hoare logic for while-programs.

In paragraph 7.1, we first start from a semantical point of view (mathematicians would say a naïve set-theoretic approach) relative to a fixed interpretation specified by the operational semantics. Hoare logic is therefore understood as a set of general theorems for proving partial correctness by structural induction on the syntax of commands. This approach is sound (correct) and complete (always usable) with respect to the operational semantics. In practice, proofs can be presented informally using Owicki's proof outlines (that is by attaching comments to program points) which is equivalent to Floyd's method.

In paragraph 7.2, we study Hoare logic from a syntactical point of view (mathematicians would call it a logical approach) where predicates are required to be machine representable and proofs to be machine checkable (but not necessarily machine derivable). Therefore predicates P, Q and correctness formulae “{P} C {Q}” are now considered as strings of symbols written according to a precise syntax with no fixed meaning. Provability is formalized by rewriting rules for deriving valid theorems by successive transformations of given axioms. More precisely, we first define the syntax of first-order predicates P, Q (allowing only to quantify over elements, but not over subsets or functions) and correctness formulae "{P} C {Q}". Then we introduce Hilbert-like deductive systems (consisting of axioms such as "{P[X ← E]} X := E {P}" and rules of inference such as "from {P} C_1 {Q} and {Q} C_2 {R} infer {P} C_1; C_2 {Q}") and define formal proofs (i.e. finite sequences of formulae, each of which is either an axiom or else follows from earlier formulae by a rule of inference). This leads to Hoare's classical proof system $\mathbb{H}$ and to Hoare's proof system $\mathbb{H}'$ for proof outlines. Finally we define free and bound variables in predicates so as to precisely specify the syntactical rules of substitution P[X ← E].

Paragraph § 7.3 makes the link between the semantical point of view (i.e. the usual mathematical reasoning about <u>truth</u> with respect to the interpretation of programs specified by the operational semantics) and the syntactical point of view (where <u>provability</u> is understood as correctness formulae manipulations according to the formal rules of Hoare deductive system $\mathbb{H}$). For that purpose we define the semantics of predicates P, Q and correctness formulae "{P} C {Q}" in accordance with the relational semantics but parametrized by the meaning of basic symbols +, *, <,… which can be left unspecified.

Paragraph § 7.4 studies the link between truth (the semantical point of view) and provability (the syntactical point of view). Kurt Gödel showed that truth and provability do not necessarily coincide: provable implies true, refutable implies false but some formulae may be undecidable that is neither provable nor refutable (using proofs that can be checked mechanically) although they are either true or false. Therefore the question is whether Hoare's formal proof system captures the true partial correctness formulae, only these (soundness) and ideally all of these (completeness).

Soundness is proved in paragraph § 7.4.1.

Completeness and incompleteness issues for Hoare logic are discussed in paragraph § 7.4.2. It is shown that the formalism of while-programs is more powerful than the formalism of first-order predicates in the sense that, for example, the reflexive transitive closure of a given basic relation is easily defined by a program whereas it is not (in general) first-order definable (except when full arithmetic on natural numbers is available). Incompleteness results for Hoare logic follows since intermediate assertions needed in the correctness proofs of while loops cannot in general be expressed. The consequence is that although the logical language might be simple enough so that all true facts about data expressible in this language are machine derivable, some programs using the same basic symbols might be either unspecifiable or unprovable within this restricted language. It follows that the logical language must be enriched up to the point where arithmetic is included. Hence the logical language can be used to describe its own deductive system but then, as shown by Kurt Gödel, the deductive system is not powerful enough to prove all true facts expressible in the logical language. It follows that we can only prove Cook's relative completeness of Hoare logic assuming that the logical language is expressive enough to specify the intermediate assertions and that all needed mathematical facts about the data of the program are given. (The alternative which consists in using richer second-order logical systems is not considered since proofs would then not even be machine checkable). Expressiveness can be defined à la Clarke (using Dijkstra's weakest liberal preconditions) or à la Cook (using strongest liberal postconditions), both notions of expressiveness being equivalent. Examples of inexpressive (abacus arithmetic) and expressive (Peano arithmetic) first-order languages are given. It is shown that expressiveness is sufficient to obtain relative completeness but it is not necessary. Finally we study Clarke's characterization problem which consists in characterizing which programming languages have a sound and relatively complete Hoare logic. This is not the case of Algol-like or Pascal-like languages (since soundness and relative completeness of Hoare logic would imply the decidability of the halting problem for finite interpretations where variables can only take a finite number of values). The intuitive reason is again that first-order logical languages are less powerful than Algol or Pascal-like languages which can manipulate a potentially infinite run-time stack.

In paragraph § 8, we briefly consider complements on Hoare logic: data structures (§ 8.1), undefinedness due to runtime errors (§ 8.3), aliasing and side effects (§ 8.4), block structured local variables (§ 8.5), goto statements (§ 8.6), functions and expressions with side effects (§ 8.7), coroutines (§ 8.8) and a guide to the literature on examples of program verification (§ 8.11) and other logics extending first-order logic with programs (§ 8.12). Three topics are treated more extensively:

In paragraph § 8.2 we consider procedures. First we define the syntax and relational semantics of recursive parameterless procedures. Then we consider partial correctness proofs based upon computation induction (a generalization of Scott induction) which leads to Hoare's recursion rule. Since this only rule is not complete, we consider Park's fixpoint induction, which, using auxiliary variables, can be indirectly transcribed into Hoare's rule of adaptation. Then these rules are generalized for value-result parameters and numerous examples of application are provided. References to the literature are given for variable parameters and procedures as parameters.

Parallel programs are handled in paragraph § 8.9. First we define the syntax and operational semantics of parallel programs with shared variables and await commands. We review a number of à la Floyd proof methods for such parallel programs using the unifying point of view of paragraph § 5 where proof methods are shown to derive from a single induction principle and only differ by the way of decomposing the global invariant on memory states and control states (or auxiliary variables) into local invariants. In particular we study in detail the Lamport and Owicki & Gries method as well as various strengthened or weakened versions, its stepwise (à la Floyd) and syntax-directed (à la Hoare) presentations. This proof method is formalized by Owicki & Gries logic and a compositional version is given as introduced by Stirling following Jones. We end by a guide to the literature on Hoare logics for communicating sequential processes.

In paragraph § 8.10 we consider proof systems for total correctness. Such proof systems cannot be of pure first-order logical character and must incorporate an external well-founded relation. We first consider Harel proof rule for while-programs with finitely bounded nondeterminism and its arithmetical completeness. Then we explain the use of transfinite ordinals to deal with unbounded nondeterminism, a situation where a program state may have infinitely many possible successor states. This leads to the total correctness of fair parallel programs. Finally we introduce Dijkstra's weakest preconditions calculus for unbounded nondeterminism.

Numerous references to the literature are given at paragraph § 9.

1.6 Hints for reading this survey

This chapter can be read in a number of ways. In all cases, proofs should be omitted on first reading. Here are two examples of non sequential readings:

- Readers purely interested in Hoare logic can read as follows: § 2 (logical, set and order theoretic notations), § 3 (syntax and semantics of the programming language), § 4 (partial correctness of a command), § 5.1 (an example of partial correctness proof by Floyd-Naur method), § 5.3 (a Hoare style presentation of Floyd-Naur partial correctness proof method by syntax-directed induction), § 7.1.1 (general theorems for proof construction), § 7.2.1 (syntax of predicates and correctness formulae), § 7.2.2 (deductive systems and formal proofs), § 7.2.3 (Hoare's proof system $\mathbb{H}$), § 7.2.5 (syntactical rules of substitution). Then the various complements on Hoare logic of paragraph § 8 can be read in any order or readers more interested in the soundness and completeness problems can go on by § 7.1.2 (semantical soundness and completeness), § 7.3 (the semantics of Hoare logic), § 7.4 (the link between syntax and semantics : soundness and completeness issues in Hoare logic).
- Readers interested in liveness proof methods but willing to ignore logic can read successively § 2 (logical, set and order theoretic notations), § 3 (syntax and semantics of the programming language), § 4 (partial correctness of a command), § 5 (Floyd-Naur partial correctness proof method and some equivalent variants), § 6 (liveness proof methods), § 8.9.1 (operational semantics of parallel programs with shared variables), § 8.9.2 (à la Floyd proof methods for parallel programs with shared variables), § 8.10.4 (Dijkstra's weakest preconditions calculus).

2. Logical, set and order theoretic notations

For terms of logical character we use the following notations : tt denotes *truth*, ff *falsity*, $\neg$ *negation*, $\wedge$ *conjunction*, $\vee$ *inclusive disjunction*, $\Rightarrow$ *logical implication*, = (or $\Leftrightarrow$) *logical equivalence*, $\forall$ v. p *universal quantification* (p is true for any v), $\exists$ v. p *existential quantification* (there are some v such that p is true), $\exists!$ v. p *unique existential quantification* (there is a unique v such that p is true). $\forall v \in E$. p is an abbreviation for $\forall$ v .(($v \in E$) $\Rightarrow$ p) and $\exists v \in E$. p is $\exists$ v. (($v \in E$) $\wedge$ p). $\forall v_1, \ldots, v_n \in E$. p is an abbreviation for $\forall v_1 \in E. \ldots \forall v_n \in E$. p. The same way $\exists v_1, \ldots, v_n \in E$. p is an abbreviation for $\exists v_1 \in E. \ldots \exists v_n \in E$. p. We write $(B \rightarrow X \lozenge Y)$ to denote X when B is true and Y otherwise.

$\mathbb{Z}$ (respectively $\mathbb{N}$, $\mathbb{N}^+$) is the set of integers (positive, strictly positive integers). As usual in computer science, + is the addition, - the subtraction, * the product, *div* the division, *mod* the modulus and ** the exponentiation of integers. *odd*(x) (respectively *even*(x)) is true if and only if x is an odd (respectively even) integer.

We accept the intuitive concept of a *set* as a collection of objects called *elements* of the set. The notation $e \in E$ means that e is an element of the set E and $e \notin E = \neg(e \in E)$. The void set is denoted $\emptyset$. $E \subset E'$, $E \subseteq E'$ and $E = E'$ respectively denote *proper inclusion*, *inclusion* and *equality* of sets and $(E \supset E') = (E' \subset E)$, $(E \supseteq E') = (E' \subseteq E)$ and $(E \neq E') = \neg(E = E')$. We use the set theoretic operations $\cup$ (*union*), $\cap$ (*intersection*) and - (*difference*). If a set D is fixed then for subsets E of D the *complement* $\neg E$ of E is (D - E). If E is a set then $\mathcal{P}(E)$ (called the *power set* of E) denotes the set of all subsets of E. If $E_0, \ldots, E_{n-1}$ are sets, the *Cartesian product* $E_0 \times \ldots \times E_{n-1}$ is the set of n-tuples $<e_0, \ldots, e_{n-1}>$, with $e_i \in E_i$ for $i = 0, \ldots, n-1$. In symbols : $E_0 \times \ldots \times E_{n-1} = \{<e_0, \ldots, e_{n-1}> : e_0 \in E_0 \wedge \ldots \wedge e_{n-1} \in E_{n-1}\}$ where ":" reads "for those which satisfy". The *projection* $<e_0, \ldots, e_{n-1}>_i$ is the i-th component e_i of the n-tuple $<e_0, \ldots, e_{n-1}>$. If $n > 1$, the *elimination* $<e_0, \ldots, e_{n-1}>_{\sim i}$ is the $(n-1)$-tuple $<e_0, \ldots, e_{i-1}, e_{i+1}, \ldots, e_{n-1}>$. If $E_0 = \ldots = E_{n-1} = E$ then $E^n = E_0 \times \ldots \times E_{n-1}$. We define E^0 to be $\{\emptyset\}$ and identify E^1 with E. The cardinality of a set E is denoted $| E |$ hence if E is finite, $| E |$ is the number of elements of E.

Given a positive integer n and a set E, we define an *n-ary relation* r on E as a subset of E^n. n is called the arity of r and is denoted #r. We say that r is *binary* when #r = 2 and often use the infix notation x r y for $<x, y> \in r$. For example we write $x \leq y$ to mean that x is less than or equal to y.

If $r, r' \subseteq E \times E$ are binary relations on E then $r \circ r' = \{<e, e'> : \exists e'' \in E. <e, e''> \in r \wedge <e'', e'> \in r'\}$ is the *product* of r and r', $r^{-1} = \{<e', e> : <e, e'> \in r\}$ is the *inverse* of r. If, moreover, $p \subseteq E$ is a subset of E then $p \rceil r = \{<e, e'> \in r : e$

$\in$ p} is the *left restriction* of r to p and r $\lceil$ p = {<e, e'> $\in$ r : e' $\in$ p} is the *right restriction* of r to p. The equality relation on E also called the *diagonal* of E^2 is δ = {<e, e> : e $\in$ E}.

The *power* of a binary relation r on a set E is defined by recurrence as $r^0 = \delta$, $r^{n+1} = r^n \circ r$ for n $\geq$ 0. The (*strict*) *transitive closure* r^+ of r is $r^+ = \cup\{r^n : n > 0\}$ and the *reflexive transitive closure* r^* of r is $r^* = r^0 \cup r^+$.

A *partially ordered set* is a pair <E, ≤> where E is a nonvoid set and ≤ is a binary relation on E which is *reflexive* ($\forall$ a $\in$ E. a ≤ a), *antisymmetric* ($\forall$ a, b $\in$ E.(a ≤ b $\wedge$ b ≤ a) $\Rightarrow$ (a = b)) and transitive ($\forall$ a, b, c $\in$ E.(a ≤ b $\wedge$ b ≤ c) $\Rightarrow$ (a ≤ c)). Given P $\subseteq$ E, a $\in$ E is an *upper bound* of P if $\forall$ b $\in$ P. b ≤ a; a is the *least upper bound* of P (in symbols *lub* P) if a is an upper bound of P and if b is any upper bound of P then a ≤ b. If *lub* P exists, then it is unique. *Lower bounds* and the *glb* (*greatest lower bound*) are defined dually, that is by replacing ≤ by its inverse (also called *dual*) ≥ defined by (a ≥ b) = (b ≤ a). A *complete lattice* is a partially ordered set <E, ≤> such that *lub* P and *glb* P exist for all P, P $\subseteq$ E. It follows that E has a greatest element or *supremum* $\top$ = *lub* E = *glb* $\emptyset$ and a least element or *infimum* $\perp$ = *glb* E = *lub* $\emptyset$. <$\mathcal{P}$(E), $\subseteq$> is a complete lattice such that *lub* = $\cup$, *glb* = $\cap$, $\top$ = E and $\perp$ = $\emptyset$.

Given two sets A and B and a binary relation ϕ on A $\cup$ B, we call ϕ a *function* of A into B (and write ϕ : A $\rightarrow$ B) if (<a, b> $\in$ ϕ) $\Rightarrow$ (a $\in$ A $\wedge$ b $\in$ B) $\wedge$ ($\forall$ a $\in$ A. $\exists$! b $\in$ B.<a, b> $\in$ ϕ). A = *dom* ϕ is called the *domain* and B = *rng* ϕ is the *range* of ϕ. We use the functional notation b = ϕ(a) instead of <a, b> $\in$ ϕ. The set of all functions of A into B will be denoted B^A or more frequently A $\rightarrow$ B.

We specify a function ϕ : A $\rightarrow$ B without giving it a name using the *lambda notation* "λ x : A $\rightarrow$ B. e" where the expression e is such that $\forall$ a $\in$ A. ϕ(a) = e(a). We write λ x $\in$ A. e (respectively λ x. e) when B (respectively A and B) can easily be inferred from e. If ϕ : A $\rightarrow$ B then ϕ[a $\leftarrow$ b] is the function ϕ' : A $\cup$ {a} $\rightarrow$ B $\cup$ {b} such that ϕ'(a) = b and $\forall$ a' $\in$ A .((a' $\neq$ a) $\Rightarrow$ (ϕ'(a') = ϕ(a'))). A function $\phi : \{a_1, \ldots, a_n\} \rightarrow \{b_1, \ldots, b_n\}$ such that $\phi(a_i) = b_i$ for i = 1, …, n will be simply written $[a_1 \leftarrow b_1, \ldots, a_n \leftarrow b_n]$ so that $[a_1 \leftarrow b_1, \ldots, a_n \leftarrow b_n](a_i) = b_i$.

A *family* $\gamma = \langle\gamma_i : i \in I\rangle$ of elements of E is a function ϕ from the *index set* I into the set E, where $\gamma_i = \phi(i)$. When I $\subseteq$ $\mathbb{N}$, γ is called a *sequence* of elements of E. More precisely, if I = $\emptyset$ then γ is called an *empty sequence* and is written ε. When I = $\mathbb{N}$, it is an *infinite sequence* and we write $\gamma = \gamma_0, \ldots, \gamma_i, \ldots$. When n $\in$ $\mathbb{N}$ and I = {i : 0 ≤ i < n}, γ is called a *finite sequence* of *length* n and we write $\gamma = \gamma_0, \ldots, \gamma_{n-1}$. We define seq^n E to be the set of sequences of elements of E of length n ≥ 0, seq^* E = $\cup_{n \geq 0}$ seq^n E, seq^ω E to be the set of infinite sequences of elements of E and *seq* E = seq^* E $\cup$ seq^ω E. Moreover, the *concatenation* γe *of* e $\in$ E *to the right of* γ is defined by εe = e, $\gamma e = \gamma_0, \ldots, \gamma_{n-1}, e$ if γ is a finite sequence of length n and γe = γ if γ is an infinite sequence.

Let <A, ≤> and <B, $\angle$> be partially ordered sets. ϕ : A $\rightarrow$ B is *monotone* (or *increasing*) if $\forall$ a, b $\in$ E. (a ≤ b) $\Rightarrow$ (ϕ(a) $\angle$ ϕ(b)). x $\in$ A is a *fixpoint* of ϕ : A $\rightarrow$ A if ϕ(x) = x. It is a *prefixpoint* if x ≤ ϕ(x) and a *postfixpoint* if ϕ(x) ≤ x. Let <L, ≤> be a complete lattice with infimum $\perp$ and $\phi : L \to L$ be monotone. The set $\{x \in L : \phi(x) = x\}$ of fixpoints of ϕ is a (nonempty) complete lattice for $\leq$ with infimum $\mathit{lfp}\ \phi = \mathit{glb}\{x \in L : \phi(x) \leq x\}$ and supremum $\mathit{gfp}\ \phi = \mathit{lub}\{x \in L : x \leq \phi(x)\}$ (TARSKI [1955]). An *increasing chain* is a sequence γ of elements of A such that $\forall\ i \in \mathit{dom}\ \gamma - \{0\}.\ \gamma_{i-1} \leq \gamma_i$. $\phi : L \to L$ is *upper-continuous* if $\mathit{lub}\{\phi(\gamma_i) : i \in \mathit{dom}\ \gamma\} = \phi(\mathit{lub}\{\gamma_i : i \in \mathit{dom}\ \gamma\})$ for any increasing chain γ of L. If ϕ is upper-continuous then $\mathit{lfp}\ \phi = \mathit{lub}\{\phi^n(\perp) : n \geq 0\}$ where $\forall\ x \in L.\ \phi^0(x) = x$ and $\forall\ n \in \mathbb{N}.\ \forall\ x \in L.\ \phi^{n+1}(x) = \phi(\phi^n(x))$, (KLEENE [1952]). In particular, if r is a binary relation r on a set E then r^* is uniquely determined by the facts that $r^* = \delta \cup r \circ r^*$ (or $r^* = \delta \cup r^* \circ r$) and if $x \subseteq E \times E$ is such that $x = \delta \cup r \circ x$ (or $x = \delta \cup x \circ r$) then $r^* \subseteq x$ that is $r^* = \mathit{lfp}\ \lambda\ x : E^2 \to E^2.\ \delta \cup r \circ x$ (or $r^* = \mathit{lfp}\ \lambda\ x : E^2 \to E^2.\ \delta \cup x \circ r$). If $\langle L, \leq, \perp\rangle$ is a complete lattice, $\langle P, \angle\rangle$ is a poset with infimum $\perp'$, $\alpha : L \to P$ is strict ($\alpha(\perp) = \perp'$), upper-continuous and $\alpha \circ \phi = \psi \circ \alpha$ and moreover $\phi : L \to L$ and $\psi : P \to P$ are monotone then $\alpha(\mathit{lfp}\ \phi) = \mathit{lfp}\ \psi$.

3. Syntax and semantics of the programming language

Since Hoare logic is closely bounded to a programming language, we introduce a very simple Pascal-like (WIRTH [1971]) language. We first define its *syntax* that is the set of well-formed programs. We use an *abstract syntax* describing the structure of programs by trees (McCARTHY [1963]) and leave unspecified the *concrete syntax* (where programs are linear strings of tokens to be parsed unambiguously (AHO, SETHI & ULLMAN [1986])). Then we define the *operational semantics* of the language that is the effect of executing syntactically correct programs. Traditionally, one imagines the program running on an abstract machine with primitive instructions (NEUHOLD [1971]). Its execution steps can be idealized by mapping abstract syntactic constructs in the program to a transition relation on configurations (KELLER [1976]). The language is non-deterministic since variables can be assigned random values so that a configuration may have several possible successors. An execution of the program is a finite or infinite sequence of configurations representing successive observable states of the machine during execution. When observation of intermediate configurations is unnecessary, we use a *relational semantics* directly defining the relationship between initial and final configurations of program executions (HOARE & LAUER [1974]). This relational semantics is latter used as a basis for justifying Hoare's logic (HOARE [1969]). This *axiomatic semantics* associates an axiom or a rule of inference with each kind of basic or structured statement of the language which states what we may assert after execution of that statement in terms of what was true beforehand.

Such complementary definitions of programming language semantics (HOARE & LAUER [1974], DONAHUE [1976], APT & PLOTKIN [1986]) are useful for describing the semantics at various levels of details. This is a natural extension of FLOYD [1967a] and HOARE [1969] primitive idea that "the specification of proof techniques provides an adequate formal definition of a programming language" where "an" had to be replaced by "one of" because not all methods have the same power of expression.

3.1 Syntax

Basic commands of our programming language are the null command "skip" and the assignment "X := E" of the value of an expression E to a programming variable X or the nondeterministic assignment "X := ?" of a random value to X. Commands C_1, C_2 can be composed sequentially "(C_1; C_2)" and conditionally "(B $\rightarrow$ C_1 $\Diamond$ C_2)" according to the value of a Boolean expression B. A command C can also be iterated "(B * C)" while B holds.

This *abstract syntax* is more formally defined below. For the time being, the sets $\mathbb{Pvar}$ of programming variables (ranged over by X), $\mathbb{Expr}$ of expressions (ranged over by E) and $\mathbb{Bexp}$ of Boolean expressions (ranged over by B) are assumed to be given. $\mathbb{Expr}$ and $\mathbb{Bexp}$ will be detailed later. The set $\mathbb{Com}$ of commands (ranged over by C) is the smallest set closed under the given formation rule (similar to a context-free grammar expressed in Backus-Naur Form (BNF, NAUR [1960])):

DEFINITION *Syntax of commands*

X : $\mathbb{Pvar}$	*Programming variables*	
E : $\mathbb{Expr}$	*Expressions*	
B : $\mathbb{Bexp}$	*Boolean expressions*	
C : $\mathbb{Com}$	*Commands*	(1)

$$C ::= \text{skip} \mid X := E \mid X := ? \mid (C_1 ; C_2) \mid (B \rightarrow C_1 \lozenge C_2) \mid (B * C)$$

When necessary we omit parentheses and use a Pascal-like *concrete syntax*.

Example *A program to compute x ** y* (2)

The program with concrete syntax :

(3)
```
Z := 1;
while Y<>0 do
 if odd(Y) then
  begin Y := Y - 1; Z := Z * X end
 else
  begin Y := Y div 2; X := X * X end;
```

has the following abstract syntax:

(4)
```
(Z := 1; (Y<>0 * (odd(Y) → (Y := Y-1; Z := Z*X) ◊ (Y := Y div 2; X := X*X))))
```
■

The components of a command C are C itself and the commands appearing in C together with their components. Two instances of the same command C' in a command C should be considered as different components of C. Therefore we mark components C' ∈ $\mathbb{Comp}⟦C⟧$ of a command C by the Dewey number designating the root of the subtree of C' in the syntactic tree of C. For example, $\mathbb{Comp}⟦((X := 1; X := 1); X := 1)⟧$ = {$((X := 1; X := 1); X := 1)^{\varepsilon}$, $(X := 1; X := 1)^{0}$, $X := 1^{00}$, $X := 1^{01}$, $X := 1^{1}$} so that "$X := 1^{00}$" is the first, "$X := 1^{01}$" the second and "$X := 1^{1}$" the third instance of assignment command "X := 1" in the sequence "((X := 1; X := 1); X := 1)". More formally:

DEFINITION *Components of a command* (5)

$$\mathbb{Comp}^{\eta}[\![C]\!] \quad = \quad \{C^{\eta}\} \qquad \text{if C is skip, X := E or X := ?} \qquad (.1)$$

$$\mathbb{Comp}^{\eta}[\![(C_1;\ C_2)]\!] \quad = \quad \{(C_1;\ C_2)^{\eta}\} \cup \mathbb{Comp}^{\eta 0}[\![C_1]\!] \cup \mathbb{Comp}^{\eta 1}[\![C_2]\!] \qquad (.2)$$

$$\mathbb{Comp}^{\eta}[\![(B \to C_1 \lozenge C_2)]\!] \quad = \quad \{(B \to C_1 \lozenge C_2)^{\eta}\} \cup \mathbb{Comp}^{\eta 0}[\![C_1]\!] \cup \mathbb{Comp}^{\eta 1}[\![C_2]\!] \qquad (.3)$$

$$\mathbb{Comp}^{\eta}[\![(B * C)]\!] \quad = \quad \{(B * C)^{\eta}\} \cup \mathbb{Comp}^{\eta 0}[\![C]\!] \qquad (.4)$$

so that :

$$\mathbb{Comp}[\![C]\!] \quad = \quad \mathbb{Comp}^{\varepsilon}[\![C]\!]. \qquad (.5)$$

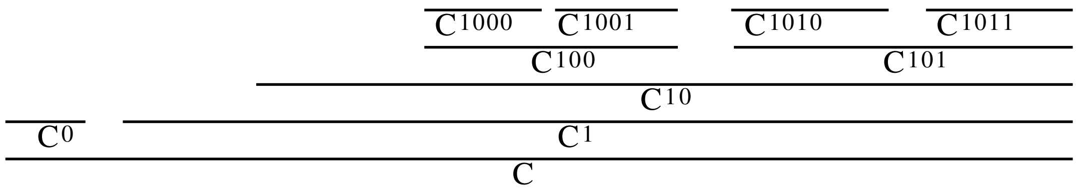
$$\mathbb{Loops}[\![C]\!] \quad = \quad \mathbb{Comp}[\![C]\!] \cap \{(B{*}C')^{\eta} : B \in \mathbb{Bexp} \wedge C' \in \mathbb{Com} \wedge \eta \in \mathit{seq}^{*}\{0, 1\}\} \qquad (.6)$$

$$\mathbb{Comp} \quad = \quad \{\mathbb{Comp}[\![C]\!] : C \in \mathbb{Com}\} \qquad (.7)$$

$$\mathbb{Loops} \quad = \quad \{\mathbb{Loops}[\![C]\!] : C \in \mathbb{Com}\} \qquad (.8)$$

Example *Components of program (4)* (6)

```
(Z := 1; (Y <> 0 * (odd(Y)  →  (Y := Y-1; Z := Z*X) ◊ (Y := Y div 2; X := X*X))))
```



■

When there is no ambiguity, the exponent η of a component C^{η} will be omitted and we identify $\mathbb{Comp}$ with $\mathbb{Com}$.

3.2 Operational semantics

A *state* (or *valuation*) is a function s with domain a given set $\mathbb{Var}$ of variables (including the set $\mathbb{Pvar}$ of programming variables) and range a nonempty set *D* of objects (called *data* in computer science). s(v) is called the *value* of variable v in state s:

d	:	D	*Data*	(7)
s	:	$S = \mathbb{Var} \to D$	*States*	(8)

The *operational semantics* of a command C is a model of its execution. Such an execution will be understood as a finite or infinite sequence "$\gamma_0, \ldots, \gamma_n, \ldots$" of *configurations* such that $<\gamma_n, \gamma_{n+1}> \in op[\![C]\!]$ and $op[\![C]\!]$ is the operational transition relation. The initial configuration γ_0 has the form $<s_0, C>$ where s_0 records the initial values of variables. Configuration γ_n has the form $<s_n, C_n>$ where s_n records the current values of variables after execution of n program steps and C_n records the command remaining to be executed. If execution terminates in configuration γ_l then $\gamma_l = s_l$ where s_l records the final values of variables. In particular, $<<s, C>, <s', C'>> \in op[\![C]\!]$ means that one step of execution of C in state s can lead to state s' with C' being the remainder of C to be executed. If $\langle\langle s, C\rangle, s'\rangle \in op[\![C]\!]$ then execution of C in state s can lead to state s' in one step and is then terminated. Hence we have:

γ	:	$\Gamma = (S \times \mathbb{Com}) \cup S$	*Configurations*	(9)
op	:	$\mathbb{Com} \rightarrow \mathcal{P}(\Gamma \times \Gamma)$	*Operational transition relation*	(10)

The definition of the semantics of expressions will be postponed. For the time being, we will assume that the semantics of expressions is given: if $E \in \mathbb{Expr}$ and $s \in S$ then $\underline{E}(s) = I[\![E]\!](s) \in D$ is the value of expression E in state s. The same way, if $B \in \mathbb{Bexp}$ then $I[\![B]\!]$ also written $\underline{B}$ is the set of states s such that B holds in state s:

I	:	$\mathbb{Expr} \rightarrow (S \rightarrow D)$,	$\underline{E} = I[\![E]\!]$	*Semantics of expressions*	(11)
I	:	$\mathbb{Bexp} \rightarrow \mathcal{P}(S)$,	$\underline{B} = I[\![B]\!]$	*Semantics of Boolean expressions*	(12)

The *operational transition relation* is defined by structural induction on the abstract syntax of commands (in the style of PLOTKIN [1981] but using a direct recursive definition) as follows :

DEFINITION Operational transition relation (13)

$op[\![\text{skip}]\!] = \{\langle\langle s, \text{skip}\rangle, s\rangle : s \in S\}$ (.1)

$op[\![X := E]\!] = \{\langle\langle s, X := E\rangle, s[X \leftarrow \underline{E}(s)]\rangle : s \in S\}$ (.2)

$op[\![X := ?]\!] = \{\langle\langle s, X := ?\rangle, s[X \leftarrow d]\rangle : s \in S \wedge d \in D\}$ (.3)

$op[\![(C_1 ; C_2)]\!] = \{\langle\langle s,(C_1';C_2)\rangle,\langle s',(C_1'';C_2)\rangle\rangle : \langle\langle s,C_1'\rangle,\langle s',C_1''\rangle\rangle \in op[\![C_1]\!]\}$ (.4)
$\cup \{\langle\langle s, (C_1'; C_2)\rangle,\langle s', C_2\rangle\rangle : \langle\langle s, C_1'\rangle, s'\rangle \in op[\![C_1]\!]\}$
$\cup\ op[\![C_2]\!]$

$op[\![(B \rightarrow C_1 \lozenge C_2)]\!] = \{\langle\langle s, (B \rightarrow C_1 \lozenge C_2)\rangle, \langle s, C_1\rangle\rangle : s \in \underline{B}\} \cup op[\![C_1]\!]$ (.5)
$\cup \{\langle\langle s, (B \rightarrow C_1 \lozenge C_2)\rangle, \langle s, C_2\rangle\rangle : s \notin \underline{B}\} \cup op[\![C_2]\!]$

$op[\![(B * C)]\!] = \{\langle\langle s, (B * C)\rangle, \langle s, (C; (B * C))\rangle\rangle : s \in \underline{B}\}$ (.6)
$\cup \{\langle\langle s, (C'; (B * C))\rangle, \langle s', (C''; (B * C))\rangle\rangle :$
$\langle\langle s, C'\rangle, \langle s', C''\rangle\rangle \in op[\![C]\!]\}$
$\cup \{\langle\langle s, (C'; (B * C))\rangle, \langle s', (B * C)\rangle\rangle : \langle\langle s, C'\rangle, s'\rangle \in op[\![C]\!]\}$
$\cup \{\langle\langle s, (B * C)\rangle, s\rangle : s \notin \underline{B}\}$

Example *Operational semantics of program (4)* (14)

Using the components (5) of program (4) given at example (2), we can define labels:

(15)

$L_1 = C^\varepsilon = C$, $L_2 = C^1$, $L_3 = (C^{10}; C^1)$, $L_4 = (C^{100}; C^1)$, $L_5 = (C^{1001}; C^1)$, $L_6 = (C^{101}; C^1)$, $L_7 = (C^{1011}; C^1)$, including a final label L_8 represented by the symbol "√"

so as to name program points as follows:

(16)

```
(Z := 1; (Y <> 0 * (odd(Y)  →  (Y := Y-1; Z := Z*X) ◊ (Y := Y div 2; X := X*X))))
|L1      |L2      |L3         |L4       |L5           |L6            |L7         |L8
```

A label L can also be understood as designating the command remaining to be executed when control is at that point L. According to definition (13), the operational transition relation of program (4) is:

$$\begin{aligned} op[\![C]\!] \quad = \quad & \{\langle\langle s, L_1\rangle, \langle s[Z \leftarrow 1], L_2\rangle\rangle : s \in S\} \\ \cup\ & \{\langle\langle s, L_2\rangle, \langle s, L_3\rangle\rangle : s(Y) \neq 0\} \\ \cup\ & \{\langle\langle s, L_3\rangle, \langle s, L_4\rangle\rangle : \mathit{odd}(s(Y))\} \\ \cup\ & \{\langle\langle s, L_4\rangle, \langle s[Y \leftarrow s(Y) - 1], L_5\rangle\rangle : s \in S\} \\ \cup\ & \{\langle\langle s, L_5\rangle, \langle s[Z \leftarrow s(Z) * s(X)], L_2\rangle\rangle : s \in S\} \\ \cup\ & \{\langle\langle s, L_3\rangle, \langle s, L_6\rangle\rangle : \mathit{even}(s(Y))\} \\ \cup\ & \{\langle\langle s, L_6\rangle, \langle s[Y \leftarrow s(Y)\ \mathit{div}\ 2], L_7\rangle\rangle : s \in S\} \\ \cup\ & \{\langle\langle s, L_7\rangle, \langle s[X \leftarrow s(X) * s(X)], L_2\rangle\rangle : s \in S\} \\ \cup\ & \{\langle\langle s, L_2\rangle, s\rangle : s(Y) = 0\} \end{aligned} \tag{17}$$

A possible execution sequence of this program is therefore as follows:

$\langle[X \leftarrow 3, Y \leftarrow 2, Z \leftarrow 0], L_1\rangle$, $\langle[X \leftarrow 3, Y \leftarrow 2, Z \leftarrow 1], L_2\rangle$, $\langle[X \leftarrow 3, Y \leftarrow 2, Z \leftarrow 1], L_3\rangle$, $\langle[X \leftarrow 3, Y \leftarrow 2, Z \leftarrow 1], L_6\rangle$, $\langle[X \leftarrow 3, Y \leftarrow 1, Z \leftarrow 1], L_7\rangle$, $\langle[X \leftarrow 9, Y \leftarrow 1, Z \leftarrow 1], L_2\rangle$, $\langle[X \leftarrow 9, Y \leftarrow 1, Z \leftarrow 1], L_3\rangle$, $\langle[X \leftarrow 9, Y \leftarrow 1, Z \leftarrow 1], L_4\rangle$, $\langle[X \leftarrow 9, Y \leftarrow 0, Z \leftarrow 1], L_5\rangle$, $\langle[X \leftarrow 9, Y \leftarrow 0, Z \leftarrow 9], L_2\rangle$, $\langle[X \leftarrow 9, Y \leftarrow 0, Z \leftarrow 9]\rangle$. ■

3.3 Relational semantics

The *relational semantics* or interpretation $I[\![C]\!]$ (also noted $\underline{C}$) of a command C is a relation between states such that $\langle s, s'\rangle \in \underline{C}$ if and only if an execution of command C started in initial state s <u>may</u> terminate into final state s' :

DEFINITION HOARE & LAUER [1974] *Relational semantics* (18)

$$I \quad : \quad \mathbb{Com} \rightarrow \mathcal{P}(S \times S)$$

$$I[\![C]\!] = \underline{C} = \{\langle s, s'\rangle : \langle\langle s, C\rangle, s'\rangle \in op[\![C]\!]^*\}$$

The relational semantics can be characterized as follows :

THEOREM HOARE & LAUER [1974], GREIF & MEYER [1981] (19)

$\underline{\text{skip}}$	$=$	$\{\langle s, s\rangle : s \in S\}$	(.1)
$\underline{X := E}$	$=$	$\{\langle s, s[X \leftarrow \underline{E}(s)]\rangle : s \in S\}$	(.2)
$\underline{X := ?}$	$=$	$\{\langle s, s[X \leftarrow d]\rangle : s \in S \wedge d \in D\}$	(.3)
$\underline{(C_1 ; C_2)}$	$=$	$\underline{C_1} \circ \underline{C_2}$	(.4)
$\underline{(B \rightarrow C_1 \lozenge C_2)}$	$=$	$(\underline{B} \rceil \underline{C_1}) \cup (\neg\underline{B} \rceil \underline{C_2})$	(.5)
$\underline{(B * C)}$	$=$	$(\underline{B} \rceil \underline{C})^* \lceil \neg\underline{B}$	(.6)
	$=$	$\mathit{lfp}\, \lambda X. (\delta \lceil \neg\underline{B}) \cup ((\underline{B} \rceil \underline{C}) \circ X)$	(.7)

$\underline{(B * C)}$ is the unique $r \subseteq S^2$ such that: (.8)

- . $r \subseteq (S \times \neg\underline{B})$ (.8.a)
- . $\forall q \subseteq S. (q \rceil (\underline{B} \rceil \underline{C}) \subseteq S \times q) \Rightarrow (q \rceil r \subseteq S \times q)$ (.8.b)
- . $((\underline{B} \rceil \underline{C}) \circ r) \subseteq r$ (.8.c)
- . $(\delta \lceil \neg\underline{B}) \subseteq (\delta \cap r)$ (.8.d)

Example *Calculus of the relational semantics of a simple program* (20)

Assume $D = \mathbb{Z}$, the relational semantics of C = (Y <> 0 * Y := Y - 1) is $\underline{C} = \{\langle s, s[Y \leftarrow 0]\rangle : s(Y) \geq 0\}$.

To prove this, observe that by (19.7), $\underline{C} = \mathit{lfp}\ F$ where $F(X) = (\delta \lceil \neg\underline{Y{<>}0}) \cup ((\underline{Y{<>}0} \rceil \underline{Y := Y\text{-}1}) \circ X) = \{\langle s, s\rangle : s(Y) = 0\} \cup \{\langle s, s[Y \leftarrow s(Y) - 1]\rangle : s[Y] \neq 0\} \circ X$. Define $X^i = F^i(\emptyset)$. We have $X^0 = \emptyset$, $X^1 = \{\langle s, s[Y \leftarrow 0]\rangle : s(Y) = 0\}$. Assume by induction hypothesis that $X^i = \{\langle s, s[Y \leftarrow 0]\rangle : 0 \leq s(Y) \leq i - 1\}$. Then $X^{i+1} = F(X^i) = \{\langle s, s\rangle : s(Y) = 0\} \cup \{\langle s, s[Y \leftarrow s(Y) - 1]\rangle : s[Y] \neq 0\} \circ \{\langle s, s[Y \leftarrow 0]\rangle : 0 \leq s(Y) \leq i - 1\} = \{\langle s, s\rangle : s(Y) = 0\} \cup \{\langle s, s'\rangle : \exists s''. s[Y] \neq 0 \wedge s'' = s[Y \leftarrow s(Y) - 1] \wedge 0 \leq s''(Y) \leq i - 1 \wedge s' = s''[Y \leftarrow 0]\} = \{\langle s, s\rangle : s(Y) = 0\} \cup \{\langle s, s[Y \leftarrow s(Y) - 1][Y \leftarrow 0]\rangle : s[Y] \neq 0 \wedge 0 \leq s[Y \leftarrow s(Y) - 1](Y) \leq i - 1\} = \{\langle s, s\rangle : s(Y) = 0\} \cup \{\langle s, s[Y \leftarrow 0]\rangle : 1 \leq s(Y) \leq i\} = \{\langle s, s[Y \leftarrow 0]\rangle : 0 \leq s(Y) \leq i\}$. It follows that $\underline{C} = \mathit{lfp}\ F = \bigcup_{i \geq 0} X^i = \{\langle s, s[Y \leftarrow 0]\rangle : \exists i \geq 0.\ 0 \leq s(Y) \leq i\} = \{\langle s, s[Y \leftarrow 0]\rangle : 0 \leq s(Y)\}$. ■

Proof of theorem 10:

For short we write *op* instead of *op*[C] when C is clear from the context.

- (19.1), (19.2) : $\underline{\text{skip}}$ and $\underline{X := E}$ are handled the same way as :
- (19.3) : $\underline{X := ?} = \{\langle s, s'\rangle : \langle\langle s, X := ?\rangle, s'\rangle \in op^*\}$ [by (18)] $= \{\langle s, s'\rangle : \langle\langle s, X := ?\rangle, s'\rangle \in \delta \cup op \circ op^*\}$ [since $r^* = \delta \cup r \circ r^*$] $= \{\langle s, s'\rangle : \exists \gamma \in \Gamma. \langle\langle s, X := ?\rangle, \gamma\rangle \in op \wedge \langle\gamma, s'\rangle \in op^*\}$ [since $\langle s, X := ?\rangle \neq s'$] $= \{\langle s, s'\rangle : d \in D \wedge \langle s[X \leftarrow d], s'\rangle \in op^*\}$ [by (13.3)] $= \{\langle s, s[X \leftarrow d]\rangle : s \in S \wedge d \in D\}$ [since by (13), $\langle\gamma'', \gamma'\rangle \in op$ implies $\gamma'' \notin S$ so that $\langle s'', s'\rangle \in op^*$ if and only if $s'' = s' \in S$].

- (19.4) : $<s, s'> \in \underline{(C_1; C_2)} \Leftrightarrow <<s, (C_1; C_2)>, s'> \in op^*$ [by (18)] $\Leftrightarrow (\exists s''. <<s, C_1>, s''> \in op^* \wedge <<s'', C_2>, s'> \in op^*)$ [by (13.4)] $\Leftrightarrow (\exists s''. <s, s''> \in \underline{C_1} \wedge <s'', s'> \in \underline{C_2})$ [by (18)] $\Leftrightarrow <s, s'> \in \underline{C_1} \circ \underline{C_2}$ [by definition of $\circ$].
- (19.5) : $\underline{(B \to C_1 \lozenge C_2)} = \{<s, s'> : \exists \gamma \in \Gamma. <<s, (B \to C_1 \lozenge C_2)>, \gamma> \in op \wedge <\gamma, s'> \in op^*\} = \{<s, s'> : s \in \underline{B} \wedge <<s, C_1>, s'> \in op^*\} \cup \{<s, s'> : s \notin \underline{B} \wedge <<s, C_2>, s'> \in op^*\} = (\underline{B} \rceil \underline{C_1}) \cup (\neg\underline{B} \rceil \underline{C_2})$ [by (13.5)].
- (19.6) : $<s, s'> \in (\underline{B} \rceil \underline{C})^* \lceil \neg\underline{B}$ if and only if there are $n \geq 1$ and $s_1, \ldots, s_n \in S$ such that $s = s_1$ and for all $i = 1, \ldots, n - 1$, $s_i \in \underline{B}$ and $<s_i, s_{i+1}> \in \underline{C}$ and $s_n = s' \notin \underline{B}$ that is if and only if there is an execution sequence of the form "$<s_1, (B * C)>, <s_1, (C; (B * C))>, \ldots, <s_2, (B * C)>, <s_2, (C; (B * C))>, \ldots, <s_n, (B * C)>, s_n$" with $s = s_1, \ldots, s_{n-1} \in \underline{B}$ and $s_n = s' \notin \underline{B}$ hence if and only if $<s, s'> \in \underline{(B * C)}$.
- (19.7) : Let $F = \lambda X. \delta \cup (\underline{B} \rceil \underline{C}) \circ X$ and $G = \lambda X. (\delta \lceil \neg\underline{B}) \cup (\underline{B} \rceil \underline{C}) \circ X$. We have $F^0(\emptyset) \lceil \neg\underline{B} = G^0(\emptyset) = (\delta \lceil \neg\underline{B})$. Assume by induction hypothesis that $F^n(\emptyset) \lceil \neg\underline{B} = G^n(\emptyset)$ then $F^{n+1}(\emptyset) \lceil \neg\underline{B} = F(F^n(\emptyset)) \lceil \neg\underline{B} = (\delta \lceil \neg\underline{B}) \cup ((\underline{B} \rceil \underline{C}) \circ F^n(\emptyset) \lceil \neg\underline{B}) = (\delta \lceil \neg\underline{B}) \cup ((\underline{B} \rceil \underline{C}) \circ G^n(\emptyset)) = G^{n+1}(\emptyset)$. Since $r^* = \mathit{lfp}\, \lambda x. \delta \cup r \circ x$, it follows that $\underline{(B * C)} = (\underline{B} \rceil \underline{C})^* \lceil \neg\underline{B} = (\mathit{lfp}\, F) \lceil \neg\underline{B} = (\bigcup_{n\geq 0} F^n(\emptyset)) \lceil \neg\underline{B} = \bigcup_{n\geq 0} (F^n(\emptyset) \lceil \neg\underline{B}) = \bigcup_{n\geq 0} G^n(\emptyset) = \mathit{lfp}\, G$.
- (19.8) : We first show that $\underline{(B * C)}$ satisfies conditions (a)-(d) :
  - (a) $\underline{(B * C)} = (\underline{B} \rceil \underline{C})^* \lceil \neg\underline{B} \subseteq S \times \neg\underline{B}$.
  - (b) If $(q \rceil r \subseteq S \times q)$ then by induction on $n \geq 0$, $(q \rceil r^n \subseteq S \times q)$. This is true for $n = 0$ since $q \rceil \delta = \delta \lceil q \subseteq S \times q$. Moreover $q \rceil r^{n+1} = q \rceil (r^n \circ r) = (q \rceil r^n) \circ r \subseteq (S \times q) \circ r = S^2 \circ (q \rceil r) \subseteq S^2 \circ (S \times q) = S \times q$. Whence if $(q \rceil (\underline{B} \rceil \underline{C}) \subseteq S \times q)$ then $(q \rceil (\underline{B} \rceil \underline{C})^n \subseteq S \times q)$ so that $q \rceil \underline{(B * C)} = q \rceil (\underline{B} \rceil \underline{C})^* \lceil \neg\underline{B} \subseteq q \rceil (\underline{B} \rceil \underline{C})^* = q \rceil (\bigcup_{n\geq 0} (\underline{B} \rceil \underline{C})^n) = \bigcup_{n\geq 0} (q \rceil (\underline{B} \rceil \underline{C})^n) \subseteq \bigcup_{n\geq 0} S \times q = S \times q$.
  - (c) Since $\underline{(B * C)} = (\delta \lceil \neg\underline{B}) \cup ((\underline{B} \rceil \underline{C}) \circ \underline{(B * C)})$, we have $(\underline{B} \rceil \underline{C}) \circ \underline{(B * C)} \subseteq \underline{(B * C)}$.
  - (d) Since $\underline{(B * C)} = (\delta \lceil \neg\underline{B}) \cup ((\underline{B} \rceil \underline{C}) \circ \underline{(B * C)})$ we have $(\delta \lceil \neg\underline{B}) \subseteq \underline{(B * C)}$. so that $(\delta \lceil \neg\underline{B}) = \delta \cap (\delta \lceil \neg\underline{B}) \subseteq \delta \cap \underline{(B * C)}$.
- Assuming that r satisfies conditions (19.8), we show that $r = \underline{(B * C)}$:
  - $(\delta \lceil \neg\underline{B}) = \delta \cap r \subseteq r$ and $(\underline{B} \rceil \underline{C}) \circ r \subseteq r$ so that r is a postfixpoint of $F = \lambda X. (\delta \lceil \neg\underline{B}) \cup ((\underline{B} \rceil \underline{C}) \circ X)$ whence $\underline{(B * C)} = \mathit{lfp}\, F \subseteq r$.
  - To show that $r \subseteq \underline{(B * C)}$, we assume that $<s, s'> \in r$ and prove that $<s, s'> \in (\underline{B} \rceil \underline{C})^* \lceil \neg\underline{B}$. Let $q = \{s_1 \in S : <s, s_1> \in (\underline{B} \rceil \underline{C})^*\}$. We have $q \rceil (\underline{B} \rceil \underline{C}) = \{<s_1, s_2> \in (\underline{B} \rceil \underline{C}) : <s, s_1> \in (\underline{B} \rceil \underline{C})^*\} \subseteq \{<s_3, s_2> : <s, s_2> \in (\underline{B} \rceil \underline{C})^+\} \subseteq \{<s_3, s_2> : <s, s_2> \in (\underline{B} \rceil \underline{C})^*\} \subseteq S \times q$. By (19.8.b), $(q \rceil r \subseteq S \times q)$ so that $s \in q$ and $<s, s'> \in r$ imply $s' \in q$ whence $<s, s'> \in (\underline{B} \rceil \underline{C})^*$. Moreover $<s, s'> \in r$ and (19.8.a) imply $s' \in \neg\underline{B}$, so that $<s, s'> \in (\underline{B} \rceil \underline{C})^* \lceil \neg\underline{B}$. ■

Observe that the relational semantics (19) of our language $\mathbb{Com}$ is not "equivalent" to its operational semantics (13) because information about program termination is lost.

Example *Programs with different operational semantics but identical relational semantics* (21)

Assuming $D = \mathbb{Z}$, C = "Y := 0" and C' = "(Y := ?; (Y <> 0 * Y := Y - 1))" have the same relational semantics $\underline{C} = \underline{C'} = \{<s, s[Y \leftarrow 0]> : s \in S\}$. We have $\underline{C} = \{<s, s[Y \leftarrow 0]> : s \in S\}$ by (19.2) and $\underline{C} = \underline{Y := ?} \circ \underline{(Y <> 0 * Y := Y\text{-}1)}$ [by 10.4] = $\{<s, s[Y \leftarrow d]> : s \in S \wedge d \in D\} \circ \{<s, s[Y \leftarrow 0]> : 0 \le s(Y)\}$ [by (19.3) and example (20)] = $\{<s, s[Y \leftarrow d][Y \leftarrow 0]> : 0 \le d\} = \{<s, s[Y \leftarrow 0]> : s \in S\}$. Hence no distinction is made between program C for which termination with Y = 0 is guaranteed and program C' for which termination with Y = 0 is possible but not guaranteed (with a usual Pascal-like implementation). ■

It follows that if we choose (19) (or latter Hoare logic) as <u>the</u> definition of the semantics of $\mathbb{C}\text{om}$ then a faithful implementation of $\mathbb{C}\text{om}$ should ignores the possibility of nontermination as a viable answer whenever termination is possible. This "angelic" nondeterminism of FLOYD [1967b] could be implemented by parallelism or breadth search to simultaneously examine all possible choices offered (we have to assume in a finite number) by random assignments (HAREL [1979]) The "demonic" nondeterminism of DIJKSTRA [1975] [1976] is a radically different alternative where results are valid only if the program examined always terminates. Again a strictly faithful implementation should use depth-first backtracking to guarantee nontermination if this is possible for one choice in random assignments. In practice, nondeterminism is implemented by choosing arbitrarily or sometimes fairly one of the alternatives offered by (19). Then angelic and demonic nondeterminism can be understood as describing the best and worst possible situation (HOARE [1978a], JACOBS & GRIES [1985]). In conclusion (19) is an approximate version of (13) where "details" about termination are deliberately ignored.

4. Partial correctness of a command

An *assertion* is a set of states. A *specification* is a pair <p, q> of assertions on states (where p is called the *input specification* or *precondition* and q is the *output specification* or *postcondition*). A command C is said to be *partially correct* with respect to a specification <p, q> (written $\{ p \} C \{ q \}$) if any terminating execution of C starting from an initial state s satisfying p must end in some final state s' satisfying q. Stated in terms of the relational semantics (18), this means that $(p \rceil \underline{C}) \subseteq (S \times q)$:

DEFINITION FLOYD [1967a], NAUR [1966] *Partial correctness* (22)

p, q	:	$\mathbb{Ass} = \mathcal{P}(S)$	*Assertions*
<p, q>	:	$\mathbb{Spec} = \mathbb{Ass} \times \mathbb{Ass}$	*Specifications*
$\{ p \} C \{ q \}$	:	$\mathbb{Ass} \times \mathbb{Com} \times \mathbb{Ass} \to \{tt, ff\}$	*Partial correctness*
$\{ p \} C \{ q \}$	=	$(p \rceil \underline{C}) \subseteq (S \times q)$	

Example *Partial correctness of program (4)* (23)

Assume for program (4) that $\mathbb{Var}$ is {X, Y, Z, x, y} and *D* is the set $\mathbb{Z}$ of integers. This program is partially correct with respect to the specification <p, q> such that:

$$p = \{s \in S : s(X) = s(x) \wedge s(Y) = s(y) \geq 0\}$$
$$q = \{s \in S : s(Z) = s(x) ** s(y)\}$$

Otherwise stated, if x and y ≥ 0 respectively denote the initial values of programming variables X, Y and if the corresponding execution terminates then the final value of Z is equal to x ** y. ■

Observe that definition (22) of partial correctness is essentially of semantical nature. It is relative to the operational semantics (13) and is defined in terms of naïve set theory. No reference is made to a particular formal logical language for describing the sets of states p and q (called “assertions” for convenience). Therefore we make no difference between a predicate P (such as “Y ≥ 0 ∧ Z * (X ** Y) = x ** y”) and the assertion $\underline{P}$ which its denote (that is “{s ∈ S : s(Y) ≥ 0 ∧ s(Z) * (s(X) ** s(Y)) = s(x) ** s(y)}” in this example). The consequences of restricting assertions p, q to those that can be formally described by first-order predicates will be studied later at paragraph § 7.2.

5. Floyd-Naur partial correctness proof method and some equivalent variants

The first method for proving partial correctness was proposed by FLOYD [1967a] and NAUR [1966]. After giving a simple introductory example, we formally derive Floyd-Naur's method from the operational semantics (13) using an elementary stepwise induction principle and predicates attached to program points to express invariant properties of programs. This systematic construction of the verification conditions ensures that the method is semantically sound (i.e. correct) and complete (i.e. always applicable). Then we introduce another presentation of Floyd-Naur's method inspired by Hoare logic where proofs are given by induction on the syntactical structure of programs. These two approaches are shown to be equivalent in the strong sense that, up to a difference of presentation, they require to verify exactly the same conditions. Few other partial correctness proof methods are shortly reviewed and shown also to be variants of the basic Floyd-Naur's method.

5.1 An example of partial correctness proof by Floyd-Naur method

Example *Partial correctness proof of program (4)* (24)

• The informal partial correctness proof of program (4) by Floyd-Naur's method first consists in discovering predicates $P_k(X, Y, Z, x, y)$ associated with each label L_k, $k = 1, \ldots, 8$ which should relate the values X, Y, Z of variables `X`, `Y`, `Z` whenever control is at L_k to the initial values x, y, z of these variables:

```
{L1}  { P1(X, Y, Z, x, y) = (X = x ∧ Y = y ≥ 0) }                              (25)
      Z := 1;
{L2}  { P2(X, Y, Z, x, y) = (Y ≥ 0 ∧ Z * (X ** Y) = x ** y) }  {-- loop invariant --}
      while Y <> 0 do
{L3}      { P3(X, Y, Z, x, y) = (Y > 0 ∧ Z * (X ** Y) = x ** y) }
          if odd(Y) then
             begin
{L4}            { P4(X, Y, Z, x, y) = (Y > 0 ∧ Z * (X ** Y) = x ** y) }
                Y := Y - 1;
{L5}            { P5(X, Y, Z, x, y) = (Y ≥ 0 ∧ Z * (X ** (Y + 1)) = x ** y) }
                Z := Z *X;
             end
          else
             begin
{L6}            { P6(X, Y, Z, x, y) = (even(Y) ∧ Y > 0 ∧ Z * (X ** Y) = x ** y) }
```

```
                    Y := Y div 2;
{L7}                { P7(X, Y, Z, x, y) = (Y ≥ 0 ∧ Z * (X ** (2 * Y)) = x ** y) }
                    X := X * X;
                end;
{L8}   { P8(X, Y, Z, x, y) = (Z = x ** y) }
```

- Observe that predicates P_k associated with labels L_k, k = 1, …, 8 can be understood as describing a set of states $\underline{P}_k$. For example:

$$\underline{P}_2 = \{s \in S : s(Y) \geq 0 \wedge s(Z) * (s(X) ** s(Y)) = s(x) ** s(y)\}$$

- Then it must be shown that these predicates satisfy *verification conditions*, which can be stated informally as follows:

(ε)	$p \Rightarrow P_1(X, Y, Z, x, y)$	where p is the input specification	(26)
(i_1)	$P_1(X, Y, Z, x, y) \Rightarrow P_2(X, Y, 1, x, y)$		
(i_2)	$[P_2(X, Y, Z, x, y) \wedge Y \neq 0] \Rightarrow P_3(X, Y, Z, x, y)$		
(i_3)	$[P_2(X, Y, Z, x, y) \wedge Y = 0] \Rightarrow P_8(X, Y, Z, x, y)$		
(i_4)	$[P_3(X, Y, Z, x, y) \wedge odd(Y)] \Rightarrow P_4(X, Y, Z, x, y)$		
(i_5)	$[P_3(X, Y, Z, x, y) \wedge even(Y)] \Rightarrow P_6(X, Y, Z, x, y)$		
(i_6)	$P_4(X, Y, Z, x, y) \Rightarrow P_5(X, Y - 1, Z, x, y)$		
(i_7)	$P_5(X, Y, Z, x, y) \Rightarrow P_2(X, Y, Z * X, x, y)$		
(i_8)	$P_6(X, Y, Z, x, y) \Rightarrow P_7(X, Y \text{ div } 2, Z, x, y)$		
(i_9)	$P_7(X, Y, Z, x, y) \Rightarrow P_2(X * X, Y, Z, x, y)$		
(σ)	$P_8(X, Y, Z, x, y) \Rightarrow q$	where q is the output specification	

- These verification conditions imply that the predicates are *local invariants*, that is P_k holds whenever control is at point L_k, that is to say, according to (15), when command L_k remains to be executed. In practice it is only necessary to discover loop invariants since other local invariants can be derived from the loop invariants using these verification conditions.
- It follows that the *local invariants* on states attached to program points are equivalent to the following *global invariant* on configurations: (27)

$$i = \bigcup_{\kappa=1}^{7} \{<s, L_k> : s \in \underline{P}_k\} \cup \underline{P}_8$$

i is called *invariant* because it is always true during execution:

$$(\{<s, C> : s \in p\} \rceil op[\![C]\!]^*) \subseteq i$$

It follows immediately that:

$$(p \rceil \underline{C}) \subseteq (S \times q)$$

■

5.2 The stepwise Floyd-Naur partial correctness proof method

A partial correctness proof can always be organized in the same way and reduced to the discovery of local invariants which are then shown to satisfy elementary verification conditions corresponding to elementary program steps. To show this, we first introduce an induction principle expressing the essence of invariance proofs. Then we specialize the induction principle for the operational semantics (13) of language $\mathbb{Com}$. This consists in representing the global invariant on configurations by local invariants on states attached to program points. Once properties of programs have been chosen to be expressed in this way, Floyd's verification conditions can be derived by calculus from the operational semantics (13). This construction of the verification conditions for local invariants a priori ensures semantical soundness and completeness of the proof method.

5.2.1 Stepwise induction principle

Floyd-Naur's method is usually understood as stepwise induction (MANNA, NESS & VUILLEMIN [1972]): to prove that some property i of a program is invariant during the course of the computation, it is sufficient to check that i is true when starting the computation and to show that if i is true at one step of the computation, it remains true after the next step. This means that Floyd-Naur's method consists in applying the following lemma to the operational semantics:

LEMMA COUSOT [1981] *Stepwise induction principle* (28)

if $p, p', q \in \mathcal{P}(E)$ and $r \in \mathcal{P}(E \times E)$ then:

$$[(p \rceil r^* \lceil p') \subseteq (E \times q)] \Leftrightarrow [\exists i \in \mathcal{P}(E). (p \subseteq i) \wedge (i \rceil r \subseteq E \times i) \wedge (i \cap p' \subseteq q)]$$

Proof

- For $\Rightarrow$, we observe that $i = \{e \in E : \exists e' \in p. \langle e', e\rangle \in r^*\}$ satisfies conditions $p \subseteq i$ and $i \rceil r \subseteq E \times i$ whereas $p \rceil r^* \lceil p' \subseteq E \times q$ implies $i \cap p' \subseteq q$.
- For $\Leftarrow$, we observe that, by induction on $n \geq 0$, $p \subseteq i$ and $i \rceil r \subseteq E \times i$ imply that $p \rceil r^n \subseteq E \times i$ whence $p \rceil r^* \subseteq E \times i$ so that $p \rceil r^* \lceil p' \subseteq E \times (i \cap p') \subseteq E \times q$. ■

Floyd-Naur partial correctness proof method consists in discovering local assertions on states attached to program points which must be shown to satisfy local verification conditions. As shown by example (24), this can be understood as the discovery of a global assertion i upon configurations which is shown to satisfy a *global verification condition* $\mathrm{gvc}[\![C]\!][p,q](i)$ derived from lemma (28) :

THEOREM KELLER [1976], PNUELI [1977], COUSOT [1981] *Induction principle for Floyd-Naur's stepwise partial correctness proof method* (29)

$\underline{\{}\ p\ \underline{\}C\{}\ q\ \underline{\}} = [\exists\, i \in \mathcal{P}(\Gamma).\ \mathrm{gvc}[\![C]\!][p, q](i)]$ (.1)

where $\mathrm{gvc}[\![C]\!][p, q](i) = (\forall\, s \in p. \langle s, C\rangle \in i) \wedge (i \rceil op[\![C]\!] \subseteq \Gamma \times i) \wedge (i \cap S \subseteq q)$ (.2)

Proof

- $\underline{\{}\ p\ \underline{\}C\{}\ q\ \underline{\}} = (p \rceil \underline{C} \subseteq S \times q)$ [by (22)] $= (p \rceil \{\langle s', s\rangle : \langle\langle s', C\rangle, s\rangle \in op[\![C]\!]^*\} \subseteq S \times q)$ [by (18)] $= (\{\langle s, C\rangle : s \in p\} \rceil op[\![C]\!]^* \lceil S \subseteq \Gamma \times q) = (\exists\, i \in \mathcal{P}(\Gamma).\ (\{\langle s, C\rangle : s \in p\} \subseteq i) \wedge (i \rceil op[\![C]\!] \subseteq \Gamma \times i) \wedge (i \cap S \subseteq q))$ [by (28)] $= (\exists\, i \in \mathcal{P}(\Gamma).(\forall\, s \in p.\langle s, C\rangle \in i) \wedge (i \rceil op[\![C]\!] \subseteq \Gamma \times i) \wedge (i \cap S \subseteq q))$. ■

Any i satisfying the verification condition $\mathrm{gvc}[\![C]\!][p, q](i)$ is always true during execution hence is a *global invariant*.. Such a global invariant always exists since there is a *strongest global invariant* which implies all others and can be characterized as a fixpoint:

THEOREM PARK [1969], CLARKE [1979b], COUSOT [1981] *Fixpoint characterization of the strongest global invariant* (30)

The strongest global invariant

$I = \{\gamma : \exists\, s \in p.\ \langle\langle s, C\rangle, \gamma\rangle \in op[\![C]\!]^*\}$ (.1)

is such that:

$I = \mathit{lfp}\ \lambda X : \mathcal{P}(\Gamma).\ \{\langle s, C\rangle : s \in p\} \cup \{\gamma : \exists\, \gamma'.\ \langle\gamma', \gamma\rangle \in X \rceil op[\![C]\!]\}$ (.2)

if $\underline{\{}\ p\ \underline{\}C\{}\ q\ \underline{\}}$ then $\mathrm{gvc}[\![C]\!][p, q](I)$ holds and $\forall\, i.\ \mathrm{gvc}[\![C]\!][p, q](i) \Rightarrow (I \subseteq i)$. (.3)

Proof

- $\langle\mathcal{P}(\Gamma), \subseteq, \emptyset\rangle$ is a complete lattice. $\phi = \lambda X : \mathcal{P}(\Gamma \times \Gamma).\ \delta \cup X \circ op[\![C]\!]$ and $\psi = \lambda X : \mathcal{P}(\Gamma).\ \{\langle s, C\rangle : s \in p\} \cup \{\gamma : \exists\, \gamma'.\ \langle\gamma', \gamma\rangle \in X \rceil op[\![C]\!]\}$ are monotone. $\alpha = \lambda X : \mathcal{P}(\Gamma \times \Gamma).\{\gamma \in \mathcal{P}(\Gamma) : \exists\, s \in p.\ \langle\langle s,C\rangle, \gamma\rangle \in X\}$ is strict, upper-continuous and $\alpha \circ \phi = \psi \circ \alpha = \lambda X.\ \{\gamma \in \mathcal{P}(\Gamma) : \exists\, s \in p.\ \langle\langle s, C\rangle, \gamma\rangle \in \delta \cup X \circ op[\![C]\!]\}$. Therefore $\alpha(\mathit{lfp}\ \phi) = \alpha(op[\![C]\!]^*) = I = \mathit{lfp}\ \psi$.
- Obviously $\{\langle s, C\rangle : s \in p\} \subseteq I$. Moreover $I \rceil op[\![C]\!] = \{\langle\gamma, \gamma'\rangle : \exists\, s \in p.\ \langle\langle s, C\rangle, \gamma\rangle \in op[\![C]\!]^* \wedge \langle\gamma, \gamma'\rangle \in op[\![C]\!]\} \subseteq \{\langle\gamma, \gamma'\rangle : \exists\, s \in p.\ \langle\langle s, C\rangle, \gamma'\rangle \in op[\![C]\!]^+\} \subseteq \Gamma \times I)$. Finally $\underline{\{}\ p\ \underline{\}C\{}\ q\ \underline{\}} \Rightarrow (p \rceil \underline{C} \subseteq S \times q)$ [by (22)] $\Rightarrow (p \rceil \{\langle s', s\rangle : \langle\langle s', C\rangle, s\rangle \in op[\![C]\!]^*\} \subseteq S \times q)$ [by (18)] $\Rightarrow (I \lceil S \subseteq \Gamma \times q) \Rightarrow (I \cap S \subseteq q)$.
- if $(\forall\, s \in p.\ \langle s, C\rangle \in i) \wedge (i \rceil op[\![C]\!] \subseteq \Gamma \times i)$ then $\psi(i) \subseteq i$ so that $I = \mathit{lfp}\ \psi = \cap\{X \in \mathcal{P}(\Gamma) : \psi(X) \subseteq X\}$ [by TARSKI [1955]] $\subseteq i$. ■

Example *Application of induction principle (29) to the correctness proof of program (4)* (31)

Let us go on with example (24) and show that (26) is equivalent to (29):

- $\forall$ s $\in$ p. <s, C_1> $\in$ i is equivalent to p $\subseteq$ $\underline{P}_1$.
- i $\rceil$ *op*⟦C⟧ $\subseteq$ Γ x i is equivalent to the conjunction {<s, L_k> : s $\in$ $\underline{P}_k$} $\rceil$ *op*⟦C⟧ $\subseteq$ Γ x i for k = 1, …, 7 and $\underline{P}_8$ $\rceil$ *op*⟦C⟧ $\subseteq$ Γ x i so that the proof can be done by cases corresponding to each possible transition from configurations <s, L_k> for any s satisfying predicate P_k attached to point L_k. Using in each case the definition of *op*⟦C⟧, we obtain Floyd's simpler verification conditions:

  - For the assignments X_k := E_k, k = 1, 4, 5, 6, 7 we have <<s, L_k>, <s', $L_{k'}$>> $\in$ *op*⟦C⟧ if and only if k' = succ(k) where succ = [1 ← 2, 4 ← 5, 5 ← 2, 6 ← 7, 7 ← 2]. Therefore the corresponding verification conditions are of the following form given by Floyd: $\forall$ s $\in$ $\underline{P}_k$. s[X_k ← $\underline{E}_k$(s)] $\in$ $\underline{P}_{succ(k)}$.

  For example when k = 4, we have to prove that [s(Y) > 0 $\wedge$ s(Z) * (s(X) ** s(Y)) = s(x) ** s(y)] ⇒ [s[Y ← s(Y) - 1] $\in$ {s' : s'(Y) ≥ 0 $\wedge$ s'(Z) * (s'(X) ** (s'(Y) + 1)) = s'(x) ** s'(y)}] or equivalently [s(Y) > 0 $\wedge$ s(Z) * (s(X) ** s(Y)) = s(x) ** s(y)] ⇒ [(s(Y) - 1) ≥ 0 $\wedge$ s(Z) * (s(X) ** ((s(Y) - 1) + 1)) = s(x) ** s(y)] which is obvious.

  - For the test k = 3, we have <<s, L_3>, <s', $L_{k'}$>> $\in$ *op*⟦C⟧ if and only if s' = s and if s $\in$ $\underline{\text{Y <> 0}}$ then k' = 4 else k' = 6, so that we have to prove that ($\underline{P}_3$ ∩ $\underline{\text{Y <> 0}}$) $\subseteq$ $\underline{P}_4$ $\wedge$ ($\underline{P}_3$ ∩ $\underline{\text{Y = 0}}$) $\subseteq$ $\underline{P}_6$.
  - The same way, for the while loop k = 2, we have to prove that ($\underline{P}_2$ ∩ $\underline{\text{odd(Y)}}$) $\subseteq$ $\underline{P}_3$ $\wedge$ ($\underline{P}_2$ ∩ $\underline{\text{even(Y)}}$) $\subseteq$ $\underline{P}_8$.
- ((i ∩ *S*) $\subseteq$ q) = ($\underline{P}_8$ $\subseteq$ q) is equivalent to $\underline{P}_8$ = q. ■

5.2.2 Representing a global invariant on configurations by local invariants on states attached to program points

However, instead of using a single global invariant i on configurations as in (29), Floyd and Naur proposed to use local invariants on states attached to program points (originally, arcs of flowcharts). Such program points L $\in$ $\mathbb{Lab}$⟦C⟧ for commands C $\in$ $\mathbb{Comp}$ can be understood as labels specifying where control can reside at before, when or after executing a step within C. According to the operational semantics (13), $\mathbb{Lab}$⟦C⟧ can be chosen as the set of control states C' of configurations <s, C'> $\in$ *S* x $\mathbb{Comp}$ encountered during execution of command C, together with a final label, arbitrarily denoted "√", corresponding to configurations γ $\in$ *S* for which execution of C is terminated:

DEFINITION *Labels designating program control points* (32)

$\mathbb{Lab}[\![C]\!] = \mathbb{At}[\![C]\!] \cup \mathbb{In}[\![C]\!] \cup \mathbb{After}[\![C]\!]$ (.1)

$\mathbb{At}[\![C]\!] = \{C\}$ (.2)

$\mathbb{In}[\![C]\!] = \emptyset$ if C is skip, X := E or X := ? (.3)

$\mathbb{In}[\![(C_1; C_2)]\!] = \{(C'_1; C_2) : C'_1 \in \mathbb{In}[\![C_1]\!]\} \cup \mathbb{At}[\![C_2]\!] \cup \mathbb{In}[\![C_2]\!]$ (.4)

$\mathbb{In}[\![(B \to C_1 \lozenge C_2)]\!] = \mathbb{At}[\![C_1]\!] \cup \mathbb{In}[\![C_1]\!] \cup \mathbb{At}[\![C_2]\!] \cup \mathbb{In}[\![C_2]\!]$ (.5)

$\mathbb{In}[\![(B * C_1)]\!] = \{(C'; (B * C_1)) : C' \in \mathbb{At}[\![C_1]\!] \cup \mathbb{In}[\![C_1]\!]\}$ (.6)

$\mathbb{After}[\![C]\!] = \{ \surd \}$ (.7)

Example *Labels of program (4)* (33)

The labels of program (4) have been defined at (15). We have $\mathbb{At}[\![C^{10}]\!] = \{C^{10}\}$ and $\mathbb{In}[\![C^{10}]\!] = \{C^{100}, C^{1001}, C^{101}, C^{1011}\}$ whereas $\mathbb{At}[\![C^{\varepsilon}]\!] = \{L_1\}$, $\mathbb{In}[\![C^{\varepsilon}]\!] = \{L_2, L_3, L_4, L_5, L_6, L_7\}$ and $\mathbb{After}[\![C^{\varepsilon}]\!] = \{L_8\}$. ■

The local invariants are assertions on states attached to program points. More formally they can be defined as a function 'inv', which maps labels of C to assertions:

DEFINITION *Local invariants* (34)

inv : $\mathbb{Lab}[\![C]\!] \to \mathbb{Ass}$

Example *Local invariants for program (4)* (35)

Local invariants for program (4) have been given at (25): $inv(L_k) = \underline{P}_k$, $k = 1, \ldots, 8$. ■

The local invariants inv(L), $L \in \mathbb{Lab}[\![C]\!]$ can be understood as describing the global invariant $\gamma(inv) \in \mathcal{P}(\Gamma)$, which is the set of configurations such that when control is at L the memory state belongs to inv(L). Reciprocally, a global invariant $i \in \mathcal{P}(\Gamma)$ can be decomposed into local invariants $\alpha(i)(L)$, $L \in \mathbb{Lab}[\![C]\!]$, defined by the fact that when control is at L the only possible memory states s are those for which the configuration <s, L> belongs to i (or s belongs to i if $L = \surd$):

DEFINITION COUSOT & COUSOT [1982] *Connection between local and global invariants*

Concretization function (36)

$\gamma : (\mathbb{Lab}[\![C]\!] \to \mathbb{Ass}) \to \mathcal{P}(\Gamma)$ (.1)

$\gamma(inv) = \{\langle s, L\rangle : s \in inv(L) \wedge L \in \mathbb{Lab}[\![C]\!] - \{ \surd \}\} \cup inv(\surd)$ (.2)

Abstraction function: (37)

$\alpha : \mathcal{P}(\Gamma) \to (\mathbb{Lab}[\![C]\!] \to \mathbb{Ass})$ (.1)

$\alpha(i)(L) = \{s : \langle s, L\rangle \in i\}$ if $L \in \mathbb{At}[\![C]\!] \cup \mathbb{In}[\![C]\!]$ (.2)

$\alpha(i)(L) = i \cap S$ if $L \in \mathbb{After}[\![C]\!]$ (.3)

Since α is a bijection, the inverse of which is γ, the discovery of a global invariant $i \in \mathcal{P}(\Gamma)$ satisfying verification condition $gvc[\![C]\!][p, q](i)$ is equivalent to the discovery of local invariants $inv(L)$, $L \in \mathbb{Lab}[\![C]\!]$ satisfying verification condition $gvc[\![C]\!][p, q](\gamma(inv))$. This leads to the construction of the local verification conditions by calculus (COUSOT & COUSOT [1982]). This equivalence is of theoretical interest only since, from a practical point of view, each local invariant is simpler than the global one and the task of checking $gvc[\![C]\!][p, q](\gamma(inv))$ can be decomposed into the verification of more numerous but simpler conditions, one for each local invariant.

5.2.3 Construction of the verification conditions for local invariants

To formally derive the local verification conditions from induction principle (29), we first express the operational semantics (13) in an equivalent form using program steps. From a syntactic point of view, the next elementary step $\mathbb{Step}[\![C]\!][L]$ which will be executed when control is at point $L \in \mathbb{At}[\![C]\!] \cup \mathbb{In}[\![C]\!]$ of command $C \in \mathbb{Comp}$ is an atomic command or a test defined by cases as follows (where $n \geq 0$):

DEFINITION *Elementary steps within a command* (38)

$\mathbb{Step}[\![C]\!][(...((C'; C_1); C_2) ...; C_n)] = C'$ if C' is skip, X := E or X := ? (.1)

$\mathbb{Step}[\![C]\!][(...(((B \rightarrow C' \lozenge C''); C_1); C_2)...; C_n)] = B$ (.2)

$\mathbb{Step}[\![C]\!][(...(((B * C'); C_1); C_2)...; C_n)] = B$ (.3)

Example *Elementary steps of program (4)* (39)

For program C defined by (4) with labels (15), we have $\mathbb{Step}[\![C]\!]$ = [L_1 ← `Z := 1`, L_2 ← `Y <> 0`, L_3 ← `odd(Y)`, L_4 ← `Y := Y - 1`, L_5 ← `Z := Z * X`, L_6 ← `Y := T div 2`, L_7 ← `X := X * X`]. ■

Again from a syntactic point of view, the next label $\mathbb{Succ}[\![C]\!][L]$ which will be reached after execution of an elementary step when control is at point $L \in \mathbb{At}[\![C]\!] \cup \mathbb{In}[\![C]\!]$ of command $C \in \mathbb{Comp}$ can be defined by cases as follows (where $n \geq 0$ and $(...(C_1; C_2)...; C_n)$ is the final label $\surd$ for $n = 0$):

DEFINITION *Successors of a program control point* (40)

$\mathbb{Succ}[\![C]\!][(...((C'; C_1); C_2)...; C_n)] =$ (.1)

$(...(C_1; C_2)...; C_n)$ if C' is skip, X := E or X := ?

$\mathbb{Succ}[\![C]\!][(...(((B \rightarrow C' \lozenge C''); C_1); C_2)...; C_n)] =$ (.2)

$[tt \leftarrow (...((C'; C_1); C_2)...; C_n), ff \leftarrow (...((C''; C_1); C_2)...; C_n)]$

$\mathbb{Succ}[\![C]\!][(...(((B * C); C_1); C_2)...; C_n)] =$ (.3)

$[tt \leftarrow (...(((C; (B * C)); C_1); C_2)...; C_n), ff \leftarrow (...(C_1; C_2)...; C_n)]$

Example *Successors of control points of program (4)* (41)

For program C defined by (4) with labels (15), we have $\mathbb{Succ}[\![C]\!] = [L_1 \leftarrow L_2, L_2 \leftarrow [tt \leftarrow L_3,\ ff \leftarrow L_8],\ L_3 \leftarrow [tt \leftarrow L_4,\ ff \leftarrow L_6],\ L_4 \leftarrow L_5,\ L_5 \leftarrow L_2, L_6 \leftarrow L_7,\ L_7 \leftarrow L_2]$. ■

Now from a semantical point of view, execution of an elementary step $\mathbb{Step}[\![C]\!][L]$ in memory state s can lead to any successor state $s' \in \mathbb{NextS}[\![C]\!]\langle s, L\rangle$ as follows:

DEFINITION *Successor states* (42)

$\mathbb{NextS}[\![C]\!]\langle s, L\rangle = \{s\}$	if $\mathbb{Step}[\![C]\!][L]$ is skip	(.1)
$\mathbb{NextS}[\![C]\!]\langle s, L\rangle = \{s[X \leftarrow \underline{E}(s)]\}$	if $\mathbb{Step}[\![C]\!][L]$ is X := E	(.2)
$\mathbb{NextS}[\![C]\!]\langle s, L\rangle = \{s[X \leftarrow d] : d \in D\}$	if $\mathbb{Step}[\![C]\!][L]$ is X := ?	(.3)
$\mathbb{NextS}[\![C]\!]\langle s, L\rangle = \{s\}$	if $\mathbb{Step}[\![C]\!][L]$ is B	(.4)

Again from a semantical point of view, the next label $\mathbb{NextL}[\![C]\!]\langle s, L\rangle$ which can be reached after execution of an elementary step in configuration $\langle s, L\rangle$ of command C $\in \mathbb{Comp}$ can be defined by cases as follows:

DEFINITION *Successor control point* (43)

$\mathbb{NextL}[\![C]\!]\langle s, L\rangle = \{\mathbb{Succ}[\![C]\!][L]\}$	if $\mathbb{Step}[\![C]\!][L]$ is skip, X := E or X := ?	(.1)
$\mathbb{NextL}[\![C]\!]\langle s, L\rangle = \{\mathbb{Succ}[\![C]\!][L](s \in \underline{B})\}$	if $\mathbb{Step}[\![C]\!][L]$ is B	(.2)

The operational semantics (13) can now be given an equivalent stepwise presentation:

LEMMA *Stepwise presentation of the operational semantics* (44)

$$op[\![C]\!] = \{\langle\langle s, L\rangle, \text{final } \langle s', L'\rangle\rangle : s \in S \ \wedge L \in \mathbb{At}[\![C]\!] \cup \mathbb{In}[\![C]\!] \ \wedge \ s' \in \mathbb{NextS}[\![C]\!]\langle s, L\rangle \wedge L' \in \mathbb{NextL}[\![C]\!]\langle s, L\rangle\ \}$$

where final $\langle s', \surd\rangle = s'$ and otherwise final $\gamma = \gamma$.

We have seen that a partial correctness proof of $\underline{\{}\ p\ \underline{\}C\{}\ q\ \underline{\}}$ by Floyd-Naur's method consists in discovering local invariants $inv \in \mathbb{Lab}[\![C]\!] \rightarrow \mathbb{Ass}$ satisfying $gvc[\![C]\!][p, q](\gamma(inv))$. This global verification condition is equivalent to a conjunction of simpler local verification conditions as follows:

THEOREM NAUR [1966], FLOYD [1967], MANNA [1969] [1971] *Floyd-Naur partial correctness proof method with stepwise verification conditions* (45)

A partial correctness proof of $\underline{\{}$ p $\underline{\}C\{}$ q $\underline{\}}$ by Floyd-Naur's method consists in discovering local invariants inv $\in \mathbb{L}\mathrm{ab}[\![C]\!] \to \mathbb{A}\mathrm{ss}$, which must be proved to satisfy the following local verification conditions :

. $p \subseteq inv(L)$ if $L \in \mathbb{A}\mathrm{t}[\![C]\!]$ (.1)

. $inv(L) \subseteq inv(\mathbb{S}\mathrm{ucc}[\![C]\!][L])$ if $L \in \mathbb{A}\mathrm{t}[\![C]\!] \cup \mathbb{I}\mathrm{n}[\![C]\!] \wedge \mathbb{S}\mathrm{tep}[\![C]\!][L]$ is skip (.2)

. $inv(L) \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in inv(\mathbb{S}\mathrm{ucc}[\![C]\!][L])\}$ (.3)
if $L \in \mathbb{A}\mathrm{t}[\![C]\!] \cup \mathbb{I}\mathrm{n}[\![C]\!] \wedge \mathbb{S}\mathrm{tep}[\![C]\!][L]$ is X := E

. $\{s[X \leftarrow d] : s \in inv(L) \wedge d \in D\} \subseteq inv(\mathbb{S}\mathrm{ucc}[\![C]\!][L])$ (.4)
if $L \in \mathbb{A}\mathrm{t}[\![C]\!] \cup \mathbb{I}\mathrm{n}[\![C]\!] \wedge \mathbb{S}\mathrm{tep}[\![C]\!][L]$ is X := ?

. $(inv(L) \cap \underline{B}) \subseteq inv(\mathbb{S}\mathrm{ucc}[\![C]\!][L](tt))$ if $L \in \mathbb{A}\mathrm{t}[\![C]\!] \cup \mathbb{I}\mathrm{n}[\![C]\!] \wedge \mathbb{S}\mathrm{tep}[\![C]\!][L]$ is B (.5)

. $(inv(L) \cap \neg\underline{B}) \subseteq inv(\mathbb{S}\mathrm{ucc}[\![C]\!][L](ff))$ if $L \in \mathbb{A}\mathrm{t}[\![C]\!] \cup \mathbb{I}\mathrm{n}[\![C]\!] \wedge \mathbb{S}\mathrm{tep}[\![C]\!][L]$ is B (.6)

. $inv(L) \subseteq q$ if $L \in \mathbb{A}\mathrm{fter}[\![C]\!]$ (.7)

The verification condition (45.3) for assignment is backward. This name arises out of the fact that the postcondition inv($\mathbb{S}\mathrm{ucc}[\![C]\!][L]$ is back-transformed into the assertion $\{s \in S : s[X \leftarrow \underline{E}(s)] \in inv(\mathbb{S}\mathrm{ucc}[\![C]\!][L])\}$ written in terms of the states before assignment. Verification condition (45.4) for random assignment is forward. Verification condition (45.3) can also be given an equivalent forward form (KING [1969]) :

. $\{s[X \leftarrow \underline{E}(s)] : s \in inv(L)\} \subseteq inv(\mathbb{S}\mathrm{ucc}[\![C]\!][L])$
if $L \in \mathbb{A}\mathrm{t}[\![C]\!] \cup \mathbb{I}\mathrm{n}[\![C]\!] \wedge \mathbb{S}\mathrm{tep}[\![C]\!][L]$ is X:=E (.8)

Proof

By (29), we have to show that γ(inv(L)) satisfies gvc$[\![C]\!]$[p, q](γ(inv(L))). We proceed by simplification of gvc$[\![C]\!]$[p, q](γ(inv(L))) which constructively leads to local verification conditions (45):

- First, $(\forall s \in p.\ \langle s, C\rangle \in \gamma(inv(L))) \Leftrightarrow (\forall s \in p.\ C \in \mathbb{A}\mathrm{t}[\![C]\!] \cup \mathbb{I}\mathrm{n}[\![C]\!] \wedge s \in inv(C))$ [by (36.2)] $\Leftrightarrow (p \subseteq inv(C))$ [by (32.2)] $\Leftrightarrow (\forall L \in \mathbb{A}\mathrm{t}[\![C]\!].\ p \subseteq inv(L))$ [by (32.2)].
- Then, according to (36) and (44), the condition $\gamma(inv) \rceil \mathit{op}[\![C]\!] \subseteq \Gamma \times \gamma(inv)$ can be decomposed into a conjunction of simpler verification conditions, one for each program step:

$\gamma(inv) \rceil \mathit{op}[\![C]\!] \subseteq \Gamma \times \gamma(inv)$

$\Leftrightarrow \{\langle s, L\rangle : s \in inv(L) \wedge L \in \mathbb{L}\mathrm{ab}[\![C]\!] - \{\surd\}\} \rceil \mathit{op}[\![C]\!] \subseteq \Gamma \times [\{\langle s, L\rangle : s \in inv(L) \wedge L \in \mathbb{L}\mathrm{ab}[\![C]\!] - \{\surd\}\} \cup inv(\surd)]$

$\Leftrightarrow \{\langle\langle s, L\rangle, final \langle s', L'\rangle\rangle : s \in inv(L) \wedge L \in \mathbb{A}\mathrm{t}[\![C]\!] \cup \mathbb{I}\mathrm{n}[\![C]\!] \wedge s' \in \mathbb{N}\mathrm{extS}[\![C]\!]\langle s,L\rangle \wedge L' \in \mathbb{N}\mathrm{extL}[\![C]\!]\langle s,L\rangle\} \subseteq \Gamma \times [\{\langle s, L\rangle : s \in inv(L) \wedge L \in \mathbb{L}\mathrm{ab}[\![C]\!] - \{\surd\}\} \cup inv(\surd)]$

$\Leftrightarrow \forall L \in \mathbb{At}[\![C]\!] \cup \mathbb{In}[\![C]\!].\ \forall s \in inv(L).\ \{final <s', L'> : s' \in \mathbb{NextS}[\![C]\!]<s, L> \wedge L' \in \mathbb{NextL}[\![C]\!]<s, L>\} \subseteq [\{<s'', L''> : s'' \in inv(L'') \wedge L'' \in \mathbb{Lab}[\![C]\!] - \{\surd\}\} \cup inv(\surd)]$

We go on by cases, according to (38), (40) and (42):

- If $\mathbb{Step}[\![C]\!][L]$ is X := E (skip and X: = ? are handled the same way), then we have to check that :

$\{final <s', L'> : s' \in \{s[X \leftarrow \underline{E}(s)]\} \wedge L' \in \{\mathbb{Succ}[\![C]\!][L]\}\} \subseteq [\{<s'', L''> : s'' \in inv(L'') \wedge L'' \in \mathbb{Lab}[\![C]\!] - \{\surd\}\} \cup inv(\surd)]$

$\Leftrightarrow final <s[X \leftarrow \underline{E}(s)], \mathbb{Succ}[\![C]\!][L]> \in [\{<s'', L''> : s'' \in inv(L'') \wedge L'' \in \mathbb{Lab}[\![C]\!] - \{\surd\}\} \cup inv(\surd)]$

$\Leftrightarrow s[X \leftarrow \underline{E}(s)] \in inv(\mathbb{Succ}[\![C]\!][L])$

(and $\forall s \in inv(L).\ s[X \leftarrow \underline{E}(s)] \in inv(\mathbb{Succ}[\![C]\!][L])$ is obviously equivalent to $inv(L) \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in inv(\mathbb{Succ}[\![C]\!][L])\}$ and to $\{s[X \leftarrow \underline{E}(s)] : s \in inv(L)\} \subseteq inv(\mathbb{Succ}[\![C]\!][L])$)),

- If $\mathbb{Step}[\![C]\!][L]$ is B then we have to check that :

$\{final <s', L'> : s' \in \{s\} \wedge L' \in \{\mathbb{Succ}[\![C]\!][L](s \in \underline{B})\}\} \subseteq [\{<s'', L''> : s'' \in inv(L'') \wedge L'' \in \mathbb{Lab}[\![C]\!] - \{\surd\}\} \cup inv(\surd)]$

$\Leftrightarrow final <s, \mathbb{Succ}[\![C]\!][L](s \in \underline{B})> \in [\{<s'', L''> : s'' \in inv(L'') \wedge L'' \in \mathbb{Lab}[\![C]\!] - \{\surd\}\} \cup inv(\surd)]$

$\Leftrightarrow s \in inv(\mathbb{Succ}[\![C]\!][L](s \in \underline{B}))$.

• Finally, $(\gamma(inv) \cap S \subseteq q) \Leftrightarrow (inv(\surd) \subseteq q)$ [by (36.2)] $\Leftrightarrow (\forall L \in \mathbb{After}[\![C]\!].\ inv(L) \subseteq q)$ [by (32.7)]. ■

5.2.4 Semantical soundness and completeness of the stepwise Floyd-Naur partial correctness proof method

A proof method is *sound* if it cannot lead to mistaken conclusions. It is *complete* if it is always applicable to prove indubitable facts.

THEOREM DE BAKKER & MEERTENS [1975] *Soundness and semantical completeness of the stepwise Floyd-Naur method* (46)

The stepwise presentation of Floyd-Naur partial correctness proof method is semantically sound and complete.

Proof

The method is sound since if inv satisfies (45) then, by construction of (45), $gvc[\![C]\!][p, q](\gamma(inv))$ holds so that $\{\!|\ p\ |\!\}C\{\!|\ q\ |\!\}$ derives from (29). It is semantically complete since if $\{\!|\ p\ |\!\}C\{\!|\ q\ |\!\}$ is true then by (29) we know that $I = \{<s, C> : s \in p\} \rceil op[\![C]\!]^*$ satisfies $gvc[\![C]\!][p,q](I)$ so that by construction, (45) holds for $inv = \alpha(I)$. ■

We insists upon *semantical* soundness and completeness as in DE BAKKER & MEERTENS [1975] or MANNA & PNUELI [1970] since (46) is relative to a given semantics of programs (13) and to a representation of invariants by sets as opposed to the existence of a formal calculus in a given language to prove partial correctness of programs (GERGELY & SZÖTS [1978], SAIN [1985]).

5.3 The compositional Floyd-Naur partial correctness proof method

HOARE [1969] introduced the idea (often called *compositionality*) that the specification of a command should be verifiable in terms of the specifications of its components. This means that partial correctness should be proved by induction on the syntax of programs using their relational semantics (19) instead of an induction on the number of transitions using their operational semantics (13). Following OWICKI [1975], we give a syntax-directed presentation of Floyd-Naur's method without appeal to a formal logic. To do this we associate preconditions and postconditions with commands and introduce structural verification conditions so that a proof of a composite command is composed of the proofs of its constituent parts. Although this later turns out to be redundant, we prove the semantical soundness and completeness of the method since the underlying reasoning constitutes a simple introduction to relative completeness proofs of Hoare logic.

5.3.1 Preconditions and postconditions of commands

A partial correctness proof of $\{ p \} C \{ q \}$ by Floyd-Naur's method consists in discovering a precondition pre(C') and a postcondition post(C') specifying the partial correctness {pre(C')}C'{post(C')} of each component C' of command C. This includes an invariant linv(C') for each loop C' within C. Formally “pre”, “post” and “linv” can be understood as functions which maps components of C to assertions :

DEFINITION *Preconditions, postconditions and loop invariants attached to commands* (47)

pre, post : $\mathbb{Comp}[\mathrm{C}] \rightarrow \mathbb{Ass}$ (.1)

linv : $\mathbb{Loops}[\mathrm{C}] \rightarrow \mathbb{Ass}$ (.2)

Example *Preconditions, postconditions and loop invariants for program* (4) (48)

For program C defined by (4) with components defined by (5), we can choose :

$\text{pre}(C^{\varepsilon}) = \text{pre}(C^{0}) = \underline{P}_1$

$\text{post}(C^{0}) = \text{pre}(C^{1}) = \text{linv}(C^{1}) = \text{post}(C^{1001}) = \text{post}(C^{100}) = \text{post}(C^{1011}) = \text{post}(C^{101}) = \text{post}(C^{10}) = \underline{P}_2$

$\text{pre}(C^{10}) = \underline{P}_3$

$\text{pre}(C^{100}) = \text{pre}(C^{1000}) = \underline{P}_4$

$\text{post}(C^{1000}) = \text{pre}(C^{1001}) = \underline{P}_5$

$\text{pre}(C^{101}) = \text{pre}(C^{1010}) = \underline{P}_6$

$\text{post}(C^{1010}) = \text{pre}(C^{1011}) = \underline{P}_7$

$\text{post}(C^{1}) = \text{post}(C^{\varepsilon}) = \underline{P}_8$

■

5.3.2 Compositional verification conditions

Then these assertions should be proved to satisfy the following verification conditions which are defined compositionally, that is by recursion on the syntax of commands :

DEFINITION OWICKI [1975] *Compositional Floyd-Naur partial correctness proof method* (49)

A partial correctness proof of $\underline{\{}\ p\ \underline{\}C\{}\ q\ \underline{\}}$ by Floyd-Naur's method consists in discovering preconditions, postconditions and loop invariants (47) which must be proved to satisfy the following compositional verification conditions :

. $p \subseteq \text{pre}(C) \wedge \text{post}(C) \subseteq q$ (.1)

For each component $C' \in \mathbb{C}\text{omp}[\![C]\!]$ of C :

. $\text{pre}(C') \subseteq \text{post}(C')$ if C' is skip (.2)

. $\text{pre}(C') \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in \text{post}(C')\}$ if C' is X := E (.3)

. $\{s[X \leftarrow d] : s \in \text{pre}(C') \wedge d \in D\} \subseteq \text{post}(C')$ if C' is X := ? (.4)

. $\text{pre}(C') \subseteq \text{pre}(C_1) \wedge \text{post}(C_1) \subseteq \text{pre}(C_2) \wedge \text{post}(C_2) \subseteq \text{post}(C')$ if C' is $(C_1; C_2)$ (.5)

. $(\text{pre}(C') \cap \underline{B}) \subseteq \text{pre}(C_1) \wedge (\text{pre}(C') \cap \neg\underline{B}) \subseteq \text{pre}(C_2) \wedge \text{post}(C_1) \subseteq \text{post}(C') \wedge \text{post}(C_2) \subseteq \text{post}(C')$ if C' is $(B \rightarrow C_1 \lozenge C_2)$ (.6)

. $\text{pre}(C') \subseteq \text{linv}(C') \wedge (\text{linv}(C') \cap \underline{B}) \subseteq \text{pre}(C_1) \wedge \text{post}(C_1) \subseteq \text{linv}(C') \wedge (\text{linv}(C') \cap \neg\underline{B}) \subseteq \text{post}(C')$ if C' is $(B * C_1)$ (.7)

Observe that these compositional verification conditions could also have been defined by an attribute grammar (KNUTH [1968b]) using context-free grammar (1) with attributes “pre”, “post” and “linv” so that (49) expresses the relationships between these attributes (see GERHART [1975] and REPS & ALPERN [1984]).

Example *Compositional verification conditions for program (4)* (50)

These verification conditions are given below for program C defined by (4). Some of them, corresponding to assertions attached to the same label, are obviously satisfied :

$$\mathrm{pre}(C^{\varepsilon}) \subseteq \mathrm{pre}(C^{0}) \wedge \mathrm{post}(C^{0}) \subseteq \mathrm{pre}(C^{1}) \wedge \mathrm{post}(C^{1}) \subseteq \mathrm{post}(C^{\varepsilon}) \wedge \mathrm{pre}(C^{100}) \subseteq \mathrm{pre}(C^{1000}) \wedge \mathrm{post}(C^{1000}) \subseteq \mathrm{pre}(C^{1001}) \wedge \mathrm{post}(C^{1001}) \subseteq \mathrm{post}(C^{100}) \subseteq \mathrm{post}(C^{10}) \wedge \mathrm{pre}(C^{101}) \subseteq \mathrm{pre}(C^{1010}) \wedge \mathrm{post}(C^{1010}) \subseteq \mathrm{pre}(C^{1011}) \wedge \mathrm{post}(C^{1011}) \subseteq \mathrm{post}(C^{101}) \subseteq \mathrm{post}(C^{10}) \subseteq \mathrm{linv}(C^{1})$$

Moreover, (49.7) distinguishes the precondition of a loop (e.g. $\mathrm{pre}(C^1) = (X = x \wedge Y = y \geq 0 \wedge Z = 1)$) from its invariant (e.g. $\mathrm{linv}(C^1) = (Y \geq 0 \wedge Z * (X ** Y) = x ** y)$) :

$$\mathrm{pre}(C^1) \subseteq \mathrm{linv}(C^1)$$

The remaining verification conditions correspond to elementary steps of the program. They are set theoretic interpretations of formulae (26) :

(ε)	$p \subseteq \mathrm{pre}(C^{\varepsilon})$
(i_1)	$\mathrm{pre}(C^{0}) \subseteq \{s \in S : s[Z \leftarrow 1] \in \mathrm{post}(C^{0})\}$
(i_2)	$(\mathrm{linv}(C^{1}) \cap \{s \in S : s(Y) \neq 0\}) \subseteq \mathrm{pre}(C^{10})$
(i_3)	$(\mathrm{linv}(C^{1}) \cap \neg\{s \in S : s(Y) \neq 0\}) \subseteq \mathrm{post}(C^{1})$
(i_4)	$(\mathrm{pre}(C^{10}) \cap \{s \in S : \mathit{odd}(s(Y))\}) \subseteq \mathrm{pre}(C^{100})$
(i_5)	$(\mathrm{pre}(C^{10}) \cap \neg\{s \in S : \mathit{odd}(s(Y))\}) \subseteq \mathrm{pre}(C^{101})$
(i_6)	$\mathrm{pre}(C^{1000}) \subseteq \{s \in S : s[Y \leftarrow s(Y) - 1] \in \mathrm{post}(C^{1000})\}$
(i_7)	$\mathrm{pre}(C^{1001}) \subseteq \{s \in S : s[Z \leftarrow s(Z) * s(X)] \in \mathrm{post}(C^{1001})\}$
(i_8)	$\mathrm{pre}(C^{1010}) \subseteq \{s \in S : s[Y \leftarrow s(Y)\ \mathit{div}\ 2] \in \mathrm{post}(C^{1010})\}$
(i_9)	$\mathrm{pre}(C^{1011}) \subseteq \{s \in S : s[X \leftarrow s(X) * s(X)] \in \mathrm{post}(C^{1011})\}$
(σ)	$\mathrm{post}(C^{\varepsilon}) \subseteq q$

■

5.3.3 Semantical soundness and completeness of the compositional Floyd-Naur partial correctness proof method

The compositional presentation Floyd-Naur's proof method is semantically sound and complete :

THEOREM *Soundness of the compositional Floyd-Naur proof method* (51)

If verification conditions (49) are satisfied for all components C' of C then :

$\forall C' \in \mathbb{Comp}[\![C]\!].\ \{\ \mathrm{pre}(C')\ \}C'\{\ \mathrm{post}(C')\ \}$

It follows from (49.1) that $\{\ p\ \}C\{\ q\ \}$ holds.

Proof

We prove that $\forall$ C' $\in \mathbb{C}\mathrm{omp}[\![C]\!]$. $\underline{\{}$ pre(C') $\underline{\}C'\{}$ post(C') $\underline{\}}$ or equivalently by (22) that $\forall$ C' $\in \mathbb{C}\mathrm{omp}[\![C]\!]$. (pre(C') $\rceil$ $\underline{C'}$) $\subseteq$ (S x post(C')) by structural induction on C' :

- If C' is X := E then by (49.3) we have $\big($pre(C') $\subseteq \{$s $\in S$: s[X $\leftarrow \underline{E}$(s)] $\in$ post(C')$\}\big) \Leftrightarrow \big(\{$s[X $\leftarrow \underline{E}$(s)] : s $\in$ pre(C')$\}$ $\subseteq$ post(C')$\big) \Leftrightarrow \big(($pre(C') $\rceil$ $\underline{C'}$) $\subseteq$ (S x post(C'))$\big)$ by (19.2). The cases skip and X := ? are handled the same way.
- If C' is (C_1; C_2) then (pre(C') $\rceil$ $\underline{C_1; C_2}$) = (pre(C') $\rceil$ $\underline{C_1} \circ \underline{C_2}$) [by (19.4)] = (pre(C') $\rceil$ $\underline{C_1}$) $\circ$ $\underline{C_2}$ $\subseteq$ (S x post(C_1)) $\circ$ $\underline{C_2}$ [since (pre(C_1) $\rceil$ $\underline{C_1}$) $\subseteq$ (S x post(C_1)) by induction hypothesis] = $S^2 \circ$ (post(C_1) $\rceil$ $\underline{C_2}$) $\subseteq S^2 \circ$ (pre(C_2) $\rceil$ $\underline{C_2}$) [by (49.5)] $\subseteq$ $S^2 \circ$ (S x post(C_2)) [since (pre(C_2) $\rceil$ $\underline{C_2}$) $\subseteq$ (S x post(C_2)) by induction hypothesis] = S x post(C_2) $\subseteq$ S x post(C') [by (49.5)]. The case C' = (B $\rightarrow$ $C_1 \lozenge C_2$) is handled the same way.
- If C' is (B * C_1) then (pre(C_1) $\rceil$ $\underline{C_1}$) $\subseteq$ (S x post($\underline{C_1}$)) holds by induction hypothesis which implies (linv(C') $\rceil$ ($\underline{B}$ $\rceil$ $\underline{C_1}$)) $\subseteq$ (S x linv(C')) by (49.7). Also pre(C') $\subseteq$ linv(C') and (linv(C') $\cap$ $\neg\underline{B}$) $\subseteq$ post(C') by (49.6) so that (pre(C') $\rceil$ $\underline{(B \rceil C_1)^*}$ $\lceil$ $\neg\underline{B}$) $\subseteq$ (S x post(C')) by (28) hence (pre(C') $\rceil$ $\underline{(B * C_1)}$) $\subseteq$ (S x post(C')) by (19.6). ■

THEOREM *Semantical completeness of the compositional Floyd-Naur proof method* (52)

If $\underline{\{}$ p $\underline{\}C\{}$ q $\underline{\}}$ holds then there are functions pre, post and linv verifying conditions (49) for all components C' of C.

Proof

The proof is by structural induction on C :

- If C is X := E then $\underline{\{}$ p $\underline{\}X := E\{}$ q $\underline{\}}$ $\Rightarrow$ (p $\rceil$ $\underline{X := E}$) $\subseteq$ (S x q) [by (22)] $\Rightarrow$ $\{$s[X $\leftarrow \underline{E}$(s)] : s $\in$ p$\}$ $\subseteq$ q [by (19.2)] $\Rightarrow$ (49.1) $\wedge$ (49.3) if we let pre(X := E) = p and post(X := E) = q. The cases skip and X := ? are handled the same way.
- If C is (C_1; C_2) then we let pre(C) = pre(C_1) = p, post(C) = post(C_2) = q and post(C_1) = pre(C_2) = $\{$s : $\exists$ s' $\in$ p. <s', s> $\in$ $\underline{C_1}\}$. Then (49.1) and (49.5) are satisfied. It remains to show that (49.2), …, (49.7) hold for all components C' of C. By induction hypothesis, we just have to show that $\underline{\{}$ pre(C_1) $\underline{\}C_1\{}$ post(C_1) $\underline{\}}$ and $\underline{\{}$ pre(C_2) $\underline{\}C_2\{}$ post(C_1) $\underline{\}}$.

We have $\underline{\{}$ p $\underline{\}C_1\{}$ $\{$s : $\exists$ s' $\in$ p.<s', s> $\in$ $\underline{C_1}\}$ $\underline{\}}$ because (p $\rceil$ $\underline{C_1}$) = $\{$<s, s'> : s $\in$ p $\wedge$ <s, s'> $\in$ $\underline{C_1}\}$ $\subseteq$ $\{$<s", s'> : $\exists$ s $\in$ p.<s, s'> $\in$ $\underline{C_1}\}$ = S x $\{$s : $\exists$ s' $\in$ p. <s', s> $\in$ $\underline{C_1}\}$. Moreover $\underline{\{}$ p $\underline{\}(C_1; C_2)\{}$ q $\underline{\}}$ $\Rightarrow$ (p $\rceil$ $\underline{(C_1; C_2)}$) $\subseteq$ (S x q) [by (22)] $\Rightarrow$ ((p $\rceil$ $\underline{C_1}$) $\circ$ $\underline{C_2}$) $\subseteq$ (S x q) [by (19.4)] $\Rightarrow$ ($\forall$s', s, s" $\in S$. (s' $\in$ p $\wedge$ <s', s> $\in$ $\underline{C_1}$ $\wedge$ <s, s"> $\in$ $\underline{C_2}$) $\Rightarrow$ (s" $\in$ q)) $\Rightarrow$ (($\{$s : $\exists$ s' $\in$ p. <s', s> $\in$ $\underline{C_1}\}$ $\rceil$ $\underline{C_2}$) $\subseteq$ (S x q)) $\Rightarrow$ $\underline{\{}$ pre(C_2) $\underline{\}C_2\{}$ post(C_1) $\underline{\}}$ by (22).

- The proof is similar when C is (B $\rightarrow$ $C_1 \lozenge C_2$) choosing pre(c') = p, pre(C_1) = p $\cap$ $\underline{B}$, pre(C_2) = p $\cap$ $\neg\underline{B}$ and post(C') = post(C_1) = post(C_2) = q.

• If C is $(B * C_1)$ then $\underline{\{}\ p\ \underline{\}(B * C_1)\{}\ q\ \underline{\}} \Rightarrow (p \rceil \underline{(B * C_1)}) \subseteq (\mathcal{S} \times q)$ [by (22)] $\Rightarrow$ $(p \rceil \underline{(B} \rceil \underline{C_1)}^* \lceil \neg\underline{B}) \subseteq (\mathcal{S} \times q)$ [by (19.6)] $\Rightarrow [\exists\, i \in \mathbb{Ass}.\ (p \subseteq i) \wedge (i \rceil \underline{(B} \rceil \underline{C_1)} \subseteq \mathcal{S} \times i) \wedge$ $(i \cap \neg\underline{B} \subseteq q)]$ [by (28)]. We now define $\mathrm{pre}((B * C_1)) = p$, $\mathrm{linv}((B * C_1)) = i$, $\mathrm{post}((B * C_1)) = q$, $\mathrm{pre}(C_1) = i \cap \underline{B}$ and $\mathrm{post}(C_1) = i$. It follows that (49.1) and (49.7) hold for C. It remains to show that $\underline{\{}\ \mathrm{pre}(C_1)\ \underline{\}C_1\{}\ \mathrm{post}(C_1)\ \underline{\}}$. This immediately follows from the definitions of $\mathrm{pre}(C_1)$ and $\mathrm{post}(C_1)$, $(i \cap \underline{B}) \rceil \underline{C_1} \subseteq \mathcal{S} \times i)$ and (22). ■

5.4 Equivalence of stepwise and compositional Floyd-Naur partial correctness proofs

Examples (26) and (50) show that the compositional Floyd-Naur partial correctness proof method introduces some trivially satisfied verification conditions which do not appear in the stepwise version. Apart from this difference in the presentation, the stepwise and compositional Floyd-Naur partial correctness proofs of program (4) are equivalent. This property is general in the sense that a proof using one presentation can always be derived from a proof using the other presentation. Since the assertions are the same in both presentations, (30.3) and (36) imply that preconditions and postconditions in the compositional presentation (hence later in Hoare logic) are local invariants, a fact which is often taken for granted. By (46), this also implies that the syntax-directed presentation is semantically sound and complete, a fact already proved by (51) and (52).

5.4.1 The compositional presentation of a stepwise Floyd-Naur partial correctness proof

The precondition pre(C') of a component C' of a command $C \in \mathbb{Com}$ (and loop invariant linv(C') when C' is a loop) can always be chosen as the local invariant inv(L) attached to the label $L = \mathbb{Lpre}[\![C]\!][C']$ designating where control is just before executing that component C'. The same way, the postcondition post(C') of a component C' of a command C can always be chosen as the local invariant inv(L) attached to the label $L = \mathbb{Lpost}[\![C]\!][C']$ designating where control is just after executing that component C'.

DEFINITION *Program points before and after components of a command* (53)

The label just before and after a component C' of a command $C \in \mathbb{Com}$:

$\mathbb{Lpre}[\![C]\!] \in \mathbb{Comp}[\![C]\!] \rightarrow \mathbb{Lab}[\![C]\!]$ (.1)

$\mathbb{Lpost}[\![C]\!] \in \mathbb{Comp}[\![C]\!] \rightarrow \mathbb{Lab}[\![C]\!]$ (.2)

is defined by structural induction on C :

. $\mathbb{Lpre}[\![C]\!][C] = C$ (.3)

. $\mathbb{Lpost}[\![C]\!][C] = \surd$ (.4)

For each $C' \in \mathbb{Comp}[\![C]\!] - \{C\}$ when C is $(C_1;C_2)$, $(B \rightarrow C_1 \lozenge C_2)$ or $(B * C_1)$:

• $\mathbb{Lpre}[\![(C_1;\ C_2)]\!][C'] = (\mathbb{Lpre}[\![C_1]\!][C'];\ C_2)$ if $C' \in \mathbb{Comp}[\![C_1]\!]$ (.5)

. $\mathbb{Lpre}[\![(C_1;\ C_2)]\!][C'] = \mathbb{Lpre}[\![C_2]\!][C']$ if $C' \in \mathbb{Comp}[\![C_2]\!]$ (.6)

. $\mathbb{Lpost}[\![(C_1;\ C_2)]\!][C'] = C_2$ (.7)

if $C' \in \mathbb{Comp}[\![C_1]\!] \wedge \mathbb{Lpost}[\![C_1]\!][C'] = \surd$

. $\mathbb{Lpost}[\![(C_1;\ C_2)]\!][C'] = (\mathbb{Lpost}[\![C_1]\!][C'];\ C_2)$ (.8)

if $C' \in \mathbb{Comp}[\![C_1]\!] \wedge \mathbb{Lpost}[\![C_1]\!][C'] \neq \surd$

. $\mathbb{Lpost}[\![(C_1;\ C_2)]\!][C'] = \mathbb{Lpost}[\![C_2]\!][C']$ if $C' \in \mathbb{Comp}[\![C_2]\!]$ (.9)

• $\mathbb{Lpre}[\![(B \rightarrow C_1 \lozenge C_2)]\!][C'] = \mathbb{Lpre}[\![C_1]\!][C']$ if $C' \in \mathbb{Comp}[\![C_1]\!]$ (.10)

. $\mathbb{Lpre}[\![(B \rightarrow C_1 \lozenge C_2)]\!][C'] = \mathbb{Lpre}[\![C_2]\!][C']$ if $C' \in \mathbb{Comp}[\![C_2]\!]$ (.11)

. $\mathbb{Lpost}[\![(B \rightarrow C_1 \lozenge C_2)]\!][C'] = \mathbb{Lpost}[\![C_1]\!][C']$ if $C' \in \mathbb{Comp}[\![C_1]\!]$ (.12)

. $\mathbb{Lpost}[\![(B \rightarrow C_1 \lozenge C_2)]\!][C'] = \mathbb{Lpost}[\![C_2]\!][C']$ if $C' \in \mathbb{Comp}[\![C_2]\!]$ (.13)

. $\mathbb{Lpre}[\![(B * C_1)]\!][C'] = (\mathbb{Lpre}[\![C_1]\!][C'];\ (B * C_1))$ if $C' \in \mathbb{Comp}[\![C_1]\!]$ (.14)

• $\mathbb{Lpost}[\![(B * C_1)]\!][C'] = (B * C_1)$ (.15)

if $C' \in \mathbb{Comp}[\![C_1]\!] \wedge \mathbb{Lpost}[\![C_1]\!][C'] = \surd$

. $\mathbb{Lpost}[\![(B * C_1)]\!][C'] = (\mathbb{Lpost}[\![C_1]\!][C'];\ (B * C_1))$ (.16)

if $C' \in \mathbb{Comp}[\![C_1]\!] \wedge \mathbb{Lpost}[\![C_1]\!][C'] \neq \surd$

THEOREM *Compositional presentation of a stepwise proof* (54)

If $\mathrm{inv} \in \mathbb{Lab}[\![C]\!] \rightarrow \mathbb{Ass}$ satisfies (45) then :

$\mathrm{pre} = \lambda\, C' \in \mathbb{Comp}[\![C]\!].\ \mathrm{inv}(\ \mathbb{Lpre}[\![C]\!][C']\ )$ (.1)

$\mathrm{linv} = \lambda\, C' \in \mathbb{Loops}[\![C]\!].\ \mathrm{inv}(\ \mathbb{Lpre}[\![C]\!][C']\ )$ (.2)

$\mathrm{post} = \lambda\, C' \in \mathbb{Comp}[\![C]\!].\ \mathrm{inv}(\ \mathbb{Lpost}[\![C]\!][C']\ )$ (.3)

satisfies (49).

5.4.2 The stepwise presentation of a compositional Floyd-Naur partial correctness proof

THEOREM *Stepwise presentation of a compositional proof* (55)

If pre, post $\in \mathbb{Comp}[\![C]\!] \rightarrow \mathbb{Ass}$ and linv $\in \mathbb{Loops}[\![C]\!] \rightarrow \mathbb{Ass}$ satisfy (49) then inv $\in \mathbb{Labs}[\![C]\!] \rightarrow \mathbb{Ass}$ defined as follows by structural induction on C :

. inv(C) = linv(C) if $C \in \mathbb{Loops}$ (.1)

. inv(C) = pre(C) if $C \in \mathbb{Comp} - \mathbb{Loops}$ (.2)

. inv(√) = post(C) (.3)

. inv($(C'_1; C_2)$) = inv(C'_1) if $C = (C_1; C_2) \wedge C'_1 \in \mathbb{In}[\![C_1]\!]$ (.4)

. inv($(C'; (B * C_1))$) = inv(C') if $C = (B * C_1) \wedge C' \in \mathbb{At}[\![C_1]\!] \cup \mathbb{In}[\![C_1]\!]$ (.5)

satisfies (45).

5.5 Variants of Floyd-Naur partial correctness proof method

Lemma (29) hence Floyd-Naur's method has a great number of equivalent variants, each one leading to a different partial correctness proof methodology (COUSOT & COUSOT [1982]). For example MANNA [1971] uses an invariant i and an output specification q which relate the possible configurations during execution to the initial states of variables. Otherwise stated, one uses relations between the current and initial values of variables instead of assertions upon their current values. More formally, we have :

DEFINITION MANNA [1971] *Relational partial correctness* (56)

p	: $\mathbb{Ispec} = \mathcal{P}(S)$	*Input specifications*	(.1)
q	: $\mathbb{Ospec} = \mathcal{P}(S \times S)$	*Output specifications*	(.2)
$\{\!\!\lfloor\, p \,\rfloor\!\!\}\, \underline{C} \langle\!\langle\, q \,\rangle\!\rangle$	: $\mathbb{Ispec} \times \mathbb{Com} \times \mathbb{Ospec} \rightarrow \{tt, ff\}$	*Relational partial*	(.3)
$\{\!\!\lfloor\, p \,\rfloor\!\!\}\, \underline{C} \langle\!\langle\, q \,\rangle\!\rangle$	$= (p \rceil \underline{C}) \subseteq q$	*correctness*	(.4)

Induction principle (29) can be rephrased as follows for relational partial correctness:

THEOREM MANNA [1971], COUSOT & COUSOT [1982] *Stepwise partial correctness relational proofs using invariants* (57)

$\{\!\!\lfloor\, p \,\rfloor\!\!\}\, \underline{C} \langle\!\langle\, q \,\rangle\!\rangle = [\ \exists i \in \mathcal{P}(S \times \Gamma).\{<s, <s, C>> : s \in p\} \subseteq i$ (.1)

$\wedge\ \{<s, \gamma'> : \exists \gamma.<s, \gamma> \in i \wedge <\gamma, \gamma'> \in op[\![C]\!]\} \subseteq i$ (.2)

$\wedge\ i \cap S^2 \subseteq q\ ]$ (.3)

It is also possible to prove relational partial correctness using an invariant i which relates the possible configurations during execution to the final states of the variables :

THEOREM MORRIS & WEGBREIT [1977], COUSOT & COUSOT [1982] *Subgoal induction* (58)

$\{\!\!\{ p \}\!\!\} \underline{C} \langle\!\langle q \rangle\!\rangle = [\, \exists i \in \mathcal{P}(\Gamma \times S).\ \{\langle s, s\rangle : s \in S\} \subseteq i$ (.1)

$\wedge\ \{\langle\gamma, s\rangle : \exists \gamma'.\ \langle\gamma, \gamma'\rangle \in op[\![C]\!] \wedge \langle\gamma', s\rangle \in i\} \subseteq i$ (.2)

$\wedge\ \{\langle s, s'\rangle : s \in p \wedge \langle\langle s, C\rangle, s'\rangle \in i\} \subseteq q\ ]$ (.3)

Assume that $\langle s, s'\rangle \in \underline{C}$ so that execution $\gamma_0, \ldots, \gamma_n$ of command C started in configuration $\gamma_0 = \langle s, C\rangle$ with $s \in p$, such that $\langle\gamma_{k-1}, \gamma_k\rangle \in op[\![C]\!]$ for k = 1, …, n does terminate in state $\gamma_n = s'$. Then $\langle\gamma_n, s'\rangle \in i$ by (58.1) and by downward induction on k = n, n-1, …, 0, $\langle\gamma_k, s'\rangle \in i$ follows from (58.2). In particular $\langle\gamma_0, s'\rangle \in i$ whence $\langle s, s'\rangle \in q$ by (58.3). It follows that $(p \rceil \underline{C}) \subseteq q$ whence $\{\!\!\{ p \}\!\!\} \underline{C} \langle\!\langle q \rangle\!\rangle$ holds. Semantical completeness follows from that fact that i can always be chosen as $op[\![C]\!]^* \lceil S$.

Semantical soundness and completeness imply that these partial correctness proof methods are all equivalent. This is also true in the stronger sense that the necessary invariants can be derived from one another:

THEOREM COUSOT & COUSOT [1982], DIJKSTRA [1982a] *Equivalence of stepwise induction and subgoal induction* (59)

- if i satisfies (57) then $i' = \{\langle\gamma, s\rangle : \forall s'.\ \langle s', \gamma\rangle \in i \Rightarrow \langle s', s\rangle \in q\}$ satisfies (58) (.1)
- if i' satisfies (58) then $i = \{\langle s, \gamma\rangle : \forall s''.\ \langle\gamma, s''\rangle \in i' \Rightarrow \langle s, s''\rangle \in q\}$ satisfies (57) (.2)

Proof

• If i satisfies (57) then $i \cap S^2 \subseteq q$ so that $\forall s \in S.\ \forall s' \in S.\ \langle s', s\rangle \in i \Rightarrow \langle s', s\rangle \in q$ hence $\{\langle s, s\rangle : s \in S\} \subseteq i'$ [by 33.1]. Moreover $(\langle\gamma, \gamma'\rangle \in op[\![C]\!] \wedge \langle\gamma', s\rangle \in i' \wedge \langle s', \gamma\rangle \in i) \Rightarrow (\langle s', \gamma'\rangle \in i \wedge \langle\gamma', s\rangle \in i')$ [by (57)] $\Rightarrow (\langle s', \gamma'\rangle \in i \wedge (\forall s'.\ \langle s', \gamma'\rangle \in i \Rightarrow \langle s', s\rangle \in q))$ [by (59.1)] $\Rightarrow \langle s', s\rangle \in q$ so that $\{\langle\gamma, s\rangle : \exists \gamma'.\ \langle\gamma, \gamma'\rangle \in op[\![C]\!] \wedge \langle\gamma', s\rangle \in i'\} \subseteq i'$. Finally, $(s \in p \wedge \langle\langle s, C\rangle, s'\rangle \in i') \Rightarrow (\langle s, \langle s, C\rangle\rangle \in i \wedge (\forall s''.\ \langle s'', \langle s, C\rangle\rangle \in i \Rightarrow \langle s'', s'\rangle \in q))$ [by (57) and (59.1)] $\Rightarrow \langle s, s'\rangle \in q$ hence $\{\langle s, s'\rangle : s \in p \wedge \langle\langle s, C\rangle, s'\rangle \in i'\} \subseteq q$.

• If i' satisfies (58) then $\{\langle s, s\rangle : s \in S\} \subseteq i'$ so that $i \cap S^2 = \{\langle s, s'\rangle : \forall s''.\ \langle s', s''\rangle \in i' \Rightarrow \langle s, s''\rangle \in q\} \subseteq \{\langle s, s'\rangle : \langle s', s'\rangle \in i' \Rightarrow \langle s, s'\rangle \in q\} = q$. Moreover $(\langle s, \gamma\rangle \in i \wedge \langle\gamma, \gamma'\rangle \in op[\![C]\!] \wedge \langle\gamma', s''\rangle \in i') \Rightarrow ((\forall s'.\ \langle\gamma, s'\rangle \in i' \Rightarrow \langle s, s'\rangle \in q) \wedge \langle\gamma, s''\rangle \in i')$ [by (59.2) and (58)] $\Rightarrow \langle s, s''\rangle \in q$ so that $\{\langle s, \gamma'\rangle : \exists \gamma.\ \langle s, \gamma\rangle \in i \wedge \langle\gamma, \gamma'\rangle \in op[\![C]\!]\} \subseteq i$. Finally, $\forall s \in p.\ \forall s''.\ \langle\langle s, C\rangle, s''\rangle \in i' \Rightarrow \langle s, s''\rangle \in q$ [by (58)] hence $\{\langle s, \langle s, C\rangle\rangle : s \in p\} \subseteq i$ [by (59.2)]. ■

Replacing i by ¬j in (58), we obtain an equivalent relational partial correctness proof method proceeding by reductio ad absurdum:

THEOREM COUSOT & COUSOT [1982] *Induction principle for contrapositive proofs* (60)

$$\underline{\{}\ p\ \underline{\}C\langle}\ q\ \underline{\rangle} = [\ \exists j \in \mathcal{P}(\Gamma \times S).\ p \rceil \neg q \subseteq \{\langle s, s' \rangle : \langle \langle s, C \rangle, s' \rangle \in j\} \quad (.1)$$

$$\wedge\ j \subseteq \{\langle \gamma, s \rangle : \forall \gamma'. \langle \gamma, \gamma' \rangle \in op[\![C]\!] \Rightarrow \langle \gamma', s \rangle \in j\} \quad (.2)$$

$$\wedge\ \{\langle s, s \rangle : s \in S\} \subseteq \neg j\ ] \quad (.3)$$

This consists in proving an invariance property by considering the situation where the contrary property should be true and in establishing that this situation is impossible. Assume that $\langle s, s' \rangle \in \underline{C}$ so that execution $\gamma_0, \ldots, \gamma_n$ of command C started in configuration $\gamma_0 = \langle s, C \rangle$ with $s \in p$, such that $\langle \gamma_{k-1}, \gamma_k \rangle \in op[\![C]\!]$ for k = 1, …, n does terminate in state $\gamma_n = s'$. Assume, by reductio ad absurdum, that $\langle s, s' \rangle \notin q$. Then $\langle \gamma_0, s' \rangle \in j$ by (60.1) and by induction on k = 1, …, n, $\langle \gamma_k, s' \rangle \in j$ follows from (60.2). In particular $\langle \gamma_n, s' \rangle \in j$ in contradiction with $\langle \gamma_n, s' \rangle \notin j$ following from (60.3). Semantical completeness follows from that fact that the contra-invariant j can always be chosen as $\neg(op[\![C]\!]^* \lceil S)$.

This may lead to simpler proofs when the "absurd" configuration is much simpler than the "sensible" one (see VERJUS [1987] for an example).

6. Liveness proof methods

Obviously termination is not implied in partial correctness since for example if true = I⟦true⟧ = S then true * skip = ∅ so that {p}true * skip{q} is true for all p, q ∈ $\mathbb{A}$ss. FLOYD [1967a] originally introduced total correctness as the conjunction of partial correctness and termination. Hoare logic has also been extended to cope with termination and more generally with *liveness* properties of programs (ALPERN & SCHNEIDER [1985]).

We first introduce execution traces generated by the operational semantics (13) so as to define total correctness and prove that it is the conjunction of partial correctness, deadlock freeness and termination. After giving a short mathematical recall on well founded relations, well orderings and ordinals, we introduce FLOYD [1967a]'s well-founded set method to prove termination of programs. We next consider an extension of this method to prove liveness properties P ~~C~~→ Q stipulating that starting from a configuration of P, program C does eventually reaches a configuration of Q (LAMPORT [1977]). Finally we study BURSTALL [1974] intermittent assertion method for proving total correctness (MANNA & WALDINGER [1978], GRIES [1979]) and generalize it to arbitrary liveness properties. After proper generalization, Burstall's method includes Floyd's method (COUSOT & COUSOT [1987]) and is more flexible since it allows the combination of inductions on various underlying structures of the program (syntax, computation, data, etc.).

6.1 Execution traces

We use execution traces to record the successive configurations that can be encountered during a terminating or non-terminating execution of a program. Since programs are nondeterministic, they can have many different possible executions so that we have to use sets of finite or infinite traces. The theory of traces is surveyed by MAZURKIEWICZ [1989].

Let be given a set $\mathbb{C}$om of programs, a set S of states so that the set of configurations is $\Gamma = (S \times \mathbb{C}\text{om}) \cup S$ and an operational semantics $op \in \mathbb{C}\text{om} \to \mathcal{P}(\Gamma \times \Gamma)$.

DEFINITION *Execution traces* (61)

The set of *finite complete execution traces of length* $n \in \mathbb{N}^+$ for command $C \in \mathbb{C}\text{om}$ starting in configuration $\gamma \in \Gamma$ is :

$\sum^n[\![C]\!]\gamma = \quad \{\sigma \in seq^n\, \Gamma : \sigma_0 = \gamma \wedge \forall\, i \in \{1, \ldots, n - 1\}. <\sigma_{i-1}, \sigma_i> \in op[\![C]\!]$

$\wedge\ \forall \gamma \in \Gamma.\ \langle\sigma_{n-1}, \gamma\rangle \notin op[\![C]\!]\}$ (.1)

the set of *infinite traces* of execution of command $C \in \mathbb{Com}$ starting in configuration $\gamma \in \Gamma$ is :

$\sum^{\omega}[\![C]\!]\gamma \quad = \{\sigma \in seq^{\omega}\ \Gamma : \sigma_0 = \gamma \wedge \forall i \in \mathbb{N}.\ \langle\sigma_i, \sigma_{i+1}\rangle \in op[\![C]\!]\}$ (.2)

so that the set of *finite traces* of execution of command $C \in \mathbb{Com}$ starting in configuration $\gamma \in \Gamma$ is :

$\sum^{*}[\![C]\!]\gamma \quad = \cup\{\sum^{n}[\![C]\!]\gamma : n \in \mathbb{N}\}$ (.3)

and the set of *traces* of execution of command $C \in \mathbb{Com}$ starting in configuration $\gamma \in \Gamma$ is :

$\sum[\![C]\!]\gamma \quad = \sum^{*}[\![C]\!]\gamma \cup \sum^{\omega}[\![C]\!]\gamma$ (.4)

Given $p \in \mathbb{Ass}$, we can define the traces of command C starting in a state of $p \in \mathbb{Ass}$ as :

$\sum^{n}[\![C]\!](p) \quad = \cup\{\sum^{n}[\![C]\!]\langle s, C\rangle : s \in p\}$ (.5)

$\sum^{*}[\![C]\!](p) \quad = \cup\{\sum^{*}[\![C]\!]\langle s, C\rangle : s \in p\}$ (.6)

$\sum^{\omega}[\![C]\!](p) \quad = \cup\{\sum^{\omega}[\![C]\!]\langle s, C\rangle : s \in p\}$ (.7)

$\sum[\![C]\!](p) \quad = \cup\{\sum[\![C]\!]\langle s, C\rangle : s \in p\}$ (.8)

These sets of traces can also be given an equational definition, see DE BRUIN [1984].

6.2 Total correctness

Let us recall (22) that a command C is said to be *partially correct* with respect to a specification <p, q> (written $\underline{\{p\}C\{q\}}$) if any terminating execution of C starting from an initial state s satisfying p must end in some final state s' satisfying q. C is said to be *totally correct* with respect to <p, q> (written $\underline{[p]C[q]}$) if any execution of C starting from an initial state s satisfying p does terminate properly in a final state s' satisfying q. Partial and total correctness can be defined in terms of execution traces as follows :

DEFINITIONS FLOYD [1967a] *Partial and total correctness*

$\underline{[p]C[q]}\ :\ \mathbb{Ass} \times \mathbb{Com} \times \mathbb{Ass} \to \{tt, ff\}$ *Total correctness* (62)

$\underline{[p]C[q]}\ = \forall \sigma \in \sum[\![C]\!](p).\ \exists n \in \mathbb{N}^{+}.\ \sigma \in \sum^{n}[\![C]\!](p) \wedge \sigma_{n-1} \in q$

$\underline{\{p\}C\{q\}}\ :\ \mathbb{Ass} \times \mathbb{Com} \times \mathbb{Ass} \to \{tt, ff\}$ *Partial correctness* (63)

$\underline{\{p\}C\{q\}}\ = \forall \sigma \in \sum[\![C]\!](p).\ \forall i \in dom\ \sigma.\ (\sigma_i \in S) \Rightarrow (\sigma_i \in q)$

Total correctness as defined by (62) does not necessarily imply partial correctness (63) because definition (62) does not imply that all states $\sigma_i \in S$ belong to q. However this follows from (13) because final states $s \in S$ have no possible successor, an hypothesis that we subsequently make upon the operational semantics:

HYPOTHESIS *Final states are blocking states* (64)

$\forall\, C \in \mathbb{Com}.\ \forall\, s \in S.\ \forall\, \gamma \in \Gamma.\ \langle s, \gamma\rangle \notin op[\![C]\!]$

Observe that (64) ⇒ [(62) ⇒ (63)]. A command C is said to *terminate* for initial states s ∈ p if and only if no execution trace starting from configuration <s, C> can be infinite:

DEFINITION *Termination* (65)

$\tau\underline{[p]C} \quad : \quad \mathbb{Ass} \times \mathbb{Com} \to \{tt, ff\}$

$\tau\underline{[p]C} \quad = \quad \forall\, \sigma \in \sum[\![C]\!](p).\ \exists\, n \in \mathbb{N}^+.\ \sigma \in \sum^n[\![C]\!](p)$

Termination is *proper* or *clean* for final states $\sigma_{n-1} \in S$. Execution may also end with other blocking states $\sigma_{n-1} \in \Gamma - S$. For example, a sequential program may be blocked by a run-time error such as division by zero or a parallel program may be permanently blocked because all processes are delayed at synchronization commands. Execution of a command C starting with initial states s ∈ p can be *blocked* if and only if it can reach some state some σ_{n-1} which is not final and has no possible successor :

DEFINITION *Blocked execution* (66)

$\beta\underline{\{p\}C} \quad : \quad \mathbb{Ass} \times \mathbb{Com} \to \{tt, ff\}$

$\beta\underline{\{p\}C} \quad = \quad \exists\, n \in \mathbb{N}^+.\ \exists\, \sigma \in \sum^n[\![C]\!](p).\ \sigma_{n-1} \notin S$

When no execution of a command C starting with initial states s ∈ p can end in a blocking configuration, we say that these executions are *deadlock free* :

DEFINITION *Deadlock freedom* (67)

$\neg\, \beta\underline{\{p\}C} = \forall\, n \in \mathbb{N}^+.\ \forall\, \sigma \in \sum^n[\![C]\!](p).\ \sigma_{n-1} \in S$

Under hypothesis (64), total correctness is the conjunction of partial correctness, deadlock freedom and termination :

THEOREM *Characterization of total correctness* (68)

(64) implies $\underline{[p]C[q]} = \underline{\{p\}C\{q\}} \wedge \neg\, \beta\underline{\{p\}C} \wedge \tau\underline{[p]C}$

According to (13), final states s ∈ *S* are the only possible states with no possible successor, an hypothesis that is sometimes made made upon the operational semantics :

HYPOTHESIS *Final states are the only blocking states* (69)

$\forall\, C \in \mathbb{Com}.\ \forall\, \gamma \in \Gamma.\ (\forall\, \gamma' \in \Gamma.\ \langle \gamma, \gamma'\rangle \notin op[\![C]\!]) \Rightarrow (\gamma \in S)$

Under hypotheses (64) and (69), total correctness can be expressed as the conjunction of partial correctness and termination :

THEOREM FLOYD [1967a] *Characterization of total correctness* (70)

(64) and (69) imply $\underline{[p]C[q]} = \underline{\{p\}C\{q\}} \wedge \underline{[p]C[\mathrm{true}]}$

Proof

(69) implies that $\tau\underline{[p]C} = \underline{[p]C[\mathrm{true}]} = \forall\, \sigma \in \sum[\![C]\!](p).\ \exists\, i \in \mathit{dom}\, \sigma.\ \sigma_i \in S$ where $\underline{\mathrm{true}} = I[\![\mathrm{true}]\!] = S$ and that $\beta\underline{\{p\}C} = \mathrm{ff}$ since no execution can be blocked. By (64) and (69), $\underline{[p]C[q]} = \forall\, \sigma \in \sum[\![C]\!](p).\ \exists\, i \in \mathit{dom}\, \sigma.\ \sigma_i \in q = \underline{\{p\}C\{q\}} \wedge \underline{[p]C[\mathrm{true}]}$. ■

6.3 Well founded relations, well orderings and ordinals

A relation $\prec$ on a class W is *well-founded* if and only if every subclass of W has a minimal element that is $\mathit{wf}(W, \prec) = [\forall\, E \subseteq W.\ (E \neq \emptyset \Rightarrow \exists\, y \in E.\ (\neg\, \exists\, z \in E.\ z \prec y))]$ is true. If $\mathit{wf}(W, \prec)$ then obviously there is no infinite decreasing sequence $x_0 \succ x_1 \succ \ldots$ where $\succ$ is the inverse of $\prec$.

A relation $\prec$ on a class W is a *strict partial ordering* if and only if it is anti-reflexive and transitive that is $\mathit{spo}(W, \prec) = [(\forall\, x \in W.\ \neg(x \prec x)) \wedge (\forall\, x, y, z \in W.\ (x \prec y \wedge y \prec z) \Rightarrow (x \prec z))]$ is true. Observe that if $\leq$ is a partial ordering on W then $x < y = (x \leq y \wedge x \neq y)$ is a strict partial ordering on W whereas if $<$ is a strict partial ordering on W then $x \leq y = (x < y \vee x = y)$ is a partial ordering on W. A *linear ordering* on W is a strict partial ordering such that any two different elements of W are comparable: $\mathit{lo}(W, \prec) = \mathit{spo}(W, \prec) \wedge [\forall\, x, y \in W.\ ((x \neq y) \Rightarrow (x \prec y \vee y \prec x))]$. A relation $\prec$ on a class W is *well-ordered* if and only if it is a well-founded linear ordering on W: $\mathit{wo}(W, \prec) = \mathit{wf}(W, \prec) \wedge \mathit{lo}(W, \prec)$. A *well-order* is a pair $\langle W, \prec\rangle$ such that $\mathit{wo}(W, \prec)$.

To study common properties of well-ordered relations independently of their support class W, mathematicians have introduced a universal well-order called the class $\mathbb{Ord}$ of *ordinals* ordered by $<$. We say that two well-orderings $\langle W_1, \prec_1\rangle$ and $\langle W_2, \prec_2\rangle$ have the same *order type* if there exists a bijection ι from W_1 onto W_2 such that $x \prec_1 y \Leftrightarrow \iota(x) \prec_2 \iota(y)$. An ordinal can be understood as the class of all well-orderings of the same order type. Intuitively $\mathbb{Ord}$ is the transfinite sequence $0 < 1 < 2 < \ldots < \omega < \omega{+}1 < \omega{+}2 < \ldots < \omega{+}\omega = \omega.2 < \omega.2{+}1 < \omega.2{+}2 < \ldots < \omega.2{+}\omega = \omega.3 < \ldots < \omega.4 < \ldots < \omega.\omega = \omega^2 < \omega^2{+}1 < \ldots < \omega^2.\omega{=}\omega^3 < \ldots < \omega^\omega = 2_\omega < \ldots < {}_{\omega}\omega^{\omega} = 3_\omega < \ldots < \varepsilon_0 = {}_{\omega}\omega^{\omega^{\cdots}}\}\,\omega\ \text{times} = \omega_\omega < \ldots < {}^{\omega}\omega_\omega < \ldots$ and so on (although what is behind may seems inaccessible indeed ineffable). An ordinal α is a *limit ordinal* if it is neither 0 nor the successor of an ordinal that is if $\beta < \alpha$ then there is an ordinal γ such that $\beta < \gamma < \alpha$. The first limit ordinal ω is the order type of $\mathbb{N}$ well-ordered by $<$. If $C \subset \mathbb{Ord}$ then $\mathit{lub}\ C$ is the least upper bound of C $[\forall\, x \in C.\ x \leq \mathit{lub}\ C \wedge \forall\, a \in \mathbb{Ord}.\ ((\forall\, x \in C.\ x \leq a) \Rightarrow (\mathit{lub}\ C \leq a)]$ and $\mathit{lub}^+\ C$ is the least strict upper bound of C $[\forall\, x \in C.\ x < \mathit{lub}^+\ C \wedge \forall\, a \in \mathbb{Ord}.\ ((\forall\, x \in$

C. x < a) $\Rightarrow$ (lub^+ C $\leq$ a)]. A more detailed and rigorous presentation of ordinals can be found in SHOENFIELD [1977].

A well-founded relation -< on a set W can be embedded into a well-ordered relation on W (using KNUTH [1968a] topological sorting algorithm when W is finite) hence into an initial segment of the ordinals. Assuming *wf*(W, -<), we can do this by the *rank function* $rk_{(W,\ -<)}$ (for short $rk_{-<}$) defined by $rk_{-<}(x) = \mathit{lub}^+\{rk_{-<}(y) : y \text{ -< } x\}$. Minimal objects x of W (with no y -< x) will have rank 0. The objects x of W which are not minimal but which are such that y -< x only for minimal objects y of W will have rank 1, and so on. One can easily verify by induction on -< that $rk_{-<}(x)$ is an ordinal. Observe also that (x -< y) $\Rightarrow$ ($rk_{-<}(x) < rk_{-<}(y)$). We call $rk_{-<}(W) = \mathit{lub}^+\{rk_{-<}(x) : x \in W\}$ the *rank* of (W, -<).

6.4 Termination proofs by Floyd's well-founded set method

To prove termination, we must discover a well-founded (FLOYD [1967a] proposed well-ordered) relation -< on a set W and a *variant function* f : $\Gamma \rightarrow$ W and show that its value decreases after each program step: $\forall\ \gamma, \gamma' \in op[\![C]\!].\ f(\gamma')$ -< $f(\gamma)$.

Example *Proof of termination of program (4)* (71)

Program (4) was proved to be partially correct at example (24) using local invariants (25). To prove termination, let W be $\mathbb{N} \times \{L_1, \ldots, L_8\}$ and -< be the well-founded relation on W defined by <Y, L> -< <Y', L'> if and only if ($0 \leq Y < Y'$) $\vee$ ($Y = Y' \wedge L \ll L'$) where < is the usual ordering on natural numbers 0 < 1 < 2 <… and « is defined by $L_8 \ll L_6 \ll L_4 \ll L_3 \ll L_2 \ll L_7 \ll L_5 \ll L_1$. Let f : $\Gamma \rightarrow$ W be defined by $f(<L_i, s>) = f_i(s(Y))$ for i = 1, …, 7 and $f(s) = f_8(s(Y))$ where $f_i(y) = <y, L_i>$. The proof that the value of f decreases after each program step amounts to the following local arguments :

(72)

(i_1)	$[P_1(X,Y,Z,x,y) \wedge Z' = 1 \wedge P_2(X,Y,Z',x,y)] \Rightarrow f_2(Y)$ -< $f_1(Y)$	(since $L_2 \ll L_1$)	(72)
(i_2)	$[P_2(X,Y,Z,x,y) \wedge Y \neq 0 \wedge P_3(X,Y,Z,x,y)] \Rightarrow f_3(Y)$ -< $f_2(Y)$	(since $L_3 \ll L_2$)	
(i_3)	$[P_2(X,Y,Z,x,y) \wedge Y = 0 \wedge P_8(X,Y,Z,x,y)] \Rightarrow f_8(Y)$ -< $f_2(Y)$	(since $L_8 \ll L_2$)	
(i_4)	$[P_3(X,Y,Z,x,y) \wedge \mathrm{odd}(Y) \wedge P_4(X,Y,Z,x,y)] \Rightarrow f_4(Y)$ -< $f_3(Y)$	(since $L_4 \ll L_3$)	
(i_5)	$[P_3(X,Y,Z,x,y) \wedge \mathrm{even}(Y) \wedge P_6(X,Y,Z,x,y)] \Rightarrow f_6(Y)$ -< $f_3(Y)$	(since $L_6 \ll L_3$)	
(i_6)	$[P_4(X,Y,Z,x,y) \wedge Y' = Y - 1 \wedge P_5(X,Y',Z,x,y)] \Rightarrow f_5(Y')$ -< $f_4(Y)$	(since $Y'<Y$)	
(i_7)	$[P_5(X,Y,Z,x,y) \wedge Z' = Z * X \wedge P_2(X,Y,Z',x,y)] \Rightarrow f_2(Y)$ -< $f_5(Y)$	(since $L_2 \ll L_5$)	
(i_8)	$[P_6(X,Y,Z,x,y) \wedge Y' = Y \text{ div } 2 \wedge P_7(X,Y',Z,x,y)] \Rightarrow f_7(Y')$ -< $f_6(Y)$	(since $Y'<Y$)	
(i_9)	$[P_7(X,Y,Z,x,y) \wedge X' = X * X \wedge P_2(X',Y,Z,x,y)] \Rightarrow f_2(Y)$ -< $f_7(Y)$	(since $L_2 \ll L_7$)	

In practice it is only necessary to prove that all program loops terminate since the ordering « on labels directly follows from the syntactic structure of the program. ■

Floyd's method for proving termination is sound because no infinite decreasing sequence $f(\sigma_0) \gg f(\sigma_1) \gg \ldots \gg f(\sigma_i) \gg \ldots$ where $\sigma \in \sum^{\omega}[\![C]\!](p)$ is possible, so that execution of the program must sooner or later terminate in a final state $\sigma_{n-1} \in S$ since by definition of $\sum^{n}[\![C]\!](p)$ we have $\forall \gamma \in \Gamma.\ \langle\sigma_{n-1}, \gamma\rangle \notin op[\![C]\!]$.

For completeness, observe that the distance n - i - 1 of the current state σ_i to the final state σ_{n-1} of any execution trace σ of a terminating program is finite and that this distance strictly decreases after each program step. Hence we can hope to be always able to define the variant function $f(\sigma_i)$ as being this distance n - i - 1. When this is true, we say after DIJKSTRA [1976] that the program *strongly terminates*. The nondeterminism of a program C is *finite* if and only if no configuration of C can have infinitely many possible successors: $\forall \gamma \in \Gamma.\ \exists n \in \mathbb{N}.\ |\{\gamma' : \langle\gamma, \gamma'\rangle \in op[\![C]\!]\}| \leq n$. It is *bounded* when there is an upper bound on the number of possible successors: $\exists n \in \mathbb{N}.\ \forall \gamma \in \Gamma.\ |\{\gamma' : \langle\gamma, \gamma'\rangle \in op[\![C]\!]\}| \leq n$. When the nondeterminism of a program is finite then it terminates if and only if it strongly terminates. Then termination can always be proved with $(W, \prec)$ chosen as $(\mathbb{N}, <)$. We say that the nondeterminism of a program is *enumerable* if and only if any configuration of C has a enumerable set of possible successors: $\forall \gamma \in \Gamma.\ |\{\gamma' : \langle\gamma, \gamma'\rangle \in op[\![C]\!]\}| \leq |\mathbb{N}| = \omega$. When the nondeterminism of a program is enumerable but not finite, there may be infinitely many execution traces σ starting with the same given initial state $\sigma_0 = \langle s, C\rangle$ with no finite upper bound on the length n of these traces σ. In this case, the program is said to *weakly terminate*. This is the case of the following example:

(X <> 0 * (X < 0 → (X := ? ; (X < 0 → X := -X ◊ skip)) ◊ X := X - 1))

where X takes its values in $\mathbb{N}$ (i.e. the set *D* of data at (7) is $\mathbb{N}$). Then f cannot be chosen as being integer valued but we can always find a convenient well-founded range $(W, \prec)$ for f.

6.5 Liveness

BURSTALL [1974] generalized Floyd's total correctness property into *liveness*. Given a specification $\langle P, Q\rangle \in \mathcal{P}(\Gamma) \times \mathcal{P}(\Gamma \times \Gamma)$ a command C is said to be *inevitably lead from* P *to* Q (written $P \leadsto C \leadsto Q$, using a variant of LAMPORT [1977]'s notation) if any execution of C starting from an initial configuration γ of P inevitably reaches a configuration γ' such that $\langle\gamma, \gamma'\rangle \in Q$. Liveness can be defined in terms of execution traces as follows :

DEFINITION BURSTALL [1974], LAMPORT [1977], ALPERN & SCHNEIDER [1985] *Liveness* (73)

$P \rightsquigarrow C \rightsquigarrow Q \;:\; \mathcal{P}(\Gamma) \times \mathbb{C}\mathrm{om} \times \mathcal{P}(\Gamma \times \Gamma) \to \{tt, ff\}$

$P \rightsquigarrow C \rightsquigarrow Q \;= \forall \gamma \in P.\ \forall \sigma \in \sum[\![C]\!]\gamma.\ \exists i \in \mathit{dom}\ \sigma.\ \langle\gamma, \sigma_i\rangle \in Q$

In particular $\underline{[p]C[q]} = \{\langle s, C\rangle : s \in p\} \rightsquigarrow C \rightsquigarrow \{\langle\gamma, s'\rangle : \gamma \in \Gamma \wedge s' \in q\}$.

6.6 Generalization Floyd's total correctness proof method to liveness

Floyd's total correctness proof method can be generalized to liveness properties by the following induction principle:

THEOREM PNUELI [1977], COUSOT & COUSOT [1985] *Floyd's liveness proof method* (74)

$P \rightsquigarrow C \rightsquigarrow Q = [\exists \alpha \in \mathbb{O}\mathrm{rd}.\ \exists i \in \alpha \to \mathcal{P}(\Gamma \times \Gamma).$

$(\forall \gamma \in P.\ \exists \beta < \alpha.\ \langle\gamma, \gamma\rangle \in i(\beta))$ (.1)

$\wedge\ (\forall \beta < \alpha.\ (\beta > 0)$
$\Rightarrow (\forall \gamma, \gamma' \in \Gamma.\ (\langle\gamma, \gamma'\rangle \in i(\beta))$
$\Rightarrow (\exists \gamma'' \in \Gamma.\ \langle\gamma', \gamma''\rangle \in op[\![C]\!])))$ (.2)

$\wedge\ (\forall \beta < \alpha.\ (\beta > 0)$
$\Rightarrow (\forall \gamma, \gamma', \gamma'' \in \Gamma.\ (\langle\gamma, \gamma'\rangle \in i(\beta) \wedge \langle\gamma', \gamma''\rangle \in op[\![C]\!])$
$\Rightarrow (\exists \beta' < \beta.\ \langle\gamma, \gamma''\rangle \in i(\beta'))))$ (.3)

$\wedge\ (i(0) \subseteq Q)]$ (.4)

Proof

- We prove soundness ($\Leftarrow$) by reductio ad absurdum. Assume we have found $\alpha \in \mathbb{O}\mathrm{rd}$ and $i \in \alpha \to \mathcal{P}(\Gamma \times \Gamma)$ satisfying verification conditions (74) and there is a configuration $\gamma \in P$ and a trace $\sigma \in \sum[\![C]\!]\gamma$ such that $\forall j \in \mathit{dom}\ \sigma.\ \langle\gamma, \sigma_j\rangle \notin Q$. Then there would be an infinite strictly decreasing sequence of ordinals $\beta \in \mathit{seq}^\omega\ \mathbb{O}\mathrm{rd}$ such that $\forall j \in \mathbb{N}.\ j \in \mathit{dom}\ \sigma \wedge \langle\gamma, \sigma_j\rangle \in i(\beta_j)$, a contradiction since $<$ is well-founded on $\mathbb{O}\mathrm{rd}$. We define $\langle\beta_j : j \in \mathbb{N}\rangle$ inductively as follows: by (74.1), $\exists \beta_0 < \alpha.\ \langle\gamma, \gamma\rangle \in i(\beta_0)$ whence $\langle\gamma, \sigma_0\rangle \in i(\beta_0)$ since $\sigma_0 = \gamma$. If $j \in \mathit{dom}\ \sigma \wedge \langle\gamma, \sigma_j\rangle \in i(\beta_j)$ then $\beta_j \neq 0$ since otherwise by (74.4) $\langle\gamma, \sigma_j\rangle \in i(\beta_j) \subseteq Q$ so that by (74.2), $\exists \gamma \in \Gamma.\ \langle\sigma_j, \gamma\rangle \in op[\![C]\!]$ so that by (61.1), $j + 1 \in \mathit{dom}\ \sigma$. Moreover by (61.1), $\langle\sigma_j, \sigma_{j+1}\rangle \in op[\![C]\!]$ so that by (74.3), $\exists \beta_{j+1} < \beta_j.\ \langle\gamma, \sigma_{j+1}\rangle \in i(\beta_{j+1})$.
- To prove completeness ($\Rightarrow$), assume $\forall \gamma \in P.\ \forall \sigma \in \sum[\![C]\!]\gamma.\ \exists i \in \mathit{dom}\ \sigma.\ \langle\gamma, \sigma_i\rangle \in Q$. Define $\langle\underline{\gamma}', \gamma'\rangle \ll \langle\underline{\gamma}, \gamma\rangle = [\underline{\gamma} \in P \wedge \exists \sigma \in \sum[\![C]\!]\underline{\gamma}.\ \exists i \in \mathit{dom}\ \sigma.\ (\forall j < i.\ \langle\sigma_0, \sigma_j\rangle \notin Q) \wedge (\langle\sigma_0, \sigma_i\rangle \in Q) \wedge (\underline{\gamma}' = \underline{\gamma} = \sigma_0) \wedge \exists k < i.\ ((\gamma = \sigma_k) \wedge (\gamma' = \sigma_{k+1}))]$.

We prove by reductio ad absurdum that $\ll$ is well-founded on $\Gamma \times \Gamma$. Assume there is a infinite sequence $\langle \underline{\sigma}_0, \sigma_0 \rangle \gg \langle \underline{\sigma}_1, \sigma_1 \rangle \gg \ldots$. If $\underline{\sigma}_0 = \sigma_0$ then we have $(\forall k \in \mathbb{N}.\ \underline{\sigma}_k = \underline{\sigma}_0 \wedge \langle \underline{\sigma}_0, \sigma_k \rangle \notin Q \wedge \langle \sigma_k, \sigma_{k+1} \rangle \in op[\![C]\!])$ so that $\sigma \in \sum[\![C]\!]\underline{\sigma}_0$ in contradiction with $\exists\, i \in dom\ \sigma.\ \langle \gamma, \sigma_i \rangle \in Q$. If $\underline{\sigma}_0 \neq \sigma_0$ then the same reasoning can be done by concatenation of a finite prefix $\langle \underline{\sigma}'_0, \sigma'_0 \rangle \gg \ldots \gg \langle \underline{\sigma}'_k, \sigma'_k \rangle$ such that $[\underline{\sigma}_1 \in P \wedge \sigma' \in \sum[\![C]\!]\underline{\sigma}_1 \wedge \exists\, i \in dom\ \sigma'.\ \exists\, k < i.\ (\forall j < i.\ \langle \sigma'_0, \sigma'_j \rangle \notin Q) \wedge (\langle \sigma'_0, \sigma'_i \rangle \in Q) \wedge (\underline{\sigma}_0 = \underline{\sigma}_1 = \sigma'_0) \wedge (\sigma_1 = \sigma'_k) \wedge (\sigma_0 = \sigma'_{k+1})]$ to the left of this sequence $\langle \underline{\sigma}_0, \sigma_0 \rangle \gg \langle \underline{\sigma}_1, \sigma_1 \rangle \gg \ldots$.

We choose $\alpha = rk_{\ll}(\Gamma \times \Gamma)$ and $i(\beta) = \{\langle \gamma, \gamma' \rangle \in P \times \Gamma : \exists\, \sigma \in \sum[\![C]\!]\gamma.\ \exists\, i \in dom\ \sigma.\ (\forall j < i.\ \langle \sigma_0, \sigma_j \rangle \notin Q) \wedge (\langle \sigma_0, \sigma_i \rangle \in Q) \wedge (\gamma = \sigma_0) \wedge (\exists\, k \leq i.\ \gamma' = \sigma_k) \wedge (\beta = rk_{\ll}\langle \gamma, \gamma' \rangle)\}$. Obviously, $(\forall \gamma \in P.\ rk_{\ll}\langle \gamma, \gamma \rangle < \alpha \wedge \langle \gamma, \gamma \rangle \in i(rk_{\ll}\langle \gamma, \gamma \rangle))$ so that (74.1) holds. If $0 < \beta < \alpha$ and $\langle \gamma, \gamma' \rangle \in i(\beta)$ then $rk_{\ll}\langle \gamma, \gamma' \rangle$ is different from 0 so that $\exists \langle \underline{\gamma}, \underline{\gamma}' \rangle \ll \langle \gamma, \gamma' \rangle$ whence $\langle \gamma', \underline{\gamma}' \rangle \in op[\![C]\!]$ and (74.2) is true. If $0 < \beta < \alpha$, $\langle \gamma, \gamma' \rangle \in i(\beta)$ and $\langle \gamma', \gamma'' \rangle \in op[\![C]\!]$ then let $\sigma \in \sum[\![C]\!]\gamma$, $i \in dom\ \sigma$ and $k \leq i$ be such that $(\forall j < i.\ \langle \sigma_0, \sigma_j \rangle \notin Q) \wedge (\langle \sigma_0, \sigma_i \rangle \in Q) \wedge (\gamma = \sigma_0) \wedge (\gamma' = \sigma_k) \wedge (\beta = rk_{\ll}\langle \gamma, \gamma' \rangle)$. Since $rk_{\ll}\langle \gamma, \gamma' \rangle$ is different from 0, $\exists \langle \underline{\gamma}, \underline{\gamma}' \rangle \ll \langle \gamma, \gamma' \rangle$ whence $[\gamma \in P \wedge \exists\, \sigma' \in \sum[\![C]\!]\gamma.\ \exists\, i' \in dom\ \sigma'.\ (\forall j < i'.\ \langle \sigma'_0, \sigma'_j \rangle \notin Q) \wedge (\langle \sigma'_0, \sigma'_{i'} \rangle \in Q) \wedge (\underline{\gamma} = \gamma = \sigma'_0) \wedge \exists\, k' < i'.\ ((\gamma' = \sigma'_{k'}) \wedge (\underline{\gamma}' = \sigma'_{k'+1}))]$ so that $\langle \gamma, \gamma' \rangle \notin Q$, hence $\langle \sigma_0, \sigma_k \rangle \notin Q$ so that $k < i$. Moreover $\exists\, \sigma'' \in \sum[\![C]\!]\gamma$ with $\sigma''_0 = \sigma_0 = \gamma$, $\sigma''_1 = \sigma_1, \ldots, \sigma''_k = \sigma_k = \gamma'$, $\sigma''_{k+1} = \gamma''$. Then $[\exists\, i'' \in dom\ \sigma''.\ \exists\, k'' < i''.\ (\forall j < i''.\ \langle \sigma''_0, \sigma''_j \rangle \notin Q) \wedge (\langle \sigma''_0, \sigma''_{i''} \rangle \in Q) \wedge (\gamma = \gamma = \sigma''_0) \wedge (\gamma' = \sigma''_{k''}) \wedge (\gamma'' = \sigma''_{k''+1})]$ with $k'' = k$ so that $\langle \gamma, \gamma'' \rangle \ll \langle \gamma, \gamma' \rangle$. It follows that $\beta' = rk_{\ll}\langle \gamma, \gamma'' \rangle < rk_{\ll}\langle \gamma, \gamma' \rangle = \beta$ and $\langle \gamma, \gamma' \rangle \in i(\beta'')$ whence (74.3) holds. Finally, if $\langle \gamma, \gamma' \rangle \in i(0)$ and $\langle \gamma, \gamma' \rangle \notin Q$ then $rk_{\ll}\langle \gamma, \gamma' \rangle = 0$ whence there is no $\langle \underline{\gamma}, \underline{\gamma}' \rangle \ll \langle \gamma, \gamma' \rangle$ so that $[\forall\, \sigma \in \sum[\![C]\!]\gamma.\ \forall\, i \in dom\ \sigma.\ (\exists\, j < i.\ \langle \sigma_0, \sigma_j \rangle \in Q) \vee (\langle \sigma_0, \sigma_i \rangle \notin Q) \vee \forall k < i.\ ((\gamma \neq \sigma_k) \vee \forall\, \underline{\gamma}' \in \Gamma.\ (\underline{\gamma}' \neq \sigma_{k+1}))]$ so that for any $\sigma \in \sum[\![C]\!]\gamma$, choosing the least $i \in dom\ \sigma.\ \langle \sigma_0, \sigma_i \rangle \in Q$, we have $i > 1$ and $\forall k < i.\ (\gamma \neq \sigma_k)$, a contradiction for k=0. ■

6.7 Burstall total correctness proof method and its generalization

FLOYD [1967a]'s total correctness proof method is by induction on the structure of computations where computations are understood as empty or as a step followed by a computation. A la Floyd proofs are elegant for programs which exactly have this linear structure of computations. For example, this is the case for a program computing the size of a list L ::= <> | <A; tl(L)> (where A is an atom and tl(L) is a list) since its structure is of the form: size(L) = (L = <> → 0 ◊ 1 + size(tl(L)). HOARE [1969] remarked that programs (at least those written in structured languages with no gotos, etc) often have a tree-like structure of computations similar to their syntactic structure, so that correctness proofs are better handled by induction on this syntactic structure.

BURSTALL [1974] adopted the point of view that proofs are better handled by induction on the data structures manipulated by the program since the structure of the computations is often similar to that of these data structures. For example, an iterative program using a stack for computing the size of a tree T ::= <> | <lf(T); A; rg(T)> (where lf(T) and rg(T) are trees) would have a structure of computations of the form: size(T) = (L = <> → 0 ◊ 1 + size(lf(L) + size(rg(L)). The following induction principle generalizes these points of view by considering that arbitrary structures of computations can be specified by a well-founded relation (W, -<) (or from a mathematical point of view ($\mathbb{Ord}$, <)) which basis ultimately corresponds to elementary program steps:

THEOREM COUSOT & COUSOT [1987] *Burstall's liveness proof method* (75)

$$P \overset{C}{\dashrightarrow} Q = [\exists \Lambda \in \mathbb{Ord}.\ \exists \theta \in \Lambda \to \mathcal{P}(\Gamma \times \Gamma).\ \exists \alpha \in \mathbb{Ord}.\ \exists i \in \Lambda \to \alpha \to \mathcal{P}(\Gamma \times \Gamma).$$

$$(\exists \pi \in \Lambda.\ \theta_\pi \subseteq P \rceil Q)\ (.1)$$

$$\wedge\ (\forall \lambda \in \Lambda. \forall \gamma \in \Gamma.\ \exists \beta < \alpha.\ <\gamma, \gamma> \in i_\lambda(\beta)) \quad (.2)$$

$$\wedge\ (\forall \lambda \in \Lambda.\ \forall \beta < \alpha.\ \forall \gamma, \gamma' \in \Gamma.$$

$$<\gamma, \gamma'> \in i_\lambda(\beta)) \Rightarrow$$

$$[\ (\exists \gamma'' \in \Gamma.\ <\gamma', \gamma''> \in op[\![C]\!]$$

$$\wedge\ \forall \gamma'' \in \Gamma.\ (<\gamma', \gamma''> \in op[\![C]\!] \Rightarrow \exists \beta' < \beta.\ <\gamma, \gamma''> \in i_\lambda(\beta'))) \quad (.3)$$

$$\vee\ (\exists \lambda' < \lambda.\ \forall \gamma'' \in \Gamma.\ <\gamma', \gamma''> \in \theta_{\lambda'} \Rightarrow \exists \beta' < \beta.\ <\gamma, \gamma''> \in i_\lambda(\beta'))) \quad (.4)$$

$$\vee\ (<\gamma, \gamma'> \in \theta_\lambda)])] \quad (.5)$$

To prove $P \overset{C}{\dashrightarrow} Q$, we must prove $\Gamma \overset{C}{\dashrightarrow} \theta_\lambda$ for $\lambda \in \Lambda$ so that by (75.1) $\Gamma \overset{C}{\dashrightarrow} \theta_\pi = P \overset{C}{\dashrightarrow} Q$ holds. The liveness proofs $\Gamma \overset{C}{\dashrightarrow} \theta_\lambda$ for $\lambda \in \Lambda$ can be done using (75.2), (75.3) and (75.5) hence by Floyd's liveness proof method (74). However, it is better to exhibit a proof showing up the recursive structure of the computations. The basis corresponds to elementary program steps (75.3) whereas induction is described by means of lemmata $\theta_{\lambda'}$, $\lambda' < \lambda$ which are first proved to be correct ($\Gamma \overset{C}{\dashrightarrow} \theta_{\lambda'}$) and can then be used in (75.4) for proving θ_λ. More precisely, $<\gamma, \gamma'> \in i_\lambda(\beta)$ if and only if starting execution in configuration γ may lead to configuration γ' from which some final configuration $\underline{\gamma}$ such that $<\gamma, \underline{\gamma}> \in \theta_\lambda$ will inevitably be reached. To prove this we can either show by (75.3) that a single program step inevitably lead to a state γ'' with the same property ($<\gamma, \gamma''> \in i_\lambda(\beta')$) but closer to the goal (since $\beta' < \beta$) or else use lemma $\theta_{\lambda'}$ which, according to a previous proof ($\lambda' < \lambda$), states that zero or more program steps inevitably lead to a state γ'' with the same property ($<\gamma, \gamma''> \in i_\lambda(\beta')$) but closer to the goal (since $\beta' < \beta$).

7. Hoare logic

HOARE [1969] introduced the idea that partial correctness can be proved compositionally, by induction on the syntax of programs. This idea turned out to be of prime importance in other domains such as denotational semantics where "The values of expressions are determined in such a way that the value of a whole expression depends functionally on the values of its parts - the exact connection being found through the clauses of the syntactical definition of the language" (SCOTT & STRACHEY [1972]) but had to be slightly modified to take context-sensitive properties of languages into account.

HOARE [1969] also introduced the idea that such proofs can be formalized using a formal logic. The first motivation is that "axioms may provide a simple solution to the problem of leaving certain aspects of a language undefined". To illustrate this point of view, Hoare gives the example of addition (+) and multiplication (*) of natural numbers. These operations can be formalized by a few axioms of which $\mathbb{N}$ is a model. They can also be given different consistent interpretations corresponding to various possible implementations $\oplus$ and $\otimes$ of + and * in a machine where only a finite subset {0, …, maxint} of $\mathbb{N}$ is representable. These interpretations include modulo arithmetic $(x \oplus y) = (x + y)$ *mod* (maxint+1), firm boundary arithmetic $(x \oplus y) = (x + y > \text{maxint} \rightarrow \text{maxint} \lozenge x + y)$ and overflow arithmetic $(x \oplus y) = (x + y > \text{maxint} \rightarrow \text{undefined} \lozenge x + y)$. More generally the idea would be that a program text may have different interpretations (a computer scientist would say computer implementations) but its correctness should be established once for all its possible interpretations (hence in a machine-independent way). This leads to the second motivation of Hoare's axiomatic semantics: "the specification of proof techniques provides an adequate formal definition of a programming language". The idea first appeared in FLOYD [1967a] and was illustrated by HOARE & WIRTH [1973] on a part of Pascal. The trouble with this axiomatic semantics is that non-standard hence computer unimplementable interpretations are not ruled out (BERGSTRA & TUCKER [1984], WAND [1978]).

7.1 Hoare logic considered from a semantical point of view

7.1.1 General theorems for proof construction

In paragraph § 5.3, we have considered Hoare logic from a semantical point of view that is to say with respect to the conventional operational semantics (13). In summary this essentially consists in the natural extension of the relational semantics (19) of commands from pairs of states to pairs of sets of states (22). This leads to the proof of partial correctness by structural induction on commands using theorem (49) which can also be rephrased as follows:

THEOREM HOARE [1969], COOK [1978] *Semantic interpretation of Hoare logic* (76)

$\underline{\{p\}\mathrm{skip}\{p\}} = tt$ (.1)

$\underline{\{\{s \in S : s[X \leftarrow \underline{E}(s)] \in p\}\}X := E\{p\}} = tt$ (.2)

$\underline{\{p\}X := ?\{\{s[X \leftarrow d] : s \in p \wedge d \in D\}\}} = tt$ (.3)

$\underline{\{p\}(C_1; C_2)\{q\}} = (\exists i \in \mathrm{Ass}. \underline{\{p\}C_1\{i\}} \wedge \underline{\{i\}C_2\{q\}})$ (.4)

$\underline{\{p\}(B \rightarrow C_1 \lozenge C_2)\{q\}} = (\underline{\{p \cap \underline{B}\}C_1\{q\}} \wedge \underline{\{p \cap \neg \underline{B}\}C_2\{q\}})$ (.5)

$\underline{\{p \cap \underline{B}\}C\{p\}} \Rightarrow \underline{\{p\}(B * C)\{p \cap \neg\underline{B}\}}$ (.6)

$(\exists p', q' \in \mathrm{Ass}. (p \subseteq p') \wedge \underline{\{p'\}C\{q'\}} \wedge (q \subseteq q')) \Rightarrow \underline{\{p\}C\{q\}}$ (.7)

In these theorems, the consequence rule (76.7) has been isolated whereas in (49) it is distributed over all theorems (76.1) to (76.6). The idea is interesting because proofs concerning properties of objects manipulated by the program (which turn out to be always of the form $p \subseteq p'$) are isolated from proofs concerning the computation (sequence of operations) performed by the program. In practice this separation leads to excessively tedious proofs. The method proposed by HOARE [1979] "of reducing the tedium of formal proofs is to derive general rules for proof construction out of the simple rules accepted as postulates". For example derived theorems such as "$(p \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in q\}) \Rightarrow \underline{\{p\}X := E\{q\}}$" or "$(\{s'[X \leftarrow \underline{E}(s')] : s' \in p\} \subseteq q) \Rightarrow \underline{\{p\}X := E\{q\}}$" are directly applicable hence often more useful than (76.2). Also the reciprocal of (76.6) is not true (for example "$\{X = 0\}$ **while** true **do** $X := X + 1$ $\{X = 0 \wedge \neg\text{true}\}$" holds but "$\{X = 0 \wedge \text{true}\}$ $X := X + 1$ $\{X = 0\}$" does not). Hence (76.6) does not make completely clear the fact that a loop invariant has to be found (most often different from the precondition and postcondition). In practice, we prefer a more explicit formulation :

COROLLARY *Partial correctness proof of while loops* (77)

$\underline{\{p\}(B * C)\{q\}} = (\exists i \in \mathrm{Ass}. (p \subseteq i) \wedge \underline{\{i \cap \underline{B}\}C\{i\}} \wedge ((i \cap \neg\underline{B}) \subseteq q))$

Example *Partial correctness proof of assignments* (78)

Let us derive "$(\{s'[X \leftarrow \underline{E}(s')] : s' \in p\} \subseteq q) \Rightarrow \underline{\{p\}X := E\{q\}}$" from (76) :

(a)	$\{s'[X \leftarrow \underline{E}(s')] : s' \in p\} \subseteq q$	by assumption,
(b)	$(s \in p) \Rightarrow (s[X \leftarrow \underline{E}(s)] \in q)$	by (a) and set theory,
(c)	$p \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in q\}$	by (b) and set theory,
(d)	$\underline{\{\{s \in S : s[X \leftarrow \underline{E}(s)] \in q\}\}X := E\{q\}}$	by (76.2),
(e)	$q \subseteq q$	from set theory,
(f)	$\underline{\{p\}X := E\{q\}}$	by (c), (d), (e) and (76.7).

■

Proof of theorems (76) and (77)

- $\underline{\{p\}\mathrm{skip}\{p\}} = ((p \rceil \underline{\mathrm{skip}}) \subseteq (S \times p))$ [by (22)] $= ((p \rceil \{<s, s> : s \in S\}) \subseteq (S \times p))$ [by (19.1)] = tt.
- $\underline{\{\{s \in S : s[X \leftarrow \underline{E}(s)] \in p\}\}X := E\{p\}} = ((\{s \in S : s[X \leftarrow \underline{E}(s)] \in p\} \rceil \{<s, s[X \leftarrow \underline{E}(s)]> : s \in S\}) \subseteq (S \times p))$ [by (22) and (19.2)] $= (\{<s, s[X \leftarrow \underline{E}(s)]> : s[X \leftarrow \underline{E}(s)] \in p\} \subseteq \{<s, s'> : s' \in p\})$ = tt .
- $\underline{\{p\}X := ?\{\{s[X \leftarrow d] : s \in p \wedge d \in D\}\}}$ is tt since $(p \rceil \underline{X := ?}) = \{<s, s[X \leftarrow d]> : s \in p \wedge d \in D\}$ by (22) and (19.3).
- $\underline{\{p\}(C_1; C_2)\{q\}} = ((\forall s, s', s''.(s \in p \wedge <s, s'> \in \underline{C_1} \wedge <s', s''> \in \underline{C_2}) \Rightarrow s'' \in q)$ by (22) and (19.4). This implies that if we let i bet $\{s' : \exists s \in p. <s, s'> \in \underline{C_1}\}$ then $(p \rceil \underline{C_1}) \subseteq (S \times i)$ and $(i \rceil \underline{C_2}) \subseteq (S \times q)$ whence $\underline{\{p\}C_1\{i\}} \wedge \underline{\{i\}C_2\{q\}}$ holds. Reciprocally if $i \in \mathbb{A}\mathrm{ss}$ is such that $\underline{\{p\}C_1\{i\}} \wedge \underline{\{i\}C_2\{q\}}$ then $(p \rceil \underline{C_1}) \subseteq (S \times i)$ and $(i \rceil \underline{C_2}) \subseteq (S \times q)$ by (22) so that $(p \rceil (\underline{C_1; C_2})) = (p \rceil (\underline{C_1} \circ \underline{C_2})) = ((p \rceil \underline{C_1}) \circ \underline{C_2}) \subseteq ((S \times i) \circ \underline{C_2}) = (S^2 \circ (i \rceil \underline{C_2})) \subseteq (S^2 \circ (S^2 \rceil q)) = (S^2 \rceil q)$.
- $\underline{\{p\}(B \rightarrow C_1 \lozenge C_2)\{q\}} = [((p \cap \underline{B}) \rceil \underline{C_1} \cup (p \cap \neg\underline{B}) \rceil \underline{C_2}) \subseteq (S \times q)] = (\underline{\{p \cap B\}C_1\{q\}} \wedge \underline{\{p \cap \neg B\}C_2\{q\}})$ by (22) and (19.5).
- $\underline{\{p\}(B * C)\{q\}} = ((p \rceil (\underline{B} \rceil \underline{C})^* \lceil \neg\underline{B}) \subseteq S \times q)$ [by (22) and (19.6)] $= (\exists i \in \mathbb{A}\mathrm{ss}. (p \subseteq i) \wedge (i \rceil (\underline{B} \rceil \underline{C}) \subseteq S \times i) \wedge ((i \cap \neg\underline{B}) \subseteq q))$ [by (28)] $= (\exists i \in \mathbb{A}\mathrm{ss}. (p \subseteq i) \wedge \underline{\{i \cap B\}C\{i\}} \wedge ((i \cap \neg\underline{B}) \subseteq q))$ [by (22)]. It follows that $\underline{\{p \cap B\}C\{p\}} \Rightarrow ((p \subseteq p) \wedge \underline{\{p \cap B\}C\{p\}} \wedge ((p \cap \neg\underline{B}) \subseteq (p \cap \neg\underline{B}))) \Rightarrow \underline{\{p\}(B * C)\{p \cap \neg B\}}$.
- $((p \subseteq p') \wedge \underline{\{p'\}C\{q'\}} \wedge (q \subseteq q')) \Rightarrow ((p \subseteq p') \wedge ((p' \rceil \underline{C}) \subseteq (S \times q')) \wedge (q \subseteq q')) \Rightarrow ((p \rceil \underline{C}) \subseteq (S \times q)) \Rightarrow \underline{\{p\}C\{q\}}$ by (22). ■

7.1.2 Semantical soundness and completeness

If we have proved $\underline{\{p\}C\{q\}}$ by application of theorems (76) to components $C' \in \mathbb{C}\mathrm{omp}[\![C]\!]$ of C then we conclude, by structural induction, that $\underline{\{p\}C\{q\}}$ holds. This is called *semantic soundness*. If we can prove by (22) using (13) that $\underline{\{p\}C\{q\}}$ holds, then this can always be proved by application of theorems (76) to components $C' \in \mathbb{C}\mathrm{omp}[\![C]\!]$ of C. This is called *semantic completeness*. We use the epithet “semantic” because soundness and completeness are relative to partial correctness as defined by (22) with respect to operational semantics (13), that is are defined in terms of mathematical structures without reference to a particular logical language for assertions.

THEOREM *Semantical soundness and completeness of Hoare logic*

Hoare's partial correctness proof method (76) is semantically sound. (79)

Hoare's partial correctness proof method (76) is semantically complete. (80)

Proof

The proof that $\underline{\{p\}C\{q\}}$ is provable by (76) is by structural induction on C.

- If $\underline{\{p\}skip\{q\}}$ holds then $p \subseteq q$ by (22) and (19.1). Also $p \subseteq p$. Therefore the proof is "$\underline{\{p\}skip\{p\}}$ by (76.1) whence $\underline{\{p\}skip\{q\}}$ by $p \subseteq p$, $p \subseteq q$ and (76.7)".
- If $\underline{\{p\}X := E\{q\}}$ holds then $\{s'[X \leftarrow \underline{E}(s')] : s' \in p\} \subseteq q$ by (22) and (19.2). Also $p \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in \{s'[X \leftarrow \underline{E}(s')] : s' \in p\}\}$. Therefore the proof is simply as follows: "$\underline{\{}\{s \in S : s[X \leftarrow \underline{E}(s)] \in \{s'[X \leftarrow \underline{E}(s')] : s' \in p\}\}\underline{\}}\ \underline{X := E}\ \underline{\{}\{s'[X \leftarrow \underline{E}(s')] : s' \in p\}\underline{\}}$ by (76.2) whence $\underline{\{p\}X := E\{q\}}$ by $p \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in \{s'[X \leftarrow \underline{E}(s')] : s' \in p\}\}$, $\{s'[X \leftarrow \underline{E}(s')] : s' \in p\} \subseteq q$ and (76.7)".
- If $\underline{\{p\}X := ?\{q\}}$ holds then $\{s[X \leftarrow d] : s \in p \wedge d \in D\} \subseteq q$ by (22) and (19.3). Also $p \subseteq p$. Therefore the proof is "$\underline{\{p\}X := ?\{}\{s[X \leftarrow d] : s \in p \wedge d \in D\}\underline{\}}$ by (76.3) whence $\underline{\{p\}X := ?\{q\}}$ by $p \subseteq p$, $\{s[X \leftarrow d] : s \in p \wedge d \in D\} \subseteq q$ and (76.7)".
- If $\underline{\{p\}C_1; C_2\{q\}}$ holds then by (76.4) there is an assertion $i \in \mathbb{A}ss$ such that $\underline{\{p\}C_1\{i\}}$ and $\underline{\{i\}C_2\{q\}}$ are true. Hence, by induction hypothesis, $\underline{\{p\}C_1\{i\}}$ and $\underline{\{i\}C_2\{q\}}$ are provable by induction on C_1 and C_2 using theorems (76). Then applying (76.4), we conclude $\underline{\{p\}C_1; C_2)\{q\}}$.
- If $\underline{\{p\}(B \to C_1 \lozenge C_2)\{q\}}$ holds then by (76.5) $\underline{\{p \cap B\}C_1\{q\}}$ and $\underline{\{p \cap \neg B\}C_2\{q\}}$ are true whence provable by induction on C_1 and C_2 using theorems (76). Then we conclude by (76.5).
- If $\underline{\{p\}(B * C)\{q\}}$ holds then by (77) there is an assertion $i \in \mathbb{A}ss$ such that $(p \subseteq i)$, $\underline{\{i \cap B\}C\{i\}}$ and $(i \cap \neg\underline{B} \subseteq q)$ are true. Hence, by induction hypothesis, $\underline{\{i \cap B\}C\{i\}}$ is provable by induction on C using theorems (76). The proof goes on with "$\underline{\{i\}(B * C)\{}i \cap \neg\underline{B}\underline{\}}$ by (76) whence $\underline{\{p\}(B * C)\{q\}}$ by $(p \subseteq i)$, $(i \cap \neg\underline{B} \subseteq q)$ and (76.7)". ■

7.1.3 Proof outlines

HOARE [1969] was aware that a formal proof is often tedious and claimed that "it would be fairly easy to introduce notational conventions which would significantly shorten it". From a practical point of view, it is indeed essential to be able to present proofs by Hoare's method informally together with the program text. OWICKI [1975] showed that a proof à la Hoare can be presented as a *proof outline* that is to say as a proof à la Floyd where local invariants are attached to program points:

DEFINITION OWICKI [1975] *Proof outline* (81)

The proof outline of a proof of $\underline{\{p\}C\{q\}}$ by theorems (76) is the triple pre, post : $\mathbb{C}omp[\![C]\!] \to \mathbb{A}ss$, linv : $\mathbb{L}oops[\![C]\!] \to \mathbb{A}ss$ such that pre(C) = p, post(C) = q and by structural induction on $C' \in \mathbb{C}omp[\![C]\!]$:

- If C' is $(B \to C_1 \lozenge C_2)$ then $pre(C_1) = pre(C') \cap \underline{B}$, $pre(C_2) = pre(C') \cap \neg\underline{B}$, $post(C_1) = post(C_1) = post(C')$,

- If C' is $(C_1; C_2)$ then we can only prove $\underline{\{pre(C')\}(C_1; C_2)\{post(C')\}}$ by application of (76.4) once and of (76.7) $n \geq 0$ times. Therefore we have $pre(C') = p_1 \subseteq \ldots \subseteq p_n$, $\underline{\{p_n\}C_1\{i\}}$, $\underline{\{i\}C_2\{q_n\}}$, $q_n \subseteq \ldots \subseteq q_1 = post(C')$. We let $pre(C_1) = pre(C')$, $post(C_1) = pre(C_2) = i$ and $post(C_2) = post(C')$.
- If C' is $(B * C_1)$ then we can only prove $\underline{\{pre(C')\}(B * C_1)\{post(C')\}}$ by application of (76.6) once and of (76.7) $n \geq 0$ times. Therefore we have $pre(C') = p_1 \subseteq \ldots \subseteq p_n = i$, $\underline{\{i \cap \underline{B}\}C_1\{i\}}$, $i \cap \neg\underline{B} = q_n \subseteq \ldots \subseteq q_1 = post(C')$. We let $pre(C_1) = i \cap \underline{B}$, $post(C_1) = i$ and $linv(C') = i$.

The following two theorems show that Hoare's method (76) is equivalent to the syntax-directed presentation of Floyd's method of paragraph 5.3, hence by (55) is semantically equivalent to Floyd's original stepwise proof method, otherwise stated that assertions in a proof outline are local invariants:

THEOREM *À la Floyd presentation of a proof by Hoare logic* (82)

A proof outline of $\underline{\{p\}C\{q\}}$ by Hoare's method (76) satisfies (49)

Proof

(49.1) is true by definition (81). If C is skip then $\underline{\{p\}skip\{q\}}$ can only be proved using (76.1) and (76.7). It follows that $p \subseteq q$ so that (49.2) holds. The proof is similar when C is X := E or X := ?. (49.5), (49.6) and (49.7) directly follow from definition (81). ■

THEOREM *À la Hoare presentation of a proof by Floyd's method* (83)

If assertions attached to program points are local invariants in the sense of Floyd (i.e. satisfies (45) or equivalently by (54) and (55) satisfies (49)) then they can be used to prove partial correctness by Hoare's method (76).

Proof

The proof is by structural induction on C. If C is X := E then the proof à la Hoare is "$p \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in q\}$ [which is true by (81) and (49.3)], $\underline{\{\{s \in S : s[X \leftarrow \underline{E}(s)] \in q\}\}X := E\{q\}}$ [by (76.2) and $q \subseteq q$] hence $\underline{\{p\}X := E\{q\}}$ [by (76.7)]". The proofs are similar when is is skip or X := ?.

If C is $(C_1; C_2)$ then by induction hypothesis there are proofs à la Hoare of $\underline{\{pre(C_1)\}C_1\{post(C_1)\}}$ and $\underline{\{pre(C_2)\}C_2\{post(C_2)\}}$. It follows from (81) and (76.4) that $p = pre(C_1)$ and $post(C_2) = q$. If we let i be $post(C_1) = pre(C_2)$ then we can use (76.7) to prove $\underline{\{p\}C_1\{i\}} \wedge \underline{\{i\}C_2\{q\}}$ and conclude $\underline{\{p\}C_1\{i\}}$ by (76.4). The proofs are similar when C is $(B \rightarrow C_1 \lozenge C_2)$ or $(B * C_1)$. ■

7.2 Hoare logic considered from a syntactical point of view

The function of mathematical logic is to provide formal languages for describing the structures with which the mathematician work, and to study the methods of proof available to them. HOARE [1969] introduced the same idea in computer science: "Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs".

Hoare logic $\mathbb{HL}$ consists of first order predicates P, Q, … for describing assertions p, q, … and correctness formulae {P}C{Q} for describing the partial correctness <u>{p}C{q}</u> of commands C. First we define the set $\mathbb{HL}$ of Hoare's formulae that is the syntax of predicates P, Q, … and correctness formulae {P}C{Q}. We also fill in the details of the syntax (1) of expressions and Boolean expressions of the programming language $\mathbb{Com}$. This syntactical definition is parametrized by a *basis* ∑ = <$\mathbb{Cte}$, $\mathbb{Fun}$, $\mathbb{Rel}$, #> (also called *signature*, *type*,…) that is sets of constant, function and relation symbols together with their arity.

Then we introduce Hoare's proof system to define inductively which formulae of Hoare logic are <u>provable to be true</u>. The proof system consists of sets of postulates (axioms) and of syntactic rules of manipulation of formulae by rewritting (rules of inference) to logically derive conclusions from hypotheses. These axioms and rules of inference are defined finistically so that given formal proofs are checkable by algorithmic means (although the proof itself may not be derivable by a machine). This approach is "syntactical" in that the emphasis is upon a formal language for describing assertions about the values taken by the program variables and upon a formal deductive system for deriving proofs based upon combinatorial manipulations of formal assertions. Proofs are parametrized by a set of axioms, which are supposed to describe properties of the data manipulated by the programs into all the details of which we do not want to enter. This is usual in logic where the primitive notions (0, +, …) of mathematical theories (such as groups,…) have no fixed meaning. This is also consistent with HOARE [1969] point of view that "Another of the great advantages of using an axiomatic approach is that axioms offer a simple and flexible technique for leaving certain aspects of a language *undefined* for example, range of integers, accuracy of floating point, and choice of overflow technique. This is absolutely essential for standardization purposes, since otherwise the language will be impossible to implement efficiently on different hardware designs. Thus a programming language standard should consist of a set of axioms of universal applicability, together with a choice of supplementary axioms describing the range of choices facing an implementor".

Then we define which mathematical structures can be understood as being models or interpretations of Hoare logic. Otherwise stated we define which formulae of Hoare logic $\mathbb{HL}$ are semantically true with respect to a relational semantics $\underline{C}$ of programs $C \in \mathbb{Com}$. This programming language semantics (19) itself depends upon the semantics (also called a *model* or *interpretation*) $<D, V>$ of the basis $\sum = <\mathbb{Cte}, \mathbb{Fun}, \mathbb{Rel}, \#>$ that is upon the domain D of data on which programs operate and the exact interpretation $V[\![c]\!]$ of the basic constants $c \in \mathbb{Cte}$, $V[\![f]\!]$ of the functions $f \in \mathbb{Fun}$ and $V[\![r]\!]$ of the relations $r \in \mathbb{Rel}$ involved in the programs and predicates. By leaving this interpretation as a parameter, we define a family of semantics of Hoare logic with respect to a family of possible operational semantics of the programming language.

Kurt Gödel showed that truth and provability do not necessarily coincide: provable implies true, refutable implies false but some formulae may be undecidable that is neither provable nor refutable (using proofs that can be checked mechanically) although they are either true or false (SMORYNSKI [1977]). Therefore the question is whether Hoare's formal proof system captures the true partial correctness formulae, only these (soundness) and ideally all of these (completeness).

7.2.1 Syntax of predicates and correctness formulae

We have defined the syntax of the programming language $\mathbb{Com}$ at (1). We now define the syntax of the logical language $\mathbb{HL}$ which will be used to specify the partial correctness of programs. Hoare logic is a first-order language allowing only to quantify over elements, but not over subsets or functions (for example "$\forall A \in \mathcal{P}(\mathbb{N}). (0 \in A \wedge \forall x \in A. (x + 1) \in A) \Rightarrow (A = \mathbb{N})$" or "$\forall P. \forall x_1. \ldots \forall x_n. \{P\}\mathrm{skip}\{P\}$" are not a first-order sentences).

In order to specify the syntax of predicates, we consider given disjoint sets of symbols as follows :

DEFINITION *Symbols* (84)

X, Y	: $\mathbb{Pvar}$	*Programming variables*	(.1)
x, y	: $\mathbb{Lvar}$	*Logical variables*	(.2)
v, u	: $\mathbb{Var} = \mathbb{Pvar} \cup \mathbb{Lvar}$	*Variables*	(.3)
c	: $\mathbb{Cte}$	*Constant symbols*	(.4)
f	: $\mathbb{Fun}$	*Function symbols*	(.5)
r	: $\mathbb{Rel}$	*Relation symbols*	(.6)

DEFINITION *Arity of function and relation symbols* (85)

$\# : \quad \mathbb{Fun} \cup \mathbb{Rel} \to \mathbb{N}^+$

At any moment we shall use only a finite number of variables but assume that we are given a countably infinite supply of these (in the examples we use capital letters for programming variables and lower-case letters for logical variables). The basis $\sum$ = <$\mathbb{Cte}$, $\mathbb{Fun}$, $\mathbb{Rel}$, #> of the logical language $\mathbb{HL}$ is assumed to be given but is otherwise left unspecified.

The purpose of the syntactical definition of the logical language $\mathbb{HL}$ is to define algorithmically which finite strings of symbols (chosen in $\mathbb{Var} \cup \mathbb{Cte} \cup \mathbb{Fun} \cup \mathbb{Rel} \cup$ {(,), =, ¬, ∧, ∨, ⇒, ∃, ∀, ., {, }, skip, :=, ?, ;, →, ◊, *}) belong to $\mathbb{HL}$, logicians would say are well-formed formulae. The sets of *terms*, *atomic formulae*, *first-order predicates*, *correctness formulae* and *formulae* are the smallest sets closed under the following formation rules (from which a recognizer is easy to derive, see for example AHO, SETHI & ULLMAN [1986]):

DEFINITION *Syntax of formulae*

T : $\mathbb{Ter}$ — *Terms*

$T ::= v \mid c \mid f(T_1, \ldots, T_{\#f})$ (86)

A : $\mathbb{Afo}$ — *Atomic formulae*

$A ::= (T_1 = T_2) \mid r(T_1, \ldots, T_{\#r})$ (87)

P, Q : $\mathbb{Pre}$ — *Predicates*

$P ::= A \mid \neg P \mid (P_1 \wedge P_2) \mid (P_1 \vee P_2) \mid (P_1 \Rightarrow P_2) \mid \exists\, x.\, P \mid \forall\, x.\, P$ (88)

H : $\mathbb{Hcf}$ — *Hoare correctness formulae*

$H ::= \{P\}C\{Q\}$ (89)

F : $\mathbb{HL}$ — *Formulae of Hoare logic*

$F ::= P \mid H$ (90)

(in (87), “=” is a relation of arity 2 in infix notation. The reason why we keep it separate from the relations in $\mathbb{Rel}$ is that its intended interpretation is fixed as the diagonal (equality) relation δ whereas the interpretations of members of $\mathbb{Rel}$ can be specified arbitrarily).

In paragraph § 3.1 the syntax of expressions was left unspecified. From now on, we assume that $\mathbb{Expr}$ is included in the set of terms containing only programming variables and that $\mathbb{BExp}$ is included in the set of *propositions* (i.e. quantifier-free predicates) containing only programming variables:

DEFINITION *Syntax of expressions*

E : $\mathbb{Expr}$ *Expressions*

$$E ::= X \mid c \mid f(E_1, \ldots, E_{\#f}) \tag{91}$$

B : $\mathbb{BExp}$ *Boolean expressions*

$$B ::= (E_1 = E_2) \mid r(E_1, \ldots, E_{\#r}) \mid \neg B \mid (B_1 \wedge B_2) \mid (B_1 \vee B_2) \mid (B_1 \Rightarrow B_2) \tag{92}$$

Definitions (84) to (90) are justified by the (restrictive) assumption that all we shall ever have to say about programs is expressible by sentences of $\mathbb{HL}$. To formally define the *axiomatic semantics* of the programming language $\mathbb{Com}$, it only remains to define exactly all we can assert to be true about programs. This consists in partitioning $\mathbb{HL}$ into $\mathbb{HL}_{tt}$ (what is truth) and $\mathbb{HL}_{ff}$ (what is falsity). To do this logicians have proposed to complementary approaches:

- The *semantical point of view* consists in defining an interpretation $I : \mathbb{HL} \rightarrow \{tt,ff\}$ with $\mathbb{HL}_{tt} = \{F \in \mathbb{HL} : I[\![F]\!] = tt\}$ and $\mathbb{HL}_{ff} = \{F \in \mathbb{HL} : I[\![F]\!] = ff\}$.
- The *syntactical point of view* consists in defining which sentences of $\mathbb{HL}$ are provable to be true (with given limited means, so that (hopefully) proofs can be checked by mechanical computation).

We first start with provability.

7.2.2 Deductive systems and formal proofs

The basis of the inductive definition of formal proofs for a logical language $\mathbb{HL}$ is provided by a set of *axioms*, that is a set of formulae of $\mathbb{HL}$ the truth of which is postulated. Since the set of axioms is usually not finite, we use a finite set $AS = \{AS_i : i \in \Delta_\alpha\}$ of *axiom schemata* AS_i the instances of which are axioms. Otherwise stated an axiom schema AS_i is a syntactic rule specifying a set $\{A \in \mathbb{HL} : \text{IsAxiom}(A, AS_i)\}$ of axioms by their syntax. This set of axioms is said to be *recursive* because membership is *decidable* that is there is a program IsAxiom(A, AS_i) (called a recognizer) which given any formula $A \in \mathbb{HL}$ will always terminate and answer "tt" or "ff" whether or not formula A belongs to the set of axioms generated by the axiom schema AS_i.

The induction step in the definition of formal proofs is provided by a set of *inferences* $<F_1, \ldots, F_n, F> \in \mathbb{HL}^{n+1}$ with $n \geq 1$, traditionally written under the form:

$$\frac{F_1, \ldots, F_n}{F}$$

which means that if formulae $F_1, \ldots, F_n$ are provable then so is formula F. A further requirement is again that this set of valid inferences should be recursive. Hence the set of inferences is usually specified by a finite set $IR = \{IR_i : i \in \Delta_\iota\}$ of syntactical rules (called *rules of inference*) so that there is a program $\mathrm{IsInference}(F_1, \ldots, F_n, F, IR_i)$ which given any formulae $F_1, \ldots, F_n, F$ of $\mathbb{HL}$ will always terminate and answer "tt" or "ff" whether or not the inference is correct according to one of the inferences generated by the rule IR_i.

The *deductive system* $\mathbb{H}$ is the (recursive) set of all axioms generated by the axiom schemata and all inferences generated by the rules of inference:

DEFINITION *Deductive system* (93)

$AS = \{AS_i : i \in \Delta_\alpha\}$ *Axiom schemata*,

$\mathrm{IsAxiom}(A, AS_i)$ *Axiom recognizer*

$IR = \{IR_i : i \in \Delta_\iota\}$ *Rules of inference*

$\mathrm{IsInference}(F_1, \ldots, F_n, F, IR_i)$ *Inference recognizer*

$\mathbb{H} = \bigcup\{\{A \in \mathbb{HL} : \mathrm{IsAxiom}(A, AS_i)\} : i \in \Delta_\alpha\}$ *Deductive system*

$\cup \bigcup\{\{<F_1, \ldots, F_n, F> \in \mathbb{HL}^{n+1} : \mathrm{IsInference}(F_1, \ldots, F_n, F, IR_i)\} : i \in \Delta_\iota\}$

In order to be able to leave unspecified some aspects of the programming language $\mathbb{Com}$ (such as machine dependent features) we assume that we are given an additional set Th $\subseteq \mathbb{HL}$ of axioms, the truth of which is taken for granted. Th can be understood as a *specification* of the data and operations on these data which are used by the programs of $\mathbb{Com}$. A *proof* of F from Th in the deductive system $\mathbb{H}$ is a finite sequence $F_0,\ldots,F_n$ of formulae, with $F_n = F$, each of which is either a member of Th, an axiom of $\mathbb{H}$, or else follows from earlier F_i by one of the inferences of $\mathbb{H}$:

DEFINITION *Formal proof* (94)

$\mathrm{Proof}(F_0, \ldots, F_n, F, Th, \mathbb{H}) =$

$[F_n = F] \wedge [\forall i \in \{0,\ldots,n\}.\ [(F_i \in Th) \vee (\exists k \in \Delta_\alpha.\ \mathrm{IsAxiom}(F_i, AS_k)) \vee$

$(\exists k \in \Delta_\iota.\ \exists m \geq 1.\ \exists j_1 < i, \ldots, \exists j_m < i.\ \mathrm{IsInference}(F_{j1}, \ldots, F_{jm}, F_i, IR_k))]]$

Clearly from the above specification we can write a simple combinatorial program to check proofs (provided Th is recursive). Using the more traditional notations of logic (BARWISE [1977]) we say that F *is provable from* Th $\subseteq \mathbb{HL}$ in $\mathbb{H}$, and write $\vdash_{Th \cup \mathbb{H}} F$, if there is a proof of F from Th in $\mathbb{H}$:

DEFINITION *Provability* (95)

$\vdash_{Th \cup \mathbb{H}} F$ if $F \in Th$ (.1)

$\vdash_{Th \cup \mathbb{H}} F$ if $F \in \mathbb{H}$ (.2)

$\vdash_{Th \cup \mathbb{H}} F$ if $\vdash_{Th \cup \mathbb{H}} F_1, \ldots, \vdash_{Th \cup \mathbb{H}} F_m$ and $\dfrac{F_1, \ldots, F_m}{F} \in \mathbb{H}$ (.3)

7.2.3 Hoare's proof system $\mathbb{H}$

In HOARE [1969]'s proof system $\mathbb{H}$ below, $P[v \leftarrow T]$ denotes the predicate obtained by simultaneously substituting term T for all free occurrences of variable v in predicate P. If the substitution would cause an identifier in T to become bound (e.g. $\forall x .(f(x) = y)[y \leftarrow x]$) then a suitable replacement of bound identifiers in P must take place before the substitution in order to avoid the conflict (e.g.$\forall z. (f(z) = y)[y \leftarrow x]$ is $\forall z. (f(z) = x)$). Substitution will be defined more rigorously later on by (118).

DEFINITION *Hoare proof system*

Axiom schemata of $\mathbb{H}$:

$\{P\}$ skip $\{P\}$	*Skip axiom*	(96)
$\{P[X \leftarrow E]\}$ X := E $\{P\}$	*(Backward) assignment axiom*	(97)
$\{P\}$ X := ? $\{\exists X.\ P\}$	*Random assignment axiom*	(98)

(A schema of axioms of the form $\{P\}$skip$\{P\}$ means that $\forall v_1. \ldots \forall v_n. \{Q\}$ skip $\{Q\}$ is true for all formulae $Q \in \mathbb{Pre}$ with free variables $v_1,\ldots,v_n$. This is an approximation of the second-order axiom $\forall P. \forall v_1. \ldots \forall v_n. \{P\}$ skip $\{P\}$ where quantification over predicates is mimicked by a recursive set of axioms corresponding to all possible instances Q of quantified predicate P.)

Rules of inference of $\mathbb{H}$:

$$\frac{\{P_1\}\ C_1\ \{P_2\},\ \{P_2\}\ C_2\ \{P_3\}}{\{P_1\}\ (C_1;\ C_2)\ \{P_3\}}$$ *Composition rule* (99)

$$\frac{\{P \wedge B\}\ C_1\ \{Q\},\ \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ (B \rightarrow C_1 \lozenge C_2)\ \{Q\}}$$ *Conditional rule* (100)

$$\frac{\{P \wedge B\}\ C\ \{P\}}{\{P\}(B * C)\ \{P \wedge \neg B\}}$$ *While rule* (101)

$$\frac{P \Rightarrow P',\ \{P'\}\ C\ \{Q'\},\ Q' \Rightarrow Q}{\{P\}\ \ C\ \ \{Q\}}$$ *Consequence rule* (102)

The backward assignment axiom (97) which corresponds to (45.3) can be given an equivalent forward form corresponding to (45.8) as proposed by FLOYD [1967a]:

{P} X := E {∃ X'. P[X ← X'] ∧ X = E[X ← X']}   *(Forward) assignment axiom*   (103)

Example *Formal partial correctness proof of program (4) using* $\mathbb{H}$ (104)

(a) {(Y-1)≥0 ∧ Z*X*(X**(Y-1))=x**y} Y := Y-1 {Y≥0 ∧ Z*X*(X**Y)=x**y}   by (97)

(b) {Y≥0 ∧ Z*X*(X**Y)=x**y} Z := Z*X {Y≥0 ∧ Z*(X**Y)=x**y}   by (97)

(c) {(Y-1)≥0 ∧ Z*X*(X**(Y-1))=x**y} (Y := Y-1; Z := Z*X) {Y≥0 ∧ Z*(X**Y)=x**y}   by a, b, (99)

(d) (Y>0 ∧ Z*(X**Y)=x**y ∧ odd(Y)) ⇒ ((Y-1)≥0 ∧ Z*X*(X**(Y-1))=x**y)   from Th

(e) (Y≥0 ∧ Z*(X**Y)=x**y) ⇒ (Y≥0 ∧ Z*(X**Y)=x**y)   from Th

(f) {Y>0 ∧ Z*(X**Y)=x**y ∧ odd(Y)} (Y := Y-1; Z := Z*X) {Y≥0 ∧ Z*(X**Y)=x**y}   by d, c, e, (102)

(g) {(Y div 2)≥0 ∧ Z*((X*X)**(Y div 2))=x**y} Y := Y **div** 2 {Y≥0 ∧ Z*((X*X)**Y)=x**y}   by (97)

(h) {Y≥0 ∧ Z*((X*X)**Y)=x**y} X := X*X {Y≥0 ∧ Z*(X**Y)=x**y}   by (97)

(i) {(Y div 2)≥0 ∧ Z*((X*X)**(Y div 2))=x**y} (Y := Y **div** 2; X := X*X){Y≥0 ∧ Z*(X**Y)=x**y}
by g, h, (99)

(j) (Y>0 ∧ Z*(X**Y)=x**y ∧ ¬odd(Y)) ⇒ ((Y div 2)≥0 ∧ Z*((X*X)**(Y div 2))=x**y)   from Th

(k) {Y>0 ∧ Z*(X**Y)=x**y ∧ ¬odd(Y)} (Y := Y **div** 2; X := X*X) {Y≥0 ∧ Z*(X**Y)=x**y}
by j, i, e, (102)

(l) {Y>0 ∧ Z*(X**Y)=x**y} (odd(Y) → (Y := Y-1; Z := Z*X) ◊ (Y := Y **div** 2; X := X*X))
{Y≥0 ∧ Z*(X**Y)=x**y}   by f, k, (100)

(m) (Y≥0 ∧ Z*(X**Y)=x**y ∧ Y<>0) ⇒ (Y>0 ∧ Z*(X**Y)=x**y)   from Th

(n) {Y≥0 ∧ Z*(X**Y)=x**y ∧ Y<>0} (odd(Y) → (Y := Y-1; Z := Z*X) ◊
(Y := Y **div** 2; X := X*X)) {Y≥0 ∧ Z*(X**Y)=x**y}   by m, l, e, (102)

(o) {Y≥0 ∧ Z*(X**Y)=x**y} (Y<>0 * (odd(Y) → (Y := Y-1; Z := Z*X) ◊
(Y := Y **div** 2; X := X*X))) {Y≥0 ∧ Z*(X**Y)=x**y ∧ ¬(Y<>0)}   by n, (101)

(p) {Y≥0 ∧ 1*(X**Y)=x**y} Z := 1 {Y≥0 ∧ Z*(X**Y)=x**y}   by (97)

(q) {Y≥0 ∧ 1*(X**Y)=x**y} (Z := 1; (Y<>0 * (odd(Y) → (Y := Y-1; Z := Z*X) ◊
(Y := Y **div** 2; X := X*X)))) {Y≥0 ∧ Z*(X**Y)=x**y ∧ ¬(Y<>0)}   by p, o, (99)

(r) (X=x ∧ Y=y≥0) ⇒ (Y≥0 ∧ 1*(X**Y)=x**y)   from Th

(s) (Y≥0 ∧ Z*(X**Y)=x**y ∧ ¬(Y<>0)) ⇒ (Z=x**y)   from Th

(t) {X=x ∧ Y=y≥0} (Z := 1; (Y<>0 * (odd(Y) → (Y: = Y-1; Z: = Z*X) ◊
(Y := Y **div** 2; X := X*X)))) {Z=x**y}   by r, q, s, (102)

■

7.2.4 Hoare's proof system $\mathbb{H}^{\circ}$ for proof outlines

If the deductive system $\mathbb{H}$ is useful for reasoning about Hoare logic, formal proofs using this proof system are totally unworkable (as shown be the level of details needed in example (104)). Proof outlines (81), as introduced by OWICKI [1975] and OWICKI & GRIES [1976a], offer a much more useful linear notation for proofs in which the program is given with assertions interleaved at cutpoints. A natural deduction programming logic of proof outlines was first presented in CONSTABLE & O'DONNELL [1978]. Hoare proof outline system $\mathbb{H}'$ below is due to BERGTRA & KLOP [1984], LIFSCHITZ[1984]. Further developments can be found in SCHNEIDER & ANDREWS [1986].

DEFINITION *Hoare proof outline system*

$C' : \mathbb{Com}'$ — *Asserted commands* (105)

$$
\begin{aligned}
C' ::=\ & \{P_1\}\ \mathbf{skip}\ \{P_2\} \\
\mid\ & \{P_1\}\ \mathbf{X := E}\ \{P_2\} \\
\mid\ & \{P_1\}\ \mathbf{X := ?}\ \{P_2\} \\
\mid\ & \{P_1\}\ (\mathbf{C'_1};\ \{P_2\}\ \mathbf{C'_2})\ \{P_3\} \\
\mid\ & \{P_1\}\ (\mathbf{B} \rightarrow \{P_2\}\mathbf{C'_1} \between \{P_3\}\ \mathbf{C'_2})\ \{P_4\} \\
\mid\ & \{P_1\}\ (\mathbf{B} * \{P_2\}\ \mathbf{C'}\ \{P_3\})\ \{P_4\} \\
\mid\ & \{P\}\ \mathbf{C'} \\
\mid\ & \mathbf{C'}\{P\}
\end{aligned}
$$

$\{P\}\ \mathbf{skip}\ \{P\}$ — *Skip axiom* (106)

$\{P[X \leftarrow E]\}\ \mathbf{X := E}\ \{P\}$ — *Assignment axiom* (107)

$\{P\}\ \mathbf{X := ?}\ \{\exists X.\ P\}$ — *Random assignment axiom* (108)

$$\frac{\{P_1\}\ \mathbf{C'_1}\ \{P_2\},\ \{P_2\}\ \mathbf{C'_2}\ \{P_3\}}{\{P_1\}\ (\mathbf{C'_1};\ \{P_2\}\ \mathbf{C'_2})\ \{P_3\}}$$ *Composition rule* (109)

$$\frac{\{P \wedge B\}\ \mathbf{C'_1}\ \{Q\},\ \{P \wedge \neg B\}\ \mathbf{C'_2}\ \{Q\}}{\{P\}\ (\mathbf{B} \rightarrow \{P \wedge B\}\mathbf{C'_1} \between \{P \wedge \neg B\}\mathbf{C'_2})\ \{Q\}}$$ *Conditional rule* (110)

$$\frac{\{P \wedge B\}\ \mathbf{C'}\ \{P\}}{\{P\}\ (\mathbf{B} * \{P \wedge B\}\mathbf{C'}\{P\})\ \{P \wedge \neg B\}}$$ *While rule* (111)

$$\frac{(P \Rightarrow P'),\ \{P'\}\ \mathbf{C'}\ \{Q\}}{\{P\}\ \{P'\}\ \mathbf{C'}\ \{Q\}}$$ *Left consequence rule* (112)

$$\frac{\{P\}\ \mathbf{C'}\ \{Q'\},\ (Q' \Rightarrow Q)}{\{P\}\ \mathbf{C'}\{Q'\}\ \{Q\}}$$ *Right consequence rule* (113)

Example *Proof outline of program (4) using* ℍ' (114)

```
{X = x ∧ Y = y ≥0}                by (112), (113)
    {Y ≥ 0 ∧ 1*(X**Y) = x**y}                                         by (107), (109)
      (Z := 1;
    {Y ≥ 0 ∧ Z*(X**Y) = x**y}                                         by (111)
      (Y <> 0 *
        {Y ≥ 0 ∧ Z*(X**Y) = x**y ∧ Y <> 0}                            by (112)
           {Y > 0 ∧ Z*(X**Y) = x**y}                                  by (110)
             (odd(Y) →
               {Y > 0 ∧ Z*(X**Y) = x**y ∧ odd(Y)}                     by (112)
                 {(Y-1) ≥ 0 ∧ Z*X*(X**(Y-1)) =x**y}                   by (107), (109)
                   (Y := Y - 1;
                 {Y ≥ 0 ∧ Z*X*(X**Y) = x**y}                          by (107)
                   Z := Z * X)
             ◊
               {Y > 0 ∧ Z*(X**Y) = x**y ∧ ¬odd(Y)}                    by (113)
                 {(Y div 2) ≥ 0 ∧ Z*((X*X)**(Y div 2)) = x**y}        by (107), (109)
                   (Y := Y div 2;
                 {Y ≥ 0 ∧ Z*((X*X)**Y) = x**y}                        by (107)
                   X := X * X)
             )
           {Y ≥ 0 ∧ Z*(X**Y) = x**y}
      ))
    {Y ≥ 0 ∧ Z*(X**Y) = x**y ∧ ¬(Y <> 0)}
{Z = x**y}
```

■

7.2.5 Syntactical rules of substitution

Up to now, we have used informal definitions of variables, bound variables and free variables occurring in a predicate or a command and of the substitution P[v ← T] of a term T for a variable v in a predicate P. We now give the fully formal definitions. This paragraph can be omitted and one can go on with paragraph § 7.3.

7.2.5.1 Variables appearing in a term, predicate, command or correctness formula

The set of variables appearing in a term, predicate, command or correctness formula is defined by structural induction as follows :

DEFINITION *Variables appearing in a formula* (115)

$Var(\mathrm{c}) = \emptyset$ (.1)

$Var(\mathrm{v}) = \{\mathrm{v}\}$ (.2)

$Var(\mathrm{f}(\mathrm{T}_1, \ldots, \mathrm{T}_{\#\mathrm{f}})) = \cup\{Var(\mathrm{T}_\mathrm{i}) : 1 \leq \mathrm{i} \leq \#\mathrm{f}\}$ (.3)

$Var((\mathrm{T}_1 = \mathrm{T}_2)) = Var(\mathrm{T}_1) \cup Var(\mathrm{T}_2)$ (.4)

$Var(\mathrm{r}(\mathrm{T}_1, \ldots, \mathrm{T}_{\#\mathrm{r}})) = \cup\{Var(\mathrm{T}_\mathrm{i}) : 1 \leq \mathrm{i} \leq \#\mathrm{r}\}$ (.5)

$Var(\neg \mathrm{P}) = Var(\mathrm{P})$ (.6)

$Var((\mathrm{P}_1 \wedge \mathrm{P}_2)) = Var((\mathrm{P}_1 \vee \mathrm{P}_2)) = Var((\mathrm{P}_1 \Rightarrow \mathrm{P}_2)) = Var(\mathrm{P}_1) \cup Var(\mathrm{P}_2)$ (.7)

$Var(\exists\, \mathrm{v}.\ \mathrm{P}) = Var(\forall\, \mathrm{v}.\ \mathrm{P}) = \{\mathrm{v}\} \cup Var(\mathrm{P})$ (.8)

$Var(\mathrm{skip}) = \emptyset$ (.9)

$Var(\mathrm{X} := \mathrm{E}) = \{\mathrm{X}\} \cup Var(\mathrm{E})$ (.10)

$Var(\mathrm{X} := ?) = \{\mathrm{X}\}$ (.11)

$Var((\mathrm{C}_1; \mathrm{C}_2)) = Var(\mathrm{C}_1) \cup Var(\mathrm{C}_2)$ (.12)

$Var((\mathrm{B} \rightarrow \mathrm{C}_1 \lozenge \mathrm{C}_2)) = Var(\mathrm{B}) \cup Var(\mathrm{C}_1) \cup Var(\mathrm{C}_2)$ (.13)

$Var((\mathrm{B} * \mathrm{C})) = Var(\mathrm{B}) \cup Var(\mathrm{C})$ (.14)

$Var(\{\mathrm{P}\}\ \mathrm{C}\ \{\mathrm{Q}\}) = Var(\mathrm{P}) \cup Var(\mathrm{C}) \cup Var(\mathrm{Q})$ (.15)

$Var(\mathrm{F}_1, \ldots, \mathrm{F}_\mathrm{n}) = Var(\mathrm{F}_1) \cup \ldots \cup Var(\mathrm{F}_\mathrm{n})$ (.16)

7.2.5.2 Bound and free variables appearing in a term, predicate, command or correctness formula

In the formula $\exists\, \mathrm{x}.\ ((+(\mathrm{x}, \mathrm{y}) = 0))$, variable x is "bounded" up by $\exists$ whereas y is sort of floating "free". The notions of *bound* and *free variable* can be made more precise as follows :

DEFINITION *Bound variables appearing in a formula* (116)

$Bound(\mathrm{A}) = \emptyset$ (.1)

$Bound(\neg \mathrm{P}) = Bound\ (\mathrm{P})$ (.2)

$Bound((\mathrm{P}_1 \wedge \mathrm{P}_2)) = Bound((\mathrm{P}_1 \vee \mathrm{P}_2)) = Bound((\mathrm{P}_1 \Rightarrow \mathrm{P}_2)) = Bound(\mathrm{P}_1) \cup Bound(\mathrm{P}_2)$ (.3)

$Bound(\exists\, \mathrm{v}.\ \mathrm{P}) = Bound(\forall\, \mathrm{v}.\ \mathrm{P}) = Bound(\mathrm{P}) \cup \{\mathrm{v}\}$ (.4)

$Bound(\mathrm{C}) = \emptyset$ (.5)

$Bound(\{\mathrm{P}\}\ \mathrm{C}\ \{\mathrm{Q}\}) = Bound(\mathrm{P}) \cup Bound(\mathrm{Q})$ (.6)

$Bound(\mathrm{F}_1, \ldots, \mathrm{F}_\mathrm{n}) = Bound(\mathrm{F}_1) \cup \ldots \cup Bound(\mathrm{F}_\mathrm{n})$ (.7)

DEFINITION *Free variables appearing in a formula* (117)

$Free(A)$	$= Var(A)$	(.1)
$Free(\neg P)$	$= Free(P)$	(.2)
$Free((P_1 \wedge P_2)) = Free((P_1 \vee P_2)) = Free((P_1 \Rightarrow P_2))$	$= Free(P_1) \cup Free(P_2)$	(.3)
$Free(\exists\, v.\, P) = Free(\forall\, v.\, P)$	$= Free(P) - \{v\}$	(.4)
$Free(C)$	$= Var(C)$	(.5)
$Free(\{P\}\ C\ \{Q\})$	$= Free(P) \cup Free(C) \cup Free(Q)$	(.6)
$Free(F_1, \dots, F_n)$	$= Free(F_1) \cup \dots \cup Free(F_n)$	(.7)

7.2.5.3 Formal definition of substitution of a term for a variable in a term or predicate

The substitution $P[v \leftarrow T]$ denotes the result of renaming bounded occurrences of variables in P so that none of them is v or belongs to $Var(T)$ and then replacing all free occurrences of variable v by term T. Substitution can be formally defined as follows :

DEFINITION *Substitution of a term for a variable* (118)

$v'[v \leftarrow T] = T$ if $v'=v$ (.1)

$\quad = v'$ if $v' \neq v$

$c[v \leftarrow T] = c$ (.2)

$f(T_1,\dots,T_{\#f})[v \leftarrow T] = f(T_1[v \leftarrow T],\dots,T_{\#f}[v \leftarrow T])$ (.3)

$(T_1 = T_2)[v \leftarrow T] = (T_1[v \leftarrow T] = T_2[v \leftarrow T])$ (.4)

$r(T_1,\dots,T_{\#r})[v \leftarrow T] = r(T_1[v \leftarrow T],\dots,T_{\#r}[v \leftarrow T])$ (.5)

$(\neg P)[v \leftarrow T] = \neg(P[v \leftarrow T])$ (.6)

$(P_1 \wedge P_2)[v \leftarrow T] = (P_1[v \leftarrow T] \wedge P_2[v \leftarrow T])$ (.7)

$(P_1 \vee P_2)[v \leftarrow T] = (P_1[v \leftarrow T] \vee P_2[v \leftarrow T])$ (.8)

$(P_1 \Rightarrow P_2)[v \leftarrow T] = (P_1[v \leftarrow T] \Rightarrow P_2[v \leftarrow T])$ (.9)

$(\exists v'.P)[v \leftarrow T]$

$\quad = \exists v'.P$ if $v' = v$ (.10)

$\quad = \exists v'.(P[v \leftarrow T])$ if $v' \neq v$ and $v' \notin Var(T)$

$\quad = \exists w.(P[v' \leftarrow w])[v \leftarrow T]$ where $w \notin \{v\} \cup Var(T) \cup Var(P)$,

$\qquad$ if $v' \neq v$ and $v' \in Var(T)$

$(\forall v'.P)[v \leftarrow T]$

$\quad = \forall v'.P$ if $v' = v$ (.11)

$\quad = \forall v'.(P[v \leftarrow T])$ if $v' \neq v$ and $v' \notin Var(T)$

$\quad = \forall w.(P[v' \leftarrow w])[v \leftarrow T]$ where $w \notin \{v\} \cup Var(T) \cup Var(P)$,

$\qquad$ if $v' \neq v$ and $v' \in Var(T)$

7.3 The semantics of Hoare logic

We now define the semantics of Hoare logic that is an interpretation $I : \mathbb{HL} \rightarrow \{tt, ff\}$ defining the truth of predicates and correctness formulae with respect to a relational semantics (19) of the programming language $\mathbb{C}\mathrm{om}$. This programming language semantics depends upon the semantics (also called a *model* or *interpretation*) $<D, V>$ of the basis $\sum$. By leaving this interpretation unspecified, we define a family of semantics of Hoare logic with respect to a family of possible relational semantics of the programming language.

7.3.1 Semantics of predicates and correctness formulae

A *model* or *interpretation* $<D, V>$ of the basis $\sum = <\mathbb{C}\mathrm{te}, \mathbb{F}\mathrm{un}, \mathbb{R}\mathrm{el}, \#>$ specifies the semantics of the common part of the programming and logical languages. It consists of a nonempty set D of data and a function V with domain $\mathbb{C}\mathrm{te} \cup \mathbb{F}\mathrm{un} \cup \mathbb{R}\mathrm{el}$ which define the intended meaning of constants, functions and relations :

DEFINITION *Interpretation of symbols* (119)

$V[\![c]\!] \in D$ (.1)

$V[\![f]\!] : D^{\#f} \rightarrow D$ (.2)

$V[\![r]\!] \subseteq D^{\#r}$ (.3)

Let us also recall that we have defined *states* (or *valuations*) s assigning a value $s(v) \in D$ to variables $v \in \mathbb{V}\mathrm{ar}$ (8) and $s[v \leftarrow d]$ for the state s' which agrees with s except that $s'(v) = d$:

DEFINITIONS *States and assignments*

$s \ : \quad S = \mathbb{V}\mathrm{ar} \rightarrow D$ *States* (120)

$s[v \leftarrow d](u) = (v = u \rightarrow d \lozenge s(u))$ *Assignment* (121)

These states have been used to remember the values assigned to programming variables during program execution. They will also be used to specify values for free variables in first order predicates. Remarkably, programming and logical variables can be handled the same way . This is not always possible for more complicated programming languages.

We now define the semantics or interpretations $\underline{T} = I[\![T]\!]$ of terms T, $\underline{P} = I[\![P]\!]$ of predicates P and $\underline{\{P\}C\{Q\}} = I[\![\{P\}C\{P\}]\!]$ of correctness formulae {P}C{Q} with respect to a given model $<D, V>$ (and a given state for terms and predicates):

DEFINITION *Interpretation of terms* (122)

$$I : \mathbb{Ter} \rightarrow (S \rightarrow D),\ \ \underline{\mathrm{T}} = I\llbracket \mathrm{T} \rrbracket$$

$$I\llbracket \mathrm{v} \rrbracket(\mathrm{s}) = \mathrm{s}(\mathrm{v}) \qquad (.1)$$

$$I\llbracket \mathrm{c} \rrbracket(\mathrm{s}) = V\llbracket \mathrm{c} \rrbracket \qquad (.2)$$

$$I\llbracket \mathrm{f}(\mathrm{T}_1, \ldots, \mathrm{T}_{\#\mathrm{f}}) \rrbracket(\mathrm{s}) = V\llbracket \mathrm{f} \rrbracket(I\llbracket \mathrm{T}_1 \rrbracket(\mathrm{s}), \ldots, I\llbracket \mathrm{T}_{\#\mathrm{f}} \rrbracket(\mathrm{s})) \qquad (.3)$$

DEFINITION *Interpretation of predicates* (123)

$$I : \mathbb{Pre} \rightarrow \mathbb{Ass},\quad \underline{\mathrm{P}} = I\llbracket \mathrm{P} \rrbracket$$

$$I\llbracket (\mathrm{T}_1 = \mathrm{T}_2) \rrbracket = \{\mathrm{s} \in S : \langle I\llbracket (\mathrm{T}_1) \rrbracket(\mathrm{s}), I\llbracket \mathrm{T}_2 \rrbracket(\mathrm{s}) \rangle \in \delta\} \qquad (.1)$$

$$I\llbracket \mathrm{r}(\mathrm{T}_1, \ldots, \mathrm{T}_{\#\mathrm{r}}) \rrbracket = \{\mathrm{s} \in S : \langle I\llbracket \mathrm{T}_1 \rrbracket(\mathrm{s}), \ldots, I\llbracket \mathrm{T}_{\#\mathrm{r}} \rrbracket(\mathrm{s}) \rangle \in V\llbracket \mathrm{r} \rrbracket\} \qquad (.2)$$

$$I\llbracket \neg \mathrm{P} \rrbracket = S - I\llbracket \mathrm{P} \rrbracket \qquad (.3)$$

$$I\llbracket (\mathrm{P}_1 \wedge \mathrm{P}_2) \rrbracket = I\llbracket \mathrm{P}_1 \rrbracket \cap I\llbracket \mathrm{P}_2 \rrbracket \qquad (.4)$$

$$I\llbracket (\mathrm{P}_1 \vee \mathrm{P}_2) \rrbracket = I\llbracket \mathrm{P}_1 \rrbracket \cup I\llbracket \mathrm{P}_2 \rrbracket \qquad (.5)$$

$$I\llbracket (\mathrm{P}_1 \Rightarrow \mathrm{P}_2) \rrbracket = (S - I\llbracket \mathrm{P}_1 \rrbracket) \cup I\llbracket \mathrm{P}_2 \rrbracket \qquad (.6)$$

$$I\llbracket \exists\, \mathrm{v}.\ \mathrm{P} \rrbracket = \{\mathrm{s} \in S : (\{\mathrm{s}[\mathrm{v} \leftarrow \mathrm{d}] : \mathrm{d} \in D\} \cap I\llbracket \mathrm{P} \rrbracket) \neq \emptyset\} \qquad (.7)$$

$$I\llbracket \forall\, \mathrm{v}.\ \mathrm{P} \rrbracket = \{\mathrm{s} \in S : \{\mathrm{s}[\mathrm{v} \leftarrow \mathrm{d}] : \mathrm{d} \in D\} \subseteq I\llbracket \mathrm{P} \rrbracket\} \qquad (.8)$$

DEFINITION *Interpretation of correctness formulae* (124)

$$I : \mathbb{Hcf} \rightarrow \{\mathrm{tt}, \mathrm{ff}\}$$

$$I\llbracket \{\mathrm{P}\}\ \mathrm{C}\ \{\mathrm{Q}\} \rrbracket = \underline{\{I\llbracket \mathrm{P} \rrbracket\}\mathrm{C}\{I\llbracket \mathrm{Q} \rrbracket\}} \quad \text{where} \quad \underline{\{\mathrm{p}\}\mathrm{C}\{\mathrm{q}\}} = (\mathrm{p} \rceil \underline{\mathrm{C}}) \subseteq (S \times \mathrm{q})$$

Observe that the truth or falsity of a formula {P}C{Q} just depends upon the model $\langle D, V \rangle$ since the semantics $\underline{\mathrm{C}}$ of command C (19) itself depends only on the semantics $\underline{\mathrm{E}}$ of expressions E and $\underline{\mathrm{B}}$ of Boolean expressions B which is the same as the semantics of terms (with no logical variables) and predicates (with no quantifiers and logical variables). An interpretation of {P}C{Q} different from (124) is investigated in ANDRÉKA & NÉMETI [1978], GERGELY & ÚRY [1978], ANDRÉKA, NÉMETI & SAIN [1979] [1981] [1983], CSIRMAZ [1981a] [1981b], HORTALÁ-GONZÁLEZ & RODRÍGUEZ-ARTALEJO [1985], NÉMETI [1980], RODRÍGUEZ-ARTALEJO [1985] using a nonstandard transfinite definition of execution traces.

DEFINITION *Interpretation of formulae* (125)

$$I : \mathbb{HL} \rightarrow \{\mathrm{tt}, \mathrm{ff}\}$$

$$I\llbracket \mathrm{F} \rrbracket = (\mathrm{F} \in \mathbb{Pre} \rightarrow I\llbracket \mathrm{F} \rrbracket = S \lozenge I\llbracket \mathrm{F} \rrbracket)$$

(The function I is polymorphic, so that when $\mathrm{P} \in \mathbb{Pre}$, and depending upon the context ,we have either $I\llbracket \mathrm{P} \rrbracket \in \mathcal{P}(S)$ (by (123)) or $I\llbracket \mathrm{P} \rrbracket \in \{\mathrm{tt}, \mathrm{ff}\}$ (by (125))).

Example *Proof of a formula with two different interpretations* (126)

Let us consider the basis <{0, 1}, {+}, ∅, #> with #(+) = 2. Then H = {X = 1} (¬(X = 0) * X := X + 1) {X = 0} is a formula of $\mathbb{HL}$.

A first interpretation would be <$\mathbb{N}$, *V*> with $V[\![0]\!] = 0_{\mathbb{N}}$, $V[\![1]\!] = 1_{\mathbb{N}}$, $V[\![+]\!](x, y) = x +_{\mathbb{N}} y$. With this interpretation formula H is semantically true ($I[\![H]\!] = tt$) because execution of program (¬(X = 0) * X := X + 1) starting in a state s such that s(X) = 1 will never terminate.

A second interpretation would be <{$0_{\mathbb{N}}$, $1_{\mathbb{N}}$}, *V*> with $V[\![0]\!] = 0_{\mathbb{N}}$, $V[\![1]\!] = 1_{\mathbb{N}}$, $V[\![+]\!](x, y) = (x +_{\mathbb{N}} y)\ mod\ 2_{\mathbb{N}}$. With this interpretation formula F is semantically true because execution of program (¬(X = 0) * X := X + 1) always terminates in a state s such that s(X) = $0_{\mathbb{N}}$.

Formula H can be proved to be formally correct from tautologies Th = {((true ∧ ¬(X = 0)) ⇒ true), (true ⇒ true), ((X = 1) ⇒ true), ((true ∧ ¬¬(X = 0)) ⇒ (X = 0))} where true denotes truth e.g.. (x = x), as follows:

(a)	(true ∧ ¬(X = 0)) ⇒ true	by Th
(b)	{true} X := X + 1 {true}	by (97)
(c)	(true) ⇒ (true)	by Th
(d)	{true ∧ ¬(X = 0)} X := X + 1 {true}	by a, b, c, (102)
(e)	{true} (¬(X = 0) * X := X + 1) {true ∧ ¬¬(X = 0)}	by d, (101)
(f)	(X = 1) ⇒ true	by Th
(g)	(true ∧ ¬¬(X = 0)) ⇒ (X = 0)	by Th
(h)	{X = 1} (¬(X = 0) * X := X + 1) {X = 0}	by f, e, g, (102)

■

7.3.2 Semantics of substitution

In (118) we have defined the substitution of a term for a variable in a predicate which is used in the assignment axiom schema (97). To prove that this axiom schema is sound we shall need a semantical characterization of substitution. This paragraph § 7.3.2 can be omitted on first reading.

Informally substitution commutes with interpretation. More precisely, the interpretation $\underline{T[v \leftarrow T']}(s)$ of term T where T' is substituted for v in state s is the interpretation $\underline{T}(s[v \leftarrow \underline{T'}(s)])$ of term T in state s' =s $[v \leftarrow \underline{T'}(s)]$ which agrees with s except that the value s'(v) of variable v is the interpretation of term T' in state s :

LEMMA *Semantics of substitution of a term for a variable in a term* (127)

$\underline{T[v \leftarrow T']}(s) = \underline{T}(s[v \leftarrow \underline{T'}(s)])$

Proof

By structural induction on the syntax of terms :

- $\underline{v[v \leftarrow T']}(s) = \underline{T'}(s)$ [by (118.1)] $= s[v \leftarrow \underline{T'}(s)](v)$ [by (121)] $= \underline{v}(s[v \leftarrow \underline{T'}(s)])$ [by (122.1)],
- when $v' \neq v$, $\underline{v'[v \leftarrow T']}(s) = \underline{v'}(s)$ [by (118.1)] $= s(v')$ [by (122.1)] $= s[v \leftarrow \underline{T'}(s)](v')$ [by (121)] $= \underline{v'}(s[v \leftarrow \underline{T'}(s)])$ [by (122.1)],
- $\underline{c[v \leftarrow T']}(s) = \underline{c}(s)$ [by (118.2)] $= V[\![c]\!]$ [by (122.2)] $= \underline{c}(s[v \leftarrow \underline{T'}(s)])$ [by (122.2)],
- $\underline{f(T_1 \ldots, T_{\#f})[v \leftarrow T']}(s) = \underline{f(T_1[v \leftarrow T'], \ldots, T_{\#f}[v \leftarrow T'])}(s)$ [by (118.3)] $= V[\![f]\!](\underline{T_1[v \leftarrow T']}(s), \ldots, \underline{T_{\#f}[v \leftarrow T']}(s))$ [by (122.3)] $= V[\![f]\!](\underline{T_1}(s[v \leftarrow \underline{T'}(s)]), \ldots, \underline{T_{\#f}}(s[v \leftarrow \underline{T'}(s)]))$ [by induction hypothesis (127)] $= \underline{f(T_1, \ldots, T_{\#f})}(s[v \leftarrow \underline{T'}(s)])$ [by (122.3)]. ■

The same way, substitution of a term for a variable in a predicate can be semantically characterized by the following :

LEMMA *Semantics of substitution of a term for a variable in a predicate* (128)

$$\underline{P[v \leftarrow T]} = \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{P}\}$$

Proof

The proof is (almost) by structural induction on the syntax of predicates. The only difficulty is for $(\forall v'. P)[v \leftarrow T]$ when $v' \neq v$ and $v' \in Var(T)$ because $\forall w. P[v' \leftarrow w]$ is not a syntactic component of $\forall v'. P$. However they have the same shapes and more variables of T appear in $\forall v'. P$ than in $\forall w. P[v' \leftarrow w]$. Thus we define the height $\eta(P,T)$ of a predicate P with respect to a term T by structural induction as follows: $\eta(A, T) = 0$; $\eta(\neg P, T) = 1 + \eta(P, T)$; $\eta(P_1 \wedge P_2, T) = \eta(P_1 \vee P_2, T) = \eta(P_1 \Rightarrow P_2, T) = 1 + \max(\eta(P_1, T), \eta(P_2, T))$; $\eta(\exists x. P, T) = \eta(\forall x. P, T) = 1 + \eta(P, T) + |Var(P) \cap Var(T)|$. For a given term T, the proof is by induction on the height $\eta(P, T)$ of P. This is long but not difficult. Therefore we only treat few typical cases :

- $\underline{(T_1 = T_2)[v \leftarrow T]} = \underline{(T_1[v \leftarrow T] = T_2[v \leftarrow T])}$ [by (118.4)] $= \{s \in S : \underline{T_1[v \leftarrow T]}(s) = \underline{T_2[v \leftarrow T]}(s)\}$ [by (123.1) and definition of δ] $= \{s \in S : \underline{T_1}(s[v \leftarrow \underline{T}(s)]) = \underline{T_2}(s[v \leftarrow \underline{T}(s)])\}$ [by (127)] $= \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{(T_1 = T_2)}\}$ [since $(\underline{T_1}(s[v \leftarrow \underline{T}(s)]) = \underline{T_2}(s[v \leftarrow \underline{T}(s)])) \Leftrightarrow (s[v \leftarrow \underline{T}(s)] \in \{s' : \underline{T_1}(s') = \underline{T_2}(s')\}) \Leftrightarrow (s[v \leftarrow \underline{T}(s)] \in \underline{(T_1 = T_2)})$, by (123.1)].
- $\underline{(P_1 \Rightarrow P_2)[v \leftarrow T]} = \underline{(P_1[v \leftarrow T] \Rightarrow P_2[v \leftarrow T])}$ [by (118.9)] $= (S - \underline{P_1[v \leftarrow T]}) \cup \underline{P_2[v \leftarrow T]}$ [by (123.6)] $= (S - \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{P_1}\}) \cup \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{P_2}\}$ [by induction hypothesis (128)] $= \{s \in S : s[v \leftarrow \underline{T}(s)] \notin \underline{P_1}\} \cup \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{P_2}\} = \{s \in S : s[v \leftarrow \underline{T}(s)] \in (S - \underline{P_1}) \cup \underline{P_2}\} = \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{(P_1 \Rightarrow P_2)}\}$ [by (123.6)].
- $\underline{(\forall v. P)[v \leftarrow T]} = \underline{\forall v. P}$ [by (118.11)] $= \{s \in S : \forall d \in D.\ s[v \leftarrow d] \in \underline{P}\}$ [by (123.8)] $= \{s \in S : \forall d \in D.\ (s[v \leftarrow \underline{T}(s)])[v \leftarrow d] \in \underline{P}\}$ [since $(s[v \leftarrow d'])[v \leftarrow d] = s[v \leftarrow d]$ by (121)] $= \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{\forall v. P}\}$ [by (123.8)].

- if $v' \neq v$ and $v' \notin Var(T)$ then $\underline{(\forall\, v'.\ P)[v \leftarrow T]} = \underline{\forall\, v'.\ (P[v \leftarrow T])}$ [by (118.11)] $= \{s \in S : \{s[v' \leftarrow d] : d \in D\} \subseteq \underline{P[v \leftarrow T]}\}$ [by (123.8)] $= \{s \in S : \{s[v' \leftarrow d] : d \in D\} \subseteq \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{P}\}\}$ [by induction hypothesis (128) since $\eta(P, T) < \eta(\forall\, v'.\ P, T)$] $= \{s \in S : \forall\, d \in D.\ ((s[v' \leftarrow d])[v \leftarrow \underline{T}(s)] \in \underline{P})\} = \{s \in S : \forall\, d \in D.\ ((s[v \leftarrow \underline{T}(s)])[v' \leftarrow d] \in \underline{P})\}$ [by (121) since $v \neq v'$] $= \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{(\forall\, v'.\ P)}\}$ [by (123.8)].
- if $v' \neq v$ and $v' \in Var(T)$ then $\underline{(\forall\, v'.\ P)[v \leftarrow T]} = \underline{\forall\, w.\ (P[v' \leftarrow w])[v \leftarrow T]}$ [by (118.11)] $= \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{\forall\, w.\ (P[v' \leftarrow w])}\}$ [by induction hypothesis (128) since $v' \in Var(T)$ and $w \notin \{v\} \cup Var(T) \cup Var(P)$ imply $Var((\forall\, v'.\ P)) \cap Var(T) = (Var(\forall\, w.\ (P[v' \leftarrow w])) \cap Var(T)) \cup \{v'\}$ whence $\eta(\forall\, w.\ (P[v' \leftarrow w]), T) < \eta(\forall\, v'.\ P, T)$ since $\eta(P[v' \leftarrow w], T) = \eta(P, T)$ and $v' \notin Var(\forall\, w.\ (P[v' \leftarrow w])) \cap Var(T)$] $= \{s \in S : s[v \leftarrow \underline{T}(s)] \in \{s' \in S : \{s'[w \leftarrow d] : d \in D\} \subseteq \underline{P[v' \leftarrow w]}\}\}$ [by (123.8)] $= \{s \in S : \forall\, d \in D.\ s[v \leftarrow \underline{T}(s)][w \leftarrow d] \in \underline{P[v' \leftarrow w]}\}\} = \{s \in S : \forall\, d \in D.\ s[v \leftarrow \underline{T}(s)][w \leftarrow d] \in \{s' \in S : s'[v' \leftarrow \underline{w}(s')] \in \underline{P}\}\}$ [by induction hypothesis (128) since $\eta(P[v' \leftarrow w], T) < \eta((\forall\, v'.\ P), T)$ because $w \notin Var(P) \cup Var(T)$ whence $\eta(P[v' \leftarrow w], T) \leq \eta(P, T)$] $= \{s \in S : \forall\, d \in D.\ s[v \leftarrow \underline{T}(s)][w \leftarrow d][v' \leftarrow \underline{w}(s[v \leftarrow \underline{T}(s)][w \leftarrow d])] \in \underline{P}\} = \{s \in S : \forall\, d \in D.\ s[v \leftarrow \underline{T}(s)][w \leftarrow d][v' \leftarrow d] \in \underline{P}\}$ [by (122.1) and (121)] $= \{s \in S : \forall\, d \in D.\ s[v \leftarrow \underline{T}(s)][v' \leftarrow d][w \leftarrow d] \in \underline{P}\}$ [by (121) since $w \neq v'$] $= \{s \in S : \forall\, d \in D.\ s[v \leftarrow \underline{T}(s)][v' \leftarrow d] \in \underline{P}\}$ [since $w \notin Var(P)$] $= \{s \in S : s[v \leftarrow \underline{T}(s)] \in \underline{(\forall\, v'.\ P)}\}$ [by (123.8)]. ■

7.4 The link between syntax and semantics: soundness and completeness issues in Hoare logic

In paragraph § 7.2 we have defined the language $\mathbb{HL}$ of Hoare logic and then the provability $\vdash_{Th \cup \mathbb{H}}$ F of formulae F. In paragraph § 7.3 we have defined the semantics of $\mathbb{HL}$ that is the truth $\underline{F}$ of formulae F. We now investigate the relationship between these two definitions that is the soundness of provability (is a provable formula always true ?) and the completeness of provability (is a true formula always provable ?). The deductive system $\mathbb{H}$ is sound for $\mathbb{HL}$ (provided all theorems in Th are true). The question of completeness is more subtle because this depends upon which class of interpretations *I* (induced by the semantics (*D*, *V*) of the basis $\sum$) is considered. Hence we can only prove *relative completeness*, a notion first delineated by WAND [1978] and COOK [1978].

7.4.1 Soundness of Hoare logic

Hoare deductive system $\mathbb{H}$ is *sound*: if we have proved {P} C {Q} from Th using $\mathbb{H}$ then C is partially correct with respect to specification <$\underline{P}$, $\underline{Q}$> (assuming that all T in Th are true) :

THEOREM COOK [1978] *Soundness of Hoare logic* (129)

$(\forall T \in Th.\ I[\![T]\!] = tt) \wedge (\vdash_{Th \cup \mathbb{H}} \{P\}\ C\ \{Q\}) \Rightarrow \underline{\{P\}\ C\ \{Q\}}$

The proof of (129) shows that Hoare's formal proof system $\mathbb{H}$ simply consists in applying theorem (76) within the framework of the restricted logical language $\mathbb{HL}$. This proof can be done by a theorem prover (SOKOLOWSKI [1987]).

Proof

Assuming $\forall T \in Th.\ I[\![T]\!] = tt$ and given a proof $H_0, \ldots, H_n$ of {P} C {Q}, we prove by induction that for all i = 0, …, n we have $I[\![H_i]\!] = tt$, so that in particular $\underline{\{P\}\ C\ \{Q\}}$ is true:

- If $H_i \in Th$, then by hypothesis $I[\![H_i]\!] = tt$.
- If H_i is an axiom of $\mathbb{H}$, then three cases have to be considered for any given $P \in \mathbb{P}\mathrm{re}$:

- For a skip axiom (96), $\underline{\{P\}\ skip\ \{P\}}$ obviously holds by (76.1),
- For a backward assignment axiom (97), we have $\underline{\{P[X \leftarrow E]\}\ X := E\ \{P\}} = \underline{\{}\{s \in S : s[v \leftarrow \underline{E}(s)] \in \underline{P}\}\underline{\}\ X := E\ \{P\}}$ [by (128)] which is true by (76.2),
- For a forward assignment axiom (103), we have $\underline{\exists X'.\ P[X \leftarrow X'] \wedge X = E[X \leftarrow X']} = \{s \in S : (\{s[X' \leftarrow d] : d \in D\} \cap \underline{P[X \leftarrow X'] \wedge X = E[X \leftarrow X']}) \neq \emptyset\}$ [by (124) and (123.7)] $= \{s \in S : (\{s[X' \leftarrow d] : d \in D\} \cap \{s \in S : s[X \leftarrow s(X')] \in \underline{P}\} \cap \{s \in S : s(X) = \underline{E}(s[X \leftarrow s(X')])\}) \neq \emptyset\}$ [by (123.4), (128), (122.1) and (127)] $= \{s \in S : \exists d \in D.\ s[X \leftarrow d] \in \underline{P} \wedge s(X) = \underline{E}(s[X \leftarrow d])\}$ whence $\underline{\{P\}\ X := E\ \{\exists X'.\ P[X \leftarrow X'] \wedge X = E[X \leftarrow X']\}} = \underline{P} \rceil \underline{X := E} \subseteq S \times \{s \in S : \exists d \in D.\ s[X \leftarrow d] \in \underline{P} \wedge s(X) = \underline{E}(s[X \leftarrow d])\}$ [by (124)] $= \{s[X \leftarrow \underline{E}(s)] : s \in \underline{P}\} \subseteq \{s \in S : \exists d \in D.\ s[X \leftarrow d] \in \underline{P} \wedge s(X) = \underline{E}(s[X \leftarrow d])\}$ [by (19.2)] $= \forall s \in \underline{P}.\ \exists d \in D.\ s[X \leftarrow \underline{E}(s)][X \leftarrow d] \in \underline{P} \wedge s[X \leftarrow \underline{E}(s)](X) = \underline{E}(s[X \leftarrow \underline{E}(s)][X \leftarrow d]) = \forall s \in \underline{P}.\ \exists d \in D.\ s[X \leftarrow d] \in \underline{P} \wedge \underline{E}(s) = \underline{E}(s[X \leftarrow d])$ which is obviously true by choosing $d = \underline{E}(s)$.
- For a random assignment axiom we have $\underline{\{P\}\ X := ?\ \{\exists X.\ P\}} = \underline{\{P\}\ X := ?\ \{}\{s \in S : (\{s[X \leftarrow d] : d \in D\} \cap \underline{P}) \neq \emptyset\}\underline{\}}$ [by (123.7)] $= \underline{\{P\}\ X := ?\ \{}\{s \in S : \exists d \in D.\ s[X \leftarrow d] \in \underline{P}\}\underline{\}} = \underline{\{P\}\ X := ?\ \{}\{s'[X \leftarrow d'] : d' \in D \wedge s' \in \underline{P}\}\underline{\}}$ [by (121) since we let $s' = s[X \leftarrow d]$ and $d' = s(X)$] = tt by (76.3).

- If H_i follows from an inference of $\mathbb{H}$, then four cases have to be considered for any given P_1, P_2, P, P', Q, $Q' \in \mathbb{P}\mathrm{re}$:

- For a composition inference, assuming $\underline{\{P_1\}\ C_1\ \{P_2\}}$ and $\underline{\{P_2\}\ C_2\ \{P_3\}}$ by induction hypothesis, we have $\underline{\{P_1\}\ (C_1;\ C_2)\ \{P_3\}}$ = tt by (76.4),
- For a conditional inference (100), assuming $\underline{\{P \wedge B\}\ C_1\ \{Q\}}$ and $\underline{\{P \wedge \neg B\}\ C_2\ \{Q\}}$ by induction hypothesis, we have $\underline{\{P} \cap \underline{B\}\ C_1\ \{Q\}}$ and $\underline{\{P} \cap \neg\underline{B\}\ C_2\ \{Q\}}$ by (123.4) and (123.3) hence $\underline{\{P\}\ (B \to C_1 \lozenge C_2)\ \{Q\}}$ is true by (76.5),
- For a while inference (101), assuming $\underline{\{P \wedge B\}\ C\ \{P\}}$, we have $\underline{\{P} \cap \underline{B\}\ C\ \{P\}}$ [by (123.4)] = $\underline{\{P\}\ (B\ *\ C)\ \{P} \cap \neg\underline{B\}}$ [by (76.6)] = $\underline{\{P\}\ (B\ *\ C)\ \{P \wedge \neg B\}}$ [by (123.4) and (123.3)],
- For a consequence inference (102), assuming $\underline{(P \Rightarrow P')}$, $\underline{\{P'\}\ C\ \{Q'\}}$ and $\underline{(Q' \Rightarrow Q)}$ by induction hypothesis, we have $\underline{P} \subseteq \underline{P'}$ and $\underline{Q'} \subseteq \underline{Q}$ by (123.6) hence $\underline{\{P\}\ C\ \{Q\}}$ by (76.7). ■

7.4.2 Relative completeness of Hoare logic

7.4.2.1 Completeness and incompleteness issues for Hoare logic

Having defined the syntax $\mathbb{C}\mathrm{om}$ of programs [see (1) with later complements (91) for expressions and (92) for Boolean expressions] parametrized by a basis <$\mathbb{C}\mathrm{te}$, $\mathbb{F}\mathrm{un}$, $\mathbb{R}\mathrm{el}$, #> [(84), (85)], we have approached Hoare's ideas on formal definition of the partial correctness of programs $C \in \mathbb{C}\mathrm{om}$, in essentially two different ways:

- From the semantic point of view of computer scientists (corresponding to the point of view of mathematicians using normal everyday set-theoretic apparatus where objects are thought of in terms of their representation, e.g. sets are understood as collections of objects), we have assumed that the semantics <*D*, *V*> of the basis <$\mathbb{C}\mathrm{te}$, $\mathbb{F}\mathrm{un}$, $\mathbb{R}\mathrm{el}$, #> is given (119), from which we have defined the sets *S* of states (120), the set $\mathbb{A}\mathrm{ss} = \mathcal{P}(S)$ of assertions as well as the semantics $\underline{E} = I[\![E]\!]$ (122) of expressions E (91) and the semantics $\underline{B} = I[\![B]\!]$ (123) of Boolean expressions B (92), whence the operational semantics $op[\![C]\!]$ (13) and then the relational semantics $\underline{C} = I[\![C]\!]$ [(18), (19)] of programs $C \in \mathbb{C}\mathrm{om}$. This leads to the introduction of Hoare's partial correctness specifications:

$$\mathbb{HS}(I) = \mathbb{A}\mathrm{ss} \times \mathbb{C}\mathrm{om} \times \mathbb{A}\mathrm{ss} \qquad \textit{Hoare's specifications} \qquad (130)$$

where the dependence upon the interpretation *I* is a shorthand for denoting the dependence upon the basis <$\mathbb{C}\mathrm{te}$, $\mathbb{F}\mathrm{un}$, $\mathbb{R}\mathrm{el}$, #> and its semantics <*D*, *V*>. A triple <p, C, q> of $\mathbb{HS}(I)$ should be understood as specifying that p is the input specification and q is the output specification of program C where p and q are sets of states specifying all possible combinations of values of the variables. The set $\mathbb{HS}(I)$ has been partitioned

(22) into a subset representing truth (i.e. which programs C are partially correct with respect to which specifications):

$$\mathbb{HS}(I)_{tt} = \{<p, C, q> : p \in \mathbb{Ass} \wedge C \in \mathbb{Com} \wedge q \in \mathbb{Ass} \wedge \underline{\{p\}C\{q\}} = tt\} \quad \textit{Hoare's valid specifications} \tag{131}$$

and a subset representing falsity (i.e. all we know not to be true about the partial correctness of programs):

$$\mathbb{HS}(I)_{ff} = \{<p, C, q> : p \in \mathbb{Ass} \wedge C \in \mathbb{Com} \wedge q \in \mathbb{Ass} \wedge \underline{\{p\}C\{q\}} = ff\} \quad \textit{Hoare's invalid specifications} \tag{132}$$

Within this framework, we have explained several (equivalent) partial correctness proof methods by decomposition of the proof that $\underline{\{p\}C\{q\}} = tt$ into simpler elementary proofs based upon stepwise induction (45) as in NAUR [1966] and FLOYD [1967a] or compositional induction (49), (76) as in HOARE [1969]. In this development, we have already used a formalized language with only vestigial traces of English but without any linguistic constrain (for example we felt free to use second order sentences [such as (19.8.b)] and regretfully to misuse English).

- Then we have defined the language $\mathbb{Hcf}$ (89) of Hoare logic for describing partial correctness (or incorrectness) of programs. By choosing the particular language $\mathbb{Hcf}$, we deliberately limit our means of expression. Hence we have adopted the syntactical point of view of computer scientists (corresponding to the use of specific formal languages by logicians, where objects are thought of in terms of their denotation and rewriting manipulations). In this framework, we can define veracity (and falsehood) of partial correctness formulae by two disjoint sublanguages $\mathbb{Hcf}_{tt} \subseteq \mathbb{Hcf}$ (and $\mathbb{Hcf}_{ff} \subseteq \mathbb{Hcf}$). We have used two methods to specify the sublanguage $\mathbb{Hcf}_{tt}$, one defining "truth" and the other "provability":
- Following the semantic development of logic (also called "model theory", BARWISE [1977], part A), we have defined the semantics of $\mathbb{Hcf}$, by means of an interpretation $I : \mathbb{Hcf} \rightarrow \{tt,ff\}$ (124) which depends upon the semantics $<D, V>$ of the basis $<\mathbb{Cte}, \mathbb{Fun}, \mathbb{Rel}, \#>$ and induces a partition of the language $\mathbb{Hcf}$ into true and false sentences:

$$\mathbb{Hcf}_{tt}(I) = \{H \in \mathbb{Hcf} : I[\![H]\!] = tt\} \quad \textit{Hoare's valid correctness formulae} \tag{133}$$

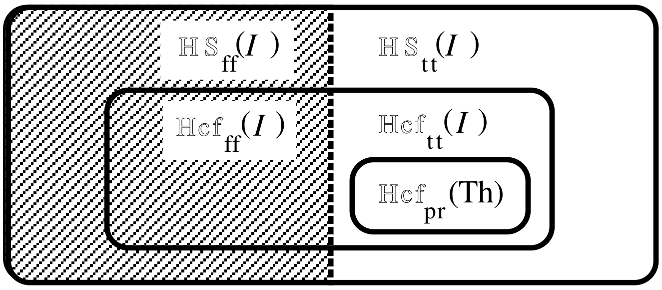
$$\mathbb{Hcf}_{ff}(I) = \{H \in \mathbb{Hcf} : I[\![H]\!] = ff\} \quad \textit{Hoare's invalid correctness formulae} \tag{134}$$

(Instead of writing $I[\![H]\!] = tt$, logicians would use the *satisfaction relation* $I \models H$. The set $\mathbb{Hcf}_{tt}(I) = \{H \in \mathbb{Hcf} : I \models H\}$ of sentences of $\mathbb{Hcf}$ true in I would be called the *theory* of I).

• Following the calculative development of logic (also called "proof theory", BARWISE [1977], part D), we have presented Hoare deductive system $\mathbb{H}$ (93) with its axiom schemata (96) to (98) and rules of inference (99) to (102) and defined provability $\vdash_{\mathrm{Th} \cup \mathbb{H}}$ H of formulae $\mathrm{H} \in \mathbb{H}$ with respect to a set Th of postulates (95). Therefore we get the subset of provable sentences (contrary to the ordinary situation in logic we do not define refutable sentences since we are not interested in partial incorrectness ¬H):

$$\mathbb{Hcf}_{pr}(\mathrm{Th}) = \{\mathrm{H} \in \mathbb{Hcf} : \vdash_{\mathrm{Th} \cup \mathbb{H}} \mathrm{H}\}. \qquad \textit{Hoare's provable correctness formulae} \qquad (135)$$

In order to compare syntactical objects Th, $\mathbb{Hcf}_{pr}(\mathrm{Th})$, $\mathbb{Hcf}_{tt}(I)$ with semantic objects I, $\mathbb{HS}_{tt}(I)$, we can map Hoare correctness formulae H = {P}C{Q} into triples $\gamma(\mathrm{H}) = \langle I[\![\mathrm{P}]\!], \mathrm{C}, I[\![\mathrm{Q}]\!]\rangle$ of $\mathbb{HS}(I)$ and assume that the interpretation I is a model of postulates Th, that is to say that every formula t of Th is true for interpretation I: $\forall\, t \in \mathrm{Th}.\ I[\![t]\!] = tt$. Then we get the following picture of inclusion between these sets (up to the correspondence γ):



The fact that $\mathbb{Hcf}_{pr}(\mathrm{Th}) \subseteq \mathbb{Hcf}_{tt}(I)$ follows from the soundness theorem (129): every provable formula is true. The inclusions $\gamma(\mathbb{Hcf}_{tt}(I)) \subseteq \mathbb{HS}(I)_{tt}$ and $\gamma(\mathbb{Hcf}_{ff}(I)) \subseteq \mathbb{HS}(I)_{ff}$ follow from the interpretation (123) of correctness formulae. Now, the incompleteness / completeness question is whether these inclusions are strict or not :

- $\forall I.\ \mathbb{HS}(I)_{tt} \subseteq \gamma(\mathbb{Hcf}_{tt}(I))$: is every true fact about the partial correctness of programs expressible in the restricted language $\mathbb{Hcf}$?
- $\forall I.\ \mathbb{Hcf}_{tt}(I) \subseteq \mathbb{Hcf}_{pr}(\mathrm{Th})$: is every true formula of $\mathbb{Hcf}$ provable by the deductive system $\mathbb{H}$?

In both cases, and without additional hypotheses, the answer is no. Intuitively, the origin of these incompleteness problems is that there exist programs $\mathrm{C} \in \mathbb{Com}$ constructing (input-output) relationships between objects of the domain D of data that, under the assumption that the programming language $\mathbb{Com}$ and first-order formulae $\mathbb{Pre}$ must have the same signature <$\mathbb{Cte}$, $\mathbb{Fun}$, $\mathbb{Rel}$, #>, cannot be described by a predicate of $\mathbb{Pre}$. To illustrate the phenomenon, we shall use "abacus arithmetic" a limited version of usual arithmetic with 0, Su(x) = x + 1 and Pr(x) = x - 1 (with 0 - 1 = 0) as only non-logical symbols.

7.4.2.2 Abacus arithmetic

The purpose of this section is to show that addition cannot be defined by some formula of abacus arithmetic $\mathbb{Pre}_A$ defined by (88) where the basis is A = <$\mathbb{Cte}$, $\mathbb{Fun}$, $\mathbb{Rel}$, #> with $\mathbb{Cte}$ = {0}, $\mathbb{Fun}$ = {Pr, Su}, $\mathbb{Rel}$ = ø, #(Pr) = #(Su) = 1 for interpretation I_A defined by (123) where $D = \mathbb{N}$, $V[\![0]\!] = 0$, $V[\![\mathrm{Pr}]\!](x) = (x = 0 \rightarrow 0 \lozenge x -_{\mathbb{N}} 1)$ and $V[\![\mathrm{Su}]\!](x) = x +_{\mathbb{N}} 1$. It is also shown that the *theory* of abacus arithmetic (i.e. $\mathbb{Pre}_{A,tt}(I_A) = \{P \in \mathbb{Pre}_A : I[\![P]\!] = tt\}$) is *decidable*, that is there is an algorithm which given $P \in \mathbb{Pre}_A$ will always terminate and answer “tt” if $P \in \mathbb{Pre}_{A,tt}(I_A)$ and answer “ff” if this is not true. We will also show that abacus arithmetic has nonstandard interpretations.

7.4.2.2.1 Inexpressibility of addition in abacus arithmetic

LEMMA ENDERTON [1972] *Disjunctive normal form* (135)

A quantifier-free predicate P and its negation ¬P have equivalent *disjunctive normal forms* $P^{\vee\wedge}$ and $P^{\underline{\vee\wedge}}$ (i.e. $I[\![P]\!] = I[\![P^{\vee\wedge}]\!]$ and $I[\![\neg P]\!] = I[\![P^{\underline{\vee\wedge}}]\!]$ for all interpretations I) such that:

$$P^{\vee\wedge} = \bigvee_{i = 1, m} \bigwedge_{j = 1, n} P_{ij} \qquad\qquad P^{\underline{\vee\wedge}} = \bigvee_{k = 1, p} \bigwedge_{l = 1, q} \underline{P}_{kl}$$

where the P_{ij} and $\underline{P}_{kl}$ are atomic formulae or negations of atomic formulae.

Proof

By induction on the syntactical structure of P:

- If P is an atomic formula A then $P^{\vee\wedge} = \bigvee_{i=1, 1} \bigwedge_{j = 1, 1} A$ and $P^{\underline{\vee\wedge}} = \bigvee_{k = 1, 1} \bigwedge_{l = 1, 1} \neg A$.
- If P is ¬Q then $P^{\vee\wedge} = Q^{\underline{\vee\wedge}}$ and $P^{\underline{\vee\wedge}} = Q^{\vee\wedge}$.
- If P is $(Q \vee R)$ then $P^{\vee\wedge} = (Q^{\vee\wedge} \vee R^{\vee\wedge})$ whereas $P^{\underline{\vee\wedge}} = \neg(Q \vee R) = \neg Q \wedge \neg R = (Q^{\underline{\vee\wedge}} \wedge R^{\underline{\vee\wedge}}) = ((\bigvee_{i = 1, m} \bigwedge_{j = 1, n} \underline{Q}_{ij}) \wedge (\bigvee_{k = 1, p} \bigwedge_{l = 1, q} \underline{R}_{kl})) = \bigvee_{r = 1, n*m} T_r$ with $T_{i+m*(k-1)} = (\bigwedge_{j = 1, n} \underline{Q}_{ij} \wedge \bigwedge_{l = 1, q} \underline{R}_{kl})$ for i = 1, …, m; k = 1, …, p; using a generalized form of the distributive law $((P_1 \vee P_2) \wedge (P_3 \vee P_4)) = (P_1 \wedge P_3) \vee (P_1 \wedge P_4) \vee (P_2 \wedge P_3) \vee (P_2 \wedge P_4)$.
- If P is $(Q \wedge R) = \neg(\neg Q \vee \neg R)$ or P is $(Q \Rightarrow R) = (\neg Q \vee R)$ then we use the previous transformations. ■

LEMMA ENDERTON [1972] *Quantifier elimination* (136)

Any predicate $P \in \mathbb{Pre}_A$ is equivalent to a quantifier-free predicate $P' \in \mathbb{Pre}_A$ (i.e. $I_A[\![P]\!] = I_A[\![P']\!]$), (with no occurrence of the Pr symbol or of terms with two occurrences of the same variable).

Proof

Generalizing the simultaneous substitution $P[x \leftarrow t]$ of term t for all occurrences of variable x in predicate P, we can define on the model of (118) the substitution $P[F' \leftarrow F]$ of F for all occurrences of F' in predicate P. We obtain P' from P by repeated application of the following transformations (such that $I_A[\![P]\!] = I_A[\![P']\!]$):

- We first eliminate the function symbols Pr from P by repeated transformation of P into $P[Pr(t) \leftarrow x] \wedge (((t = 0) \wedge (x = 0)) \vee (\neg(t = 0) \wedge (Su(x) = t)))$, where $x \in \mathbb{Var} - \mathit{Var}(P)$ is a fresh variable, as long as some term Pr(t) appears within P.
- Then we eliminate universal quantifiers $\forall v.\ Q$ from P by repeated transformation of P into $P[\forall v.\ Q \leftarrow \neg(\exists v.\ \neg Q)]$.
- Finally we eliminate all subformulae $\exists x.\ Q$ from P, starting from the innermost ones (so that Q can be assumed to be quantifier-free), by repeated application of transformations, in the following order:

\- Q is replaced by its disjunctive normal form $\exists x.\ Q = \exists x.\ Q^{\vee\wedge} = \exists x.\ \bigvee_{i = 1, m} \bigwedge_{j = 1, n} Q_{ij} = \bigvee_{i = 1, m} (\exists x.\ \bigwedge_{j = 1, n} Q_{ij})$ so that in the following we can assume that Q is of the form $\bigwedge_{j = 1, n} Q_{ij}$ with all Q_{ij} being an atomic formula A or the negation $\neg A$ of an atomic formula A.

\- $\exists x.\ \bigwedge_{i = 1, n} Q_i$ where x does not appear in Q_k, $1 \leq k \leq n$ is replaced by $Q_k \wedge (\exists x.\ \bigwedge_{i = 1, k-1} Q_i \wedge \bigwedge_{j = k+1, n} Q_j)$. Afterwards, using the commutativity of =, all Q_i can be assumed to be of the form A or $\neg A$ where A is $(Su^p(x) = Su^q(t))$ and t is 0, x or another variable $y \neq x$, with $Su^0(t) = t$ and $Su^{p+1}(t) = Su^p(Su(t))$.

\- All $(Su^p(x) = Su^q(x))$ are replaced by $(0 = 0)$ if $p = q$ and by $\neg(0 = 0)$ when $p \neq q$ whence no term has two occurrences of the same variable so that in the following we can assume that t is 0 or $y \neq x$:

. If all Q_i are of the form $\neg(Su^p(x) = Su^q(t))$ then $\exists x. \bigwedge_{i = 1, n} Q_i$ assert than we can find a value of x different from a finite number of given values, so that $\exists x.\ \bigwedge_{i = 1, n} Q_i$ is replaced by $(0 = 0)$ representing truth.

. Else some Q_k is of the form $(Su^p(x) = t')$ where t' is a term of the form $Su^r(t)$ with $t = 0$ or $t = y \neq x$ so that intuitively x can be chosen as $(t' - p) \geq 0$. Therefore Q_k is replaced by $(0 = 0)$ if $p = 0$ else by $(\neg(t' = 0) \wedge \neg(Su(t') = 0) \wedge \ldots \wedge \neg(Su^{p-1}(t') = 0))$ expressing that $x \geq 0$ whereas all terms Q_i, $i = 1, \ldots, n$, $i \neq k$, which are of the form $\neg(Su^q(x) = t_i)$ or $(Su^q(x) = t_i)$, are respectively replaced by $\neg(Su^q(t') = Su^q(t_i))$ or $(Su^q(t') = Su^q(t_i))$ since intuitively $x = (t' - p) \geq 0$ and $q + x = t_i$ is equivalent to $x = (t' - p) \geq 0$ and $q + t' = t_i + p$. Since formula Q no longer contains variable x, $\exists x.\ Q$ can be replaced by Q. ■

LEMMA ENDERTON [1972] *Definability in abacus arithmetic* (137)

A subset $E \subseteq \mathbb{N}$ is definable by $P \in \mathbb{Pre}_A$ (i.e. $\exists\, v \in \mathbb{Var}.\ E = \{s(v) \in \mathbb{N} : s \in I_A[\![P]\!]\}$) if and only if it is finite or cofinite (i.e. $\mathbb{N} - E$ is finite).

Proof

Let P' be the quantifier-free predicate equivalent to P (136) in disjunctive normal form (135). If $v \notin Var(P')$ then $\forall\, d \in \mathbb{N}.\ (s \in I_A[\![P']\!] \Leftrightarrow s[v \leftarrow d] \in I_A[\![P']\!])$ whence $E = \{s(v) \in \mathbb{N} : s \in I_A[\![P']\!]\} = \mathbb{N}$ is cofinite so that in the following we can assume that v occurs free in P'. Then we proceed by induction on the syntactical structure of P'. If P' is an atomic formula A_{ij}, it is of the form $(Su^p(v) = Su^q(t))$ where t is 0 or $x \neq v$ whence if t is 0 then $E = \emptyset$ (when $q < p$) or $E = \{q - p\}$ (when $q \geq p$) is finite else t is $x \neq v$ and $E = \{v \in \mathbb{N} : \exists\, x \geq 0.\ p + v = q + x\} = (p \geq q \to \mathbb{N} \lozenge \{v \in \mathbb{N} : v \geq (q - p)\})$ is cofinite. If P' is a negated atomic formula A_{ij} then E is the complement of a finite or cofinite set hence is cofinite or finite. If P' is a conjunction $\wedge_{j = 1, n} A_{ij}$ of atomic formulae A_{ij} or negations A_{ij} of atomic formulae then E is the intersection of finite or cofinite sets. If all are cofinite then it is cofinite else it is finite. Finally if P' is a disjunction $\vee_{i = 1, m} \wedge_{j = 1, n} A_{ij}$ then E is the union of finite or cofinite sets hence is finite or cofinite. ■

LEMMA ENDERTON [1972] *Inexpressibility of addition in abacus arithmetic* (138)

There is no formula $P \in \mathbb{Pre}_A$ equivalent to $A + B = C$ (more precisely such that $I_A[\![P]\!] = \{s \in S : (s(A) + s(B)) = s(C)\}$).

Proof

Otherwise the set of even naturals would be definable by $\exists\, k.\ (k + k = v)$ (more precisely $\exists\, k.\ P[A \leftarrow k][B \leftarrow k][C \leftarrow v]$), in contradiction with (137). ■

Let $\mathbb{Com}_A$ be the language (1) with the abacus arithmetic A as basis. Every partial recursive function (such as addition of natural numbers) can be computed by some command $C \in \mathbb{Com}_A$ (LAMBEK [1961]; BOOLOS & JEFFREY [1974], § 7). However (138) shows that addition is not expressible in $\mathbb{Pre}_A$. It follows that $\mathbb{Com}_A$ has a greater expressive power than $\mathbb{Pre}_A$. This is the source of incompleteness problems with Hoare logic.

7.4.2.2.2 Decidability of abacus arithmetic

LEMMA ENDERTON [1972] *Decidability of abacus arithmetic* (139)

The theory $\mathbb{Pre}_{A,tt}(I_A) = \{P \in \mathbb{Pre}_A : I[\![P]\!] = tt\}$ of abacus arithmetic is decidable.

Proof

The algorithm used in the proof of (136) transforms a predicate $P \in \mathbb{Pre}_A$ into an equivalent quantifier free formula P' with variables $v_1,\ldots,v_n$. Using the same algorithm we can transform $\neg\exists\, v_1. \ldots \exists\, v_n. \neg P'$ into a quantifier free formula P'' with no variables and atoms of the form $(Su^m(0) = Su^n(0))$ which are replaced by tt if m=n and ff otherwise. Then the answer is obtained using truth tables ($tt \wedge tt = tt$, $tt \wedge ff = ff$, ...). ■

The decidability of $\mathbb{Pre}_{A,tt}(I_A)$ shows that the fact that true correctness formulae in $\mathbb{Hcf}_{tt}(I)$ are not provable by $\mathbb{H}$ does not necessarily come from unprovability problems in $\mathbb{Pre}_A$ that would be inherited in Hoare logic through the consequence rule.

7.4.2.2.3 Nonstandard interpretations of abacus arithmetic

LEMMA BERGSTRA & KLOP [1984] *Nonstandard interpretations of abacus arithmetic* (140)
The *nonstandard* interpretation I'_A defined by $D' = (\{0\} \times \mathbb{N}) \cup (\mathbb{N}^+ \times \mathbb{Z})$, $V'[\![0]\!] = <0, 0>$, $V'[\![Pr]\!](<i, x>) = (i = 0 \rightarrow (x = 0 \rightarrow <0, 0> \lozenge <0, x -_{\mathbb{N}} 1>) \lozenge <i, x -_{\mathbb{Z}} 1>)$ and $V'[\![Su]\!](x) = (x = 0 \rightarrow <0, x -_{\mathbb{N}} 1> \lozenge <i, x -_{\mathbb{Z}} 1>)$ has the same theory than the standard interpretation I_A i.e. is such that $\mathbb{Pre}_{A,tt}(I'_A) = \mathbb{Pre}_{A,tt}(I_A)$.

This is an extension of the standard naturals: $<0, 0> \equiv 0 < <0, 1> \equiv Su(0) < <0, 2> \equiv Su^2(0) < \ldots < \ldots < <i, -2> < <i, -1> < <i, 0> < <i, 1> < <i, 2> < \ldots < \ldots < <j, -1> < <j, 0> < <j, 1> < \ldots$ by strictly greater nonstandard infinite numbers, organized by groups isomorphic to integers (however I'_A is not a model of Peano arithmetic with addition and multiplication, because there are too few nonstandard numbers, see BOOLOS & JEFFREY [1974], § 17).

Proof of lemma (140)

If $P' \in \mathbb{Pre}_A$ and $s \in I_A[\![P']\!]$ then obviously $s' = \lambda\, v.\ <0, s(v)> \in I'_A[\![P']\!]$. Reciprocally, if $s' \in I'_A[\![P']\!]$ then we define $s \in I_A[\![P]\!]$ where P is equivalent to P', quantifier free and in disjunctive normal form $\bigvee_{i = 1, m} \bigwedge_{j = 1, n} P_{ij}$, with the P_{ij} being atoms Q of the form $(Su^m(0) = Su^n(0))$, $(Su^m(x) = Su^n(0))$, $(Su^m(x) = Su^n(y))$ with $x \neq y$ or their negation $\neg Q$ [(136), (137)]. One of the
$\bigwedge_{j = 1, n} P_{ij}$ hence all P_{ij} must be true in s' and we define s so that all P_{ij} are also true in s. If $x \in V_P = \bigcup_{j = 1, n} Var(P_{ij})$ and $s'(x) = <k, p>$ then $s(x) = (k = 0 \rightarrow p \lozenge (k * \omega) + p - \mu)$ else $s(x) = 0$ where ω is a natural greater than $[\max(\{p : x \in V_P \wedge s'(x) = <k, p>\}) - \mu] + \kappa + 1$, $\mu = \min(\{0\} \cup \{p : x \in V_P \wedge s'(x) = <k, p>\})$ and $\kappa \in \mathbb{N}^+$ is greater than all n such that the term $Su^n(0)$ appears in P. Hence the idea is to map finitely many finite segments of groups of nonstandard numbers into disjoint segments of $\mathbb{N}$ in the same order beyond the finite initial segment used to do the standard computations. Then, the truth of $(Su^m(0) = Su^n(0))$ or its negation does not depend upon the states s' and s. If $(Su^m(x) = Su^n(0))$ is true in s' then $s'(x) = <0, n - m>$ and $n - m \geq 0$ so that it is also

true for $s(x) = n - m$. If $\neg(Su^m(x) = Su^n(0))$ is true in s' then $s'(x) = <k, p>$ with either $k = 0$ and $m+p \neq n$ in which case it is also true for $s(x) = p$ or else $k > 0$ in which case $m + s(x) = m + (k * \omega) + p - \mu > \kappa > n$. If $(Su^m(x) = Su^n(y))$ is true in s' then $s'(x) = <k, p>$ and $s'(y) = <k, q>$ with $m + p = n + q$. Therefore it is also true of $s(x) = (k * \omega) + p - \mu$ and $s(y) = (k * \omega) + q - \mu$. If $\neg(Su^m(x) = Su^n(y))$ is true in s' then $s'(x) = <k, p>$ and $s'(y) = <l, q>$ with $k = l$ and $m + p \neq n + q$ in which case $m + s(x) = m + (k * \omega) + p - \mu \neq n + (k * \omega) + q - \mu = n + s(y)$ or else $k \neq l$ in which case $m + s(x) = n + s(y)$ would imply $\omega \leq (k - l) * \omega = m - n + (q - \mu) - (p - \mu) \leq \kappa + (q - \mu) < \omega$. ■

No predicate of the standard theory $\mathbb{Pre}_{A,tt}(I_A)$ can distinguish nonstandard numbers from the standard naturals. Whence partial correctness is unchanged when considering nonstandard interpretations. For example $\{0 = 0\}$ C $\{X = 0\}$ with $C = (\neg(X = 0) * X := Pr(X))$ is true in the above nonstandard interpretation since C is either started with some value $<0, p>$ of X and terminates with the value $<0, 0>$ or else X is initially $<i, p>$ and takes successive values $<i, p>$, $<i, p - 1>$, $<i, p - 2>$, … so that execution of C never terminates (HITCHCOCK & PARK [1973]).

7.4.2.3 Incompleteness results for Hoare logic

We are now in a position to explain the incompleteness results for Hoare logic: (a) the partial correctness of a program C may not be expressible by a formula {P}C{Q} and, for the same reason that in certain cases first-order languages may be too weak assertion languages, (b) some true formula {P}C{Q} may not be provable by the deductive system $\mathbb{H} \cup$ Th where Th is the set of all theorems P true in a given interpretation and (c) Th may not be axiomatizable by a deduction system $\mathbb{T}$ so that incompleteness problems for the first-order language $\mathbb{Pre}$ show in Hoare logic through the consequence rule.

7.4.2.3.1 Unspecifiable partially correct programs

A program based upon abacus arithmetic $\mathbb{Pre}_A$ which computes addition by successive increments is not provable in $\mathbb{H}$ since, by (138), addition is not expressible in $\mathbb{Pre}_A$. It follows that Hoare logic is incomplete since partial correctness of this program is not expressible in $\mathbb{Hcf}$:

THEOREM BERGSTRA & TUCKER [1982a] *Unspecifiable partially correct programs* (141)

$\exists I.\ \gamma(\mathbb{Hcf}_{tt}(I)) \neq \mathbb{HS}(I)_{tt}$

Proof

Choosing the basis A = <{0}, {Su, Pr}, ø, #> of abacus arithmetic I_A, p = {s ∈ S : s(X) = s(x) ∧ s(Y) = s(y)}, C = (¬(X = 0) * (Y := Su(Y); X := Pr(X))), q = {s ∈ S : s(Y) = s(y) + s(x)} we have {p}C{q} hence <p, C, q> ∈ $\mathbb{HS}(I)_{tt}$ whereas by (138) there is no Q ∈ $\mathbb{Pre}_A$ such that I_A⟦Q⟧ = q whence no {P}C{Q} ∈ $\mathbb{Hcf}$ such that γ({P}C{Q}) = <p, C, q>. ■

The proof shows that "{X = x ∧ Y = y} (¬(X = 0) * (Y := Su(Y); X := Pr(X))) {Y = y + x}" is not expressible in abacus arithmetic $\mathbb{Pre}_A$ because addition cannot be represented in this logic (138). Hence we can enrich the basis A with the addition symbol + to get Presburger arithmetic PB = <{0}, {Su, Pr, +}, ø, #>. But then "{X = x ∧ Y = y} (Z := 0; (¬(X = 0) * (Z := Z + Y; X := Pr(X))) {Z = x * y}" is not expressible because multiplication cannot be represented in this logic (ENDERTON [1972], Co.32G). Presburger arithmetic has no sound and complete Hoare logic for its while programs (COOK [1978], BERGSTRA & TUCKER [1982a]) although one can be found when restricting the class of considered programs as in CHERNIAVSKY & KAMIN [1977]. Hence we must enrich the basis PB with the multiplication symbol * to get Peano arithmetic PE <{0}, {Su, Pr, +, *}, ø, #> (but then by Gödel second incompleteness theorem, Peano arithmetic I_{PE} is no longer first-order axiomatizable). This situation was first described by HAREL [1979], MEYER & HALPERN [1980] [1981], BERGSTRA, TIURYN & TUCKER [1982], MEYER [1986]: the partial correctness of a program C on basis Σ can always be expressed by a Hoare correctness formula {P}C{Q}with predicates P, Q ∉ $\mathbb{Pre}_\Sigma$ using extra symbols of Peano arithmetic PA = (0, Su, +, *, <) which may not appear in the programs and whose semantics may not be expressed in the language $\mathbb{Pre}_\Sigma$. This is because the functions and relations computed by the program are recursive, hence can be coded in arithmetic (JOHNSTONE [1987], § 4). But then incompleteness problems in $\mathbb{Pre}_{PA}$ appear through the consequence rule.

A similar incompleteness argument can be given using the fact that the transitive closure of a first-order expressible binary relation is computable by a while-program but may not be first-order expressible (GUREVICH [1984], GAIFMAN & VARDI [1985]). Then transitive closures (IMMERMAN [1983]) or least fixpoints (AHO & ULLMAN [1979]) can be turned into logical operators. Such extented first-order logics are studied in GUREVICH [1985] [1988].

7.4.2.3.2 Unprovable partially correct programs

It may also happen that partial correctness can be specified by a Hoare formula {P}C{Q}, but may not be provable using Hoare's deductive system $\mathbb{H}$ [(96) to (102)], even with the help of all true postulates Th = {P ∈ $\mathbb{Pre}$: I⟦P⟧ = tt} in the intended interpretation I. As independently shown by GERGELY & SZÖTS [1978] and WAND [1978] it might be the case that given P, Q ∈ $\mathbb{Pre}$ the proof of { P }(B * C){ Q } by (77) would involve an intermediate assertion i which cannot be expressed by a first order predicate: $\neg(\exists\, P \in \mathbb{P}\mathrm{re}.\ I[\![P]\!] = i)$. In this case, the proof using (77) cannot be carried out into $\mathbb{H}$ because intermediate (and in particular loop) invariants are not first-order definable. Otherwise stated, intermediate states in the computation of a program (e.g. with recursive procedures) are often much more complicated than the initial and final states involved in paragraph § 7.4.2.3.1. Hence the language of first order logic $\mathbb{P}\mathrm{re}$ may be too weak to describe precisely enough the set of intermediate states that is computed by a program.

7.4.2.3.2.1 Incompleteness of Hoare logic for an interpretation with decidable first-order theory

A program based upon abacus arithmetic $\mathbb{P}\mathrm{re}_A$ which involves a loop invariant I expressing addition is not provable in $\mathbb{H}$ since, by (138), the invariant I is not expressible in $\mathbb{P}\mathrm{re}_A$ (BERGSTRA & KLOP [1984]). It follows that Hoare logic is incomplete:

THEOREM GERGELY & SZÖTS [1978], WAND [1978] *Local incompleteness of Hoare logic* (142)

$$\exists\, I.\ \mathbb{H}\mathrm{cf}_{tt}(I) \neq \mathbb{H}\mathrm{cf}_{pr}(\mathrm{Th})$$

Proof (BERGSTRA & KLOP [1984])

Let us choose abacus arithmetic $\mathbb{P}\mathrm{re}_A$ and define P = ((X = x) ∧ (Y = 0)), C = (¬(X = 0) * C'), C' = (X := Pr(X); Y := Su(Y)) and Q = (Y= x).

The proof of {P}C{Q} in $\mathbb{H}$ must involve the while inference rule (101) with some I such that {I ∩ ¬(X = 0)}C'{I} and the consequence rule (102) so that (P ⇒ I) and ((I ∧ (X = 0)) ⇒ Q). Then by the soundness theorem (129), we have $\underline{\{\ I \cap \neg(X=0)\ \}C'\{\ I\ \}}$, whence $\{\langle s,\ s[X \leftarrow s(X) -_0 1]\ [Y \leftarrow s(Y) + 1]\rangle : s \in \underline{I} \wedge (s(X) \neq 0)\} \subseteq \{\langle s', s''\rangle : s'' \in \underline{I}\}$ by (19.2), (19.4) and (22) so that by (123) we must have:

(a) $\{s \in S : (s(X) = s(x)) \wedge (s(Y) = 0)\} \subseteq \underline{I}$,

(b) $\{s[X \leftarrow s(X) - 1][Y \leftarrow s(Y) + 1] : s \in \underline{I} \wedge (s(X) \neq 0)\} \subseteq \underline{I}$,

(c) $\{s \in \underline{I} : s(X) = 0\} \subseteq \{s \in S : s(Y) = s(x)\}$.

Let us show that $i = \{s \in S : s(x) = s(Y) + s(X)\}$ is the unique solution $\underline{I}$ of (a), (b), (c). If $s[X \leftarrow 0][Y \leftarrow s(X) + s(Y)] \in \underline{I}$ then $0 \leq s(X) + s(Y)$ and by (c) we have $s(X) + s(Y) = s(x)$ whence $s \in i$. By (b), we have $s[X \leftarrow n + 1][Y \leftarrow s(X) + s(Y) - (n + 1)] \in \underline{I} \wedge n + 1 \leq s(X) + s(Y)$ which implies $s[X \leftarrow n][Y \leftarrow s(X) + s(Y) - n] \in \underline{I} \wedge n \leq s(X) + s(Y)$. It follows, by induction, that $m \geq n \geq 0$ implies $(s[X \leftarrow m][Y \leftarrow s(X) + s(Y) - m] \in \underline{I} \wedge m \leq s(X) + s(Y)) \Rightarrow (s[X \leftarrow n][Y \leftarrow s(X) + s(Y) - n] \in \underline{I} \wedge n \leq s(X) + s(Y)) \Rightarrow (s \in i)$. When $m = s(X) + s(Y) \geq n = s(X) \geq 0$, we get $(s[X \leftarrow s(X) + s(Y)][Y \leftarrow 0] \in \underline{I}) \Rightarrow (s \in \underline{I}) \Rightarrow (s \in i)$. Moreover, if $s \in i$ then $s' = s[X \leftarrow s(X) + s(Y)][Y \leftarrow 0]$ is such that $s'(X) = s(Y) + s(X) = s(x) = s'(x)$ and $s'(Y) = 0$ so that by (a) we have $s \in \underline{I}$ whence $(s \in i) \Rightarrow (s \in \underline{I}) \Rightarrow (s \in i)$. In conclusion $i = \underline{I}$.

Now, by (138), an invariant I expressing that x = (Y + X) is not expressible in $\mathbb{Pre}_A$. ■

The same way, ABRAMOV [1981] proves that an invariant I expressing that A + B > C is needed to prove {A > C} (A > 0 * (A := Pr(A); B := Su(B))) {B > C} whereas ABRAMOV [1984] shows that I is not expressible in the first-order logic with basis <{0}, {Su, Pr}, {>}, #>.

7.4.2.3.2.2 Incompleteness of Hoare logic for an interpretation with undecidable first-order theory

As above (§ 7.4.2.3.1) we could enrich the predicate language $\mathbb{Pre}$ with Peano arithmetic which would allow more expressive predicates. However, as shown by COOK [1978], § 6, p. 85, this would not solve the completeness problem: if the halting problem is undecidable for interpretation *I* of $\mathbb{Com}$ then the set of valid Hoare formulae of the form {true}C{false}, $C \in \mathbb{Com}$ is not recursively enumerable hence cannot be included in the recursively enumerable set of provable Hoare formulae in any formal deductive system Th $\cup\ \mathbb{H}$. The rest of this paragraph is devoted to a detailed explanation of the argument. (Another proof of the local incompleteness of Hoare logic (142) not referring to an interpretation *I* whose first-order theory Th is decidable is given by LEIVANT & FERNANDO [1987], Theorem 1).

7.4.2.3.2.2.1 The set of provable Hoare formulae is recursively enumerable

A set E is *recursive* if there is a terminating algorithm to check that $x \in E$. A set E is *recursively enumerable* if there is an algorithm that list elements of E in some order (of course since E may be infinite, the list may never be completed but any particular element of E will appear in the list after some finite length of time).

LEMMA *Recursive enumerability of provable Hoare formulae* (143)
$\mathbb{Hcf}$ is recursively enumerable. If Th is recursive then $\mathbb{Hcf}_{pr}$(Th) is recursively enumerable.

Proof

Computer scientists know that symbols, finite lists of symbols, finite lists of lists of symbols, etc… can be coded in a machine into an integer in binary representation and that from this representation it is possible to recover the original object. *Gödel numbering* is a similar idea. An odd code $\ulcorner\sigma_i\urcorner$ is associated with the n basic logical or programming symbols σ_i : $\lceil = \rceil = 3$, $\lceil \Rightarrow \rceil = 5$, $\lceil \wedge \rceil = 7$, $\lceil \vee \rceil = 9$, $\lceil \neg \rceil = 10$, $\lceil \forall \rceil = 13$, $\lceil \exists \rceil = 15$, $\lceil \text{skip} \rceil = 17$, $\lceil := \rceil = 19$, $\lceil \Diamond \rceil = 21$, …, $\lceil \sigma_n \rceil = 2\ n + 1$ and with the constant symbols $\lceil c_i \rceil = 2\ (n + 1) + 10\ i + 1$ where $\mathbb{Cte} \subseteq \{c_i : i \in \mathbb{N}\}$, function symbols $\lceil f_i \rceil = 2\ (n+1) + 10\ i + 3$ where $\mathbb{Fun} \subseteq \{f_i : i \in \mathbb{N}\}$, relation symbols $\lceil r_i \rceil = 2(n+1) + 10\ i + 5$ where $\mathbb{Rel} \subseteq \{r_i : i \in \mathbb{N}\}$, programming variables $\lceil X_i \rceil = 2\ (n+1) + 10\ i + 7$ where $\mathbb{Pvar} \subseteq \{X_i : i \in \mathbb{N}\}$ and logical variables $\lceil x_i \rceil = 2\ (n+1) + 10\ i + 9$ where $\mathbb{Lvar} \subseteq \{x_i : i \in \mathbb{N}\}$ are assumed to be enumerable. Then a command or a predicate that is a finite string $\sigma_1,\ldots,\sigma_n$ of symbols will be coded as $\lceil \sigma_1,\ldots,\sigma_n \rceil = 2^{\lceil \sigma_1 \rceil}\ 3^{\lceil \sigma_2 \rceil}\ 5^{\lceil \sigma_3 \rceil} \ldots p_n^{\lceil \sigma_n \rceil}$ where p_n is the n^{th} prime number. Then a Hoare correctness formula $\{P\}C\{Q\}$ will be coded as $\lceil \{P\}C\{Q\} \rceil = 2^{\lceil P \rceil}\ 3^{\lceil C \rceil}\ 5^{\lceil Q \rceil}$. Then a proof in $\mathbb{H}$ that is a finite string $F_1,\ldots,F_m$ of predicates or Hoare correctness formulae will be coded as $\lceil F_1,\ldots,F_m \rceil = 2^{\lceil F_1 \rceil}\ 3^{\lceil F_2 \rceil} \ldots p_m^{\lceil F_m \rceil}$ where p_m is the m^{th} prime number.

Now given an integer n it is possible to decode it. If n is odd then n is the code $\lceil \sigma \rceil$ of a symbol $\sigma = \lceil n \rceil^{-1}$. Else it is even and can be decomposed into its prime factors $n = 2^{n_1}\ 3^{n_2} \ldots p_m^{n_m}$. The decomposition is unique. If each n_i is odd then n is the code of a finite string of symbols $\lceil n \rceil^{-1} = \lceil n_1 \rceil^{-1}\ \lceil n_2 \rceil^{-1} \ldots \lceil n_m \rceil^{-1}$. A syntactical recognizer will tell if the string is a syntactically correct command or predicate. Else each n_i can be decomposed into its prime factors. If m = 3, $\lceil n_1 \rceil^{-1}$ is a predicate P, $\lceil n_2 \rceil^{-1}$ is a command C and $\lceil n_3 \rceil^{-1}$ is a predicate Q then n is the code of Hoare formula $\{P\}C\{Q\}$. Else it can be checked if each n_i is the code of a predicate or a Hoare formula $F_i = \lceil n_i \rceil^{-1}$ so that n is the code of a proof $\lceil n \rceil^{-1} = \lceil n_1 \rceil^{-1}\ \lceil n_2 \rceil^{-1} \ldots \lceil n_m \rceil^{-1}$. Else n is not the Gödel number of a proof, Hoare formula, predicate, command or symbol. Observe that the numbering is injective: different objects have different Gödel numbers, coding and decoding is recursive that is can be done by a terminating algorithm as informally described above and the set of codes is recursive that is given any natural number n the algorithm described above always terminates with the object $\lceil n \rceil^{-1}$ coded by n or else answer that n is not a Gödel number.

To do the recursive enumeration of $\mathbb{Hcf}$, we just have to enumerate the natural numbers, for each one we check if it is the Gödel number of a Hoare formula and then output the corresponding formula. Since all Hoare correctness formulae H have a code $\lceil H \rceil \in \mathbb{N}$ and no two different formulae can have the same code, no formula H can be omitted in the enumeration.

To do the recursive enumeration of $\mathbb{Hcf}_{pr}(Th)$, we just have to enumerate the natural numbers, for each one we check that it is the Gödel number of a proof $F_1 \ldots F_n$ and then test the validity of the proof using a recognizer to check that F_i is an instance of an axiom scheme or combinatorially check that F_i follows from previous F_j by a rule of inference of $\mathbb{H}$ or we algorithmically check that $F_i \in Th$ (the algorithm exists since Th is assumed to be recursive). If the proof is correct we output the formula F_n. ■

7.4.2.3.2.2.2 The non-halting problem is not semi-decidable for Peano arithmetic

A problem P depending upon data d ∈ D with a logical answer "yes" or "no" is *decidable* (or *solvable*) (written *Decidable*(P)) if and only if there exists an algorithm (*Decision*(P) : D → {tt, ff}) which when given the data d always terminates with output "tt" or "ff" corresponding to the respective answer "yes" or "no" to the problem. A problem is *undecidable* (or *unsolvable*) when no such algorithm exists.
A problem P depending upon data d ∈ D with a logical answer "yes" or "no" is *semi-decidable* (written *Semi-decidable*(P)) if and only if there exists an algorithm (*Semi-Decision*(P) : D → {tt, ff}) which when given the data d always delivers an answer "tt" in a finite amount of time when this the answer to the problem is "yes" but may answer "ff" or may be blocked or else may not terminate when the problem for d has answer "no".
The *halting problem* is the problem of deciding whether execution of a command C ∈ $\mathbb{Com}$ started in a given initial state s ∈ *S* terminates or not (for the interpretation *I* where the basis <{0, 1}, {+, *}, ø, #> has its natural arithmetical interpretation on $\mathbb{N}$).

LEMMA CHURCH [1936], TURING [1936] [1937] *Undecidability of halting problem* (144)
The halting problem is semi-decidable but undecidable. The non-halting problem is not semi-decidable.

Sketch of proof

• Following HOARE & ALLISON [1972], we now briefly sketch a coarse proof for a subset of Pascal. A Pascal program is a finite sequence of characters. It can be represented in Pascal as a text file of arbitrary length. Obviously, we can write a Pascal function I̲ of type “**function** I(**var** F, D : text) : Boolean” such that if F is the text of a Pascal function of type “**function** F(**var** D : text) : Boolean” and D is the text of the data of F̲ then I̲(F, D) is the result F̲(D) of executing function F̲ with data D. I̲ is simply a Pascal interpreter written in Pascal but specialized in execution of Boolean functions `F` with text parameter D.

• The semi-decision algorithm for the halting problem simply consists in executing F with data D using interpreter I̲ and answering “tt” upon termination:

```
function SemiDecisionOfHaltingProblem(var F, D : text);
  var R : Boolean;
begin R := I(F, D); write('tt'); SemiDecisionOfHaltingProblem := True; end;
```

• To show that the halting problem is undecidable, we prove by reductio ad absurdum that we cannot write a termination prover in Pascal that is a function of type “**function** T̲(**var** F, D : text) : Boolean” such that for all texts F of Pascal functions

$\underline{\mathrm{F}}$ of type "**function** $\underline{\mathrm{F}}$(**var** D : text) : Boolean" and all data D of type "text", execution of T with data F and D would always terminate and yield a result $\underline{\mathrm{T}}$(F, D) which is "True" if and only if execution of $\underline{\mathrm{I}}$(F, D) i.e. of $\underline{\mathrm{F}}$ with data D does terminate. Assuming the existence of such a $\underline{\mathrm{T}}$, we let TC be the text : "**function** C(**var** F : text) : Boolean; **begin if** $\underline{\mathrm{T}}$(F, F) **then** C := **not** $\underline{\mathrm{I}}$(F, F) **else** C := True **end**;". Observe that $\underline{\mathrm{T}}$(F, F) terminates and either $\underline{\mathrm{T}}$(F, F) = True and "C := **not** $\underline{\mathrm{I}}$(F, F)" terminates or $\underline{\mathrm{T}}$(F, F) = False and "C := True" terminates so that $\underline{\mathrm{T}}$(TC, TC) is "True". Then $\underline{\mathrm{I}}$(TC, TC) = **if** $\underline{\mathrm{T}}$(TC, TC) **then not** $\underline{\mathrm{I}}$(TC, TC) **else** True = **not** $\underline{\mathrm{I}}$(TC, TC), a contradiction. In conclusion there is no algorithm by means of which we can test an arbitrary program to determine whether or not it always terminates for given data.

- The argument can be rephrased for $\mathbb{Com}$ using a coding of text files into natural numbers (or for Turing machines, see rigorous details in BOOLOS & JEFFREY [1974], § 3 & 4; DAVIS [1977]; ENDERTON [1972], § 3.5; KLEENE [1967], § 43 or ROGERS [1967], § 1.9).
- The negation ¬P of a problem P depending upon data $d \in D$ with a logical answer "yes" or "no" is the problem of answering the opposite of P : ¬P(d) = (P(d) = "yes" → "no" ◊ "yes"). We have *Decidable*(P) ⇔ [*Semi-decidable*(P) ∧ *Semi-decidable*(¬P)]. ⇒ is obvious. For ⇐ define *Decision*(P)(d) by executing alternatively one step of *Semi-Decision*(P)(d) and one step of ¬*Semi-Decision*(¬P)(d) as long as both are not terminated and by terminating *Decision*(P)(d) as soon as one of these fairly interleaved executions of *Semi-Decision*(P)(d) and ¬*Semi-Decision*(¬P)(d) is terminated.
- If the non-halting problem were semi-decidable then [*Semi-decidable*(halting) ∧ *Semi-decidable*(¬halting)] would imply *Decidable*(halting), in contradiction with the undecidability of the halting problem. ■

7.4.2.3.2.2.3 The set of valid Hoare formulae for Peano arithmetic is not recursively enumerable

LEMMA *Valid Hoare formulae for Peano arithmetic* (145)

The set $\mathbb{Hcf}_{tt}(I)$ is not recursively enumerable for Peano arithmetic (i.e.the interpretation I where the basis <{0, 1}, {+, *}, ø, #> has its natural arithmetical interpretation on $\mathbb{N}$).

Proof

Assume the contrary. Let $s \in S$ be a state, Var(C) = $X_1, \ldots, X_n$ be the variables of C with initial values $x_1 = s(X_1), \ldots, x_n = s(X_n) \in \mathbb{N}$ and $P = (X_1 = Su^{x_1}(0)) \wedge \ldots \wedge (X_n = Su^{x_n}(0))$ where $Su^0(0) = 0$ and $Su^{n+1}(0) = (Su^n(0) + 1)$. Execution of C in state s never terminates if and only if $I[\![\{P\}C\{false\}]\!] = tt$ (where false is (0 = 1)). Hence execution of C in state s never terminate if and only if the formula {P}C{false} is to be found in the recursive enumeration of $\mathbb{Hcf}_{tt}(I)$. It would follow that the non-halting problem would be semi decidable, in contradiction with (144). ■

7.4.2.3.2.2.4 Incompleteness of Hoare logic for Peano arithmetic

THEOREM COOK [1978], APT [1981a] *Incompleteness of Hoare logic for Peano arithmetic* (146)

$\exists I.\ \mathbb{Hcf}_{tt}(I) \neq \mathbb{Hcf}_{pr}(\mathrm{Th})$ (where Th is recursive and I is Peano arithmetic).

Proof

$\mathbb{Hcf}_{pr}(\mathrm{Th})$ is recursively enumerable by (143) but $\mathbb{Hcf}_{tt}(I)$ is not by (145) hence these sets are different. ■

We say that the interpretation domain D is *Herbrand definable* when all elements of D can be represented by a term:

DEFINITION *Herbrand definability* (147)

D is *Herbrand definable* if and only if $(\forall\, d \in D.\ \exists\, T \in \mathbb{Ter}.\ I[\![T]\!] = d)$

Theorem (146) is also a consequence of (145) combined with the following observation:

THEOREM BERGSTRA & TUCKER [1982a], BERGSTRA & TIURYN [1983] (148)

if Th is recursive, $|\,D\,| = |\,\mathbb{N}\,|$, D is Herbrand definable and $\mathbb{Hcf}_{tt}(I) = \mathbb{Hcf}_{pr}(\mathrm{Th})$ then $\mathbb{Hcf}_{tt}(I)$ is recursive.

Proof

If $C \in \mathbb{Com}$ then for all $n \in \mathbb{N}$ there is $C^n \in \mathbb{Com} - \mathbb{Loops}$ running at most n (assignments or test) steps of C. Indeed, since $|\,D\,| = |\,\mathbb{N}\,|$, $\mathbb{Lab}[\![C]\!]$ is finite and D is Herbrand definable there is a bijection η between $\mathbb{Lab}[\![C]\!]$ and some finite subset $\eta(\mathbb{Lab}[\![C]\!]) = \{L_0, \ldots, L_k\}$ of $\mathbb{Ter}$ with $L_0 = \eta(\surd)$ and $L_1 = \eta(C)$. Let $Xc \in \mathbb{Pvar} - Var(C)$ be a fresh variable used as program counter. C^n is $(\ldots((Xc := L_1\ ;\ I^1)\ ;\ I^2)\ ;\ \ldots I^n)$ where each $I^i = (\neg(Xc = L_0) \rightarrow (S_1\ ;\ (S_2;\ \ldots (S_{k-1}\ ;\ S_k)\ldots)) \lozenge \text{skip})$ executes one step of C (unless C is terminated). Each S_j executes the elementary step of C labeled L_j and updates the program counter Xc therefore S_j is $(Xc = L_j \rightarrow (\mathbb{Step}[\![C]\!][L_j];\ Xc := \eta(\mathbb{Succ}[\![C]\!][L_j]) \lozenge \text{skip})$ when $\mathbb{Step}[\![C]\!][L]$ is skip, X := E or X := ? and S_j is $(Xc = L_j \rightarrow (B \rightarrow Xc := \eta(\mathbb{Succ}[\![C]\!][L_j](tt)) \lozenge Xc := \eta(\mathbb{Succ}[\![C]\!][L_j](ff))) \lozenge \text{skip})$ when $\mathbb{Step}[\![C]\!][L_j]$ is B.

Moreover there is a formula R^n of $\mathbb{Pre}$ such that $\{P\}C^n\{Q\}$ holds if and only if R^n is true. R^n is $(P \Rightarrow wlp(C^n, Q))$ where by induction on the syntax of $C^n \in \mathbb{Com} - \mathbb{Loops}$, $wlp(\text{skip}, Q) = Q$, $wlp(X := E, Q) = Q[X \leftarrow E]$, $wlp(X := ?, Q) = \forall X.Q$, $wlp((B \rightarrow C_1$

$\Diamond$ C_2), Q) = ((B $\wedge$ *wlp*(C_1, Q)) $\vee$ ($\neg$B $\wedge$ *wlp*(C_2, Q))), *wlp*((C_1; C_2), Q) = *wlp*(C_1, *wlp*(C_2, Q)). *wlp*(C, Q) is explained in more details below, see (151).

If $\mathbb{Hcf}_{tt}(I) = \mathbb{Hcf}_{pr}(\mathrm{Th})$ and Th is recursive then $\mathbb{Hcf}_{tt}(I)$ is recursively enumerable by (143). Moreover $\mathbb{Hcf}_{ff}(I)$ is also recursively enumerable because {P}C{Q} $\in \mathbb{Hcf}_{ff}(I)$ if and only if $R^n \notin$ Th so that $\mathbb{Hcf}_{ff}(I)$ can be recursively enumerated by generating all formulae {P}C{Q} according to the syntax (89) and recursively testing for each of them that $R^n \notin$ Th. We conclude that $\mathbb{Hcf}_{tt}(I)$ is recursive by running alternatively one step of the algorithms to recursively enumerate $\mathbb{Hcf}_{tt}(I)$ and $\mathbb{Hcf}_{ff}(I)$. ■

BERGSTRA, CHMIELINSKA & TIURYN [1982a] have shown that the converse of (148) is not true : we may have $\mathbb{Hcf}_{tt}(I) \neq \mathbb{Hcf}_{pr}(\mathrm{Th})$ with $\mathbb{Hcf}_{tt}(I)$ recursive.

7.4.2.3.3 Unprovable valid predicates, mechanical proofs

Following COOK [1978], definition (95) of provability $\vdash_{\mathrm{Th} \cup \mathbb{H}}$ H includes the use of a given set Th of postulates which are indispensable in the consequence rule (102). Hence Hoare's system $\mathbb{H}$ can be thought of as being equipped with an infallible oracle answering questions on the validity of first-order predicates. In this way, the reasoning about the programs is separated from the reasoning about the underlying language $\mathbb{Pre}$ of invariants.

However FLOYD [1967a] propounded the definition of this set Th of postulates as the set of provable theorems in a formal deductive system. This approach was further advocated by HOARE [1969] (with the additional idea that “a programming language standard should consist of a set of axioms of universal applicability, together with a choice from a set of supplementary axioms describing the range of choices facing an implementor”). More precisely, we should define Th = $\{P \in \mathbb{Pre} : \vdash_{A \cup \mathbb{T}} P\}$ by means of a set of non-logical axioms A and a deductive system $\mathbb{T}$. The set A of non-logical axioms should be recursive (that is P $\in$ A should be decidable by a machine) and consistent (A should have at least one *model* i.e. an interpretation *I* such that $\forall$ t $\in$ A. $I[\![t]\!]$ = tt). The existence of this deductive system A $\cup$ $\mathbb{T}$ depends upon the class of interpretations *I* which is considered.

More precisely, by Gödel's 1930 completeness theorem (see BARWISE [1977], § A.1.4; BOOLOS & JEFFREY [1974], § 12; ENDERTON [1972], § 2.5; JOHNSTONE [1987], § 3 or KLEENE [1967], § 52), there is a deductive system $\mathbb{T}$ such that all (and only) formulae which holds for <u>all</u> interpretations *I* which are models of some given set of axioms A $\subseteq$ $\mathbb{Pre}$ are provable in A $\cup$ $\mathbb{T}$:

$$\{P \in \mathbb{Pre} : \vdash_{A \cup \mathbb{T}} P\} = \{P \in \mathbb{Pre} : \forall I.\ (\forall t \in A.\ I[\![t]\!] = tt) \Rightarrow I[\![P]\!] = tt\}. \quad (149)$$

However when considering <u>some</u> given intended interpretation I (for example Peano arithmetic) or a restricted non-empty class κ of such interpretations I satisfying all axioms of A, it may happen, by Gödel's 1931 second incompleteness theorem (see SMORYNSKI [1977], § A.1.4; BOOLOS & JEFFREY [1974], § 16; ENDERTON [1972], § 3.5; JOHNSTONE [1987], § 9 or KLEENE [1967], § 44) that some $P_1 \in \mathbb{Pre}$ is true for I (or all I in κ) but is neither provable nor refutable in $A \cup \mathbb{T}$. By Gödel's 1930 completeness theorem (149), this simply means that P_1 may be true for some interpretations I' and false for other interpretations I''. Hence we should add P_1 or $\neg P_1$ to A in order to eliminate the unintended interpretations I' or I''. But then there is some $P_2 \in \mathbb{Pre}$ which is true for I (or all I in κ) but is neither provable nor refutable in $A \cup \{P_1\} \cup \mathbb{T}$, and so on. This means that unintended interpretations cannot be eliminated when using only first-order concepts (ANDRÉKA, NÉMETI & SAIN [1979], BERGSTRA & TUCKER [1982a], CARTWRIGHT [1983], NÉMETI [1980]).

Since program hand-proving is tedious, long and sometimes difficult, the 1970s have lived in hopes of mechanical verification of program correctness (KING [1969], IGARASHI, LONDON & LUCKHAM [1975], BOYER & MOORE [1979]). This approach has theoretical limits determined by uncomputability problems: no computer (even ideal ones without size and time limits) can be used to <u>automatically</u> prove program partial correctness. This follows from Gödel's 1931 second incompleteness theorem which implies that the theory $\{P \in \mathbb{Pre} : \forall I \in \kappa.\ I[\![P]\!] = tt\}$ of a class κ of interpretations satisfying the axioms A (i.e. $\forall I \in \kappa.\ (\forall t \in A.\ I[\![t]\!] = tt)$) and including Peano arithmetic is not recursive (because otherwise proofs would simply consists in using the terminating algorithm to check that $\forall I \in \kappa.\ I[\![P]\!] = tt$). Hence the discovery of the invariant needed in the while rule (101) can be partly automated (WEGBREIT [1974], GERMAN & WEGBREIT [1975], KATZ & MANNA [1976]) but ultimately requires human interventions. The same way, the use of the consequence rule (102) may also call for such error-prone human interactions. But then, lonesome individuals have to carefully manage large amounts of detailed information produced by machines. This has practical limits discussed in DE MILLO, LIPTON & PERLIS [1979]. Experience with a proof editor for interactive proof checking is discussed in REPS & ALPERN [1984] and that with more ambitious verification environments in BOYER & MOORE [1988], CONSTABLE, JOHNSON & EICHENLAUB [1982] and GOOD [1984].

7.4.2.4 Cook's relative completeness of Hoare logic

COOK [1978] circumscribed these incompleteness problems by assuming that the set Th of mathematical theorems $P \Rightarrow P'$ which have to be used in the consequence rule (102) is given. This corresponds to the common mathematical practice to accept certain notions and structures as basic and work axiomatically from there on, even if we are aware that these notions cannot be completely axiomatized in the restricted language of first order logic. COOK [1978] also assumed that the intermediate invariants needed in the composition rule (99) and while rule (101) can be expressed in the first order language $\mathbb{Pre}$. This is called *relative completeness*, which consists in proving that:

$$\mathcal{Expressive}(\mathbb{Com}, \mathbb{Pre}, op, I) \Rightarrow \\ \forall C \in \mathbb{Com}. \forall P, Q \in \mathbb{Pre}. (\underline{\{ P \}C\{ Q \}} \Rightarrow \vdash_{Th \cup \mathbb{H}} \{ P \}C\{ Q \}) \qquad (150)$$

where Th = $\{P \in \mathbb{Pre} : I[\![P]\!] = tt\}$ and $\mathcal{Expressive}(\mathbb{Com}, \mathbb{Pre}, op, I)$ is a sufficient (and preferably necessary) condition implying that intermediate invariants can be expressed in $\mathbb{Pre}$. This implies that true Hoare formulae are provable: $\mathcal{Expressive}(\mathbb{Com}, \mathbb{Pre}, op, I) \Rightarrow \mathbb{Hcf}_{tt}(I) \subseteq \mathbb{Hcf}_{pr}(Th)$.

7.4.2.4.1 Expressiveness à la Clarke

Such a condition $Rc(\mathbb{Com}, \mathbb{Pre}, op, I)$ was first proposed in COOK [1978], an equivalent one was later proposed by CLARKE [1977]. Observe that in the proof of $\{P\}(C_1; C_2)\{Q\}$ we can choose an intermediate invariant I such that $\underline{I} = \{s \in S : \forall s' \in S. (\langle s, s'\rangle \in \underline{C_1}) \Rightarrow (s' \in I[\![Q]\!])\}$ whereas in the proof of $\{P\}(B * C)\{Q\}$ we can choose an intermediate invariant I such that $\underline{I} = \{s \in S : \forall s' \in S. (\langle s, s'\rangle \in \underline{B * C}) \Rightarrow (s' \in I[\![Q]\!])\}$. This leads to the following:

DEFINITON DIJKSTRA [1976] *Weakest liberal precondition* (151)

$$wlp : \mathcal{P}(S \times S) \times \mathcal{P}(S) \to \mathcal{P}(S)$$
$$wlp(r, q) = \{s \in S : \forall s' \in S. (\langle s, s'\rangle \in r) \Rightarrow (s' \in q)\}$$

so that execution of a command C starting from a state $s \in wlp(\underline{C}, \underline{Q})$ cannot reach a final state not satisfying the predicate Q. The qualifier “liberal” means that non-termination is left as an alternative. The epithet “weakest” means “making less possible restrictions on initial states”. More precisely, $\underline{P} \subseteq wlp(\underline{C}, \underline{Q})$ is another notation for { P }C{ Q } :

LEMMA DIJKSTRA [1976] *Definition of partial correctness using wlp* (152)

$$\underline{\{ p \}C\{ q \}} \Leftrightarrow p \subseteq wlp(\underline{C}, q)$$

Proof

$\underline{\{ p \}C\{ q \}} \Leftrightarrow (p \rceil \underline{C} \subseteq S \times q) \Leftrightarrow [\forall s, s' \in S. (s \in p) \Rightarrow ((\langle s, s'\rangle \in \underline{C}) \Rightarrow (s' \in q))] \Leftrightarrow (p \subseteq wlp(\underline{C}, \underline{Q}))$. ■

Termination is obviously not implied since, for example, if execution of command C never terminates then $\underline{C} = \emptyset$ whence $wlp(\underline{C}, q) = S$. DIJKSTRA [1976]'s most commonly used predicate transformer *wp* guarantees termination (see paragraph § 8.4.8). Possible alternative definitions of *wlp* are studied in MORRIS [1987a].

DEFINITION CLARKE [1977] *Expressiveness à la Clarke* (153)
The interpretation *I* is said to be *expressive* for the languages $\mathbb{Com}$ and $\mathbb{Pre}$ if and only if $\forall$ C $\in \mathbb{Com}$.$\forall$ Q $\in \mathbb{Pre}$. $\exists$ P $\in \mathbb{Pre}$. $\underline{P} = wlp(\underline{C}, \underline{Q})$.

The expressiveness requirement rules out the interpretations that have been used in paragraph § 7.4.2.3 to prove incompleteness results for Hoare logic. For example:

THEOREM *Non-expressiveness of abacus arithmetic* (154)
The standard interpretation I_A of abacus arithmetic $\mathbb{Pre}_A$ is not expressive for $\mathbb{Com}_A$

Sketch of proof
This follows from the local incompleteness (142) and relative completeness (156) below. One can also prove that for Q = (Z = 0) and C = (¬(X = 0 ∧ Y = 0) * ((X = 0 → Y := Y - 1 ◊ X := X - 1); Z := Z - 1)) we have $wlp(\underline{C}, \underline{Q}) = \{s \in S : Z(s) = X(s) + Y(s)\}$ but, by (138), no formula of $\mathbb{Pre}_A$ is equivalent to (Z = X + Y). ■

THEOREM COOK [1978] *Expressiveness of Peano arithmetic* (155)
The standard interpretation I_{PE} of Peano arithmetic PE <{0}, {Su, Pr, +, *}, {<}, #> on the domain $\mathbb{N}$ of natural numbers is expressive for $\mathbb{Com}_{PE}$ and $\mathbb{Pre}_{PE}$.

Proof
We have, by induction on $\mathbb{Com}$, $wlp(\underline{\text{skip}}, \underline{Q}) = \underline{Q}$, $wlp(\underline{X := E}, \underline{Q}) = \underline{Q[X \leftarrow E]}$, $wlp(\underline{X := ?}, \underline{Q}) = \underline{\forall X.\ Q}$, $wlp(\underline{(C_1; C_2)}, \underline{Q}) = wlp(\underline{C_1}, wlp(\underline{C_2}, \underline{Q}))$, $wlp(\underline{(B \rightarrow C_1 \lozenge C_2)}, \underline{Q}) = \underline{(B \wedge wlp(C_1, Q)) \vee (\neg B \wedge wlp(C_2, Q))}$. For a while-loop (B * C), the idea is to code the values $<s(X_1), \ldots, s(X_k)>$ of free variables $X_1, \ldots, X_k$ of B and C in state s by their Gödel number and then to code the terminating execution traces $<s_0, \ldots, s_n>$ of the loop by their Gödel number. To do this we observe that for all $n \in \mathbb{N}$ the coding of any finite sequence of naturals $<a_0, \ldots, a_n> \in seq^* \mathbb{N}$ into a natural c and the later decoding of c into the a_i is representable in Peano arithmetic by a predicate $\delta \in \mathbb{Pre}_{PE}$ with $Var(\delta) = \{c, n, i, a\}$ such that: $\forall n \in \mathbb{N}.\ \forall c \in \mathbb{N}.\ \exists <a_0, \ldots, a_n> \in \mathbb{N}^{n+1}.\ \forall i \leq n.\ \delta \Leftrightarrow (a = a_i)$ and $\forall n \in \mathbb{N}.\ \forall <a_0, \ldots, a_n> \in \mathbb{N}^{n+1}.\ \exists c \in \mathbb{N}.\ \forall i \leq n.\ \delta \Leftrightarrow (a = a_i)$ (see ENDERTON [1972], p. 247-248). Now $wlp(\underline{(B * C)}, \underline{Q}) = \{s \in S : \exists s' \in S.\ \exists n \in \mathbb{N}.\ (<s, s'> \in (\underline{B} \rceil \underline{C})^n) \wedge (s' \in \underline{\neg B \wedge Q})\}$ [by (151) and (19.6)] $= \underline{P \wedge \neg B \wedge Q}$ where we have to find P such that $\underline{P} = \{s \in S : \exists s' \in S.\ \exists n \in \mathbb{N}.\ <s, s'> \in (\underline{B} \rceil \underline{C})^n\} = \{s \in S : \exists n \in \mathbb{N}.\ \exists <s_0, \ldots, s_n> \in S^{n+1}.\ (\forall i < n.\ (s_i \in \underline{B}) \wedge (<s_i, s_{i+1}> \in \underline{C})) \wedge (s_n = s)\}$. Assume that X is the only free variable in B and C, (generalization consists in coding the values of the variables into an integer), we have $\underline{P}$

$= \{s \in S : \exists\, n \in \mathbb{N}.\ \exists <d_0, \ldots, d_n> \in \mathbb{N}^{n+1}.\ (\forall\, i < n.\ \ (s[X \leftarrow d_i] \in \underline{B}) \wedge (<s[X \leftarrow d_i], s[X \leftarrow d_{i+1}]> \in \underline{C})) \wedge (s(X) = d_n)\}$. Let $x \in \mathbb{Lvar}$ be a fresh variable not appearing in B or C. Let, by induction hypothesis, $R \in \mathbb{Pre}_{PE}$ be such that $\underline{R} = wlp(\underline{C}, \underline{(X = x)})$. We have $\underline{P} = \{s \in S : \exists\, n \in \mathbb{N}.\ \exists <d_0, \ldots, d_n> \in \mathbb{N}^{n+1}.\ (\forall\, i < n.\ s \in \underline{(B \wedge R)[X \leftarrow d_i][x \leftarrow d_{i+1}]}) \cap \underline{(X = d_n)})\} = \underline{(\exists\, n.\ \exists\, c.\ (\forall\, i < n.\ \exists\, X.\ \exists\, x.\ \delta[a \leftarrow X] \wedge \delta[i \leftarrow i+1][a \leftarrow x] \wedge B \wedge R) \wedge \delta[i \leftarrow n][a \leftarrow X])}$. ■

BERGSTRA & TUCKER [1982c] have shown that the expressiveness concept is awkward to apply for two-sorted data types. For example two independent copies of $\mathbb{N}$ can form a two-sorted sorted interpretation that is not expressive.

7.4.2.4.2 Relative completeness of Hoare logic

By the expressiveness requirement on interpretations *I*, one can always express in $\mathbb{Pre}$ intermediate invariants which are sufficient to prove the partial correctness of commands C using Hoare logic:

THEOREM COOK [1978] *Relative completeness of Hoare logic* (156)

$(I \text{ is } \textit{expressive} \text{ for } \mathbb{Com} \text{ and } \mathbb{Pre} \wedge \mathrm{Th} = \{P \in \mathbb{Pre} : I[\![P]\!] = tt\}) \Rightarrow$
$\forall\, C \in \mathbb{Com}. \forall\, P, Q \in \mathbb{Pre}.\ (\ \underline{\{\ P\ \}C\{\ Q\ \}} \Rightarrow\ \vdash_{\mathrm{Th} \cup \mathbb{H}} \{\ P\ \}C\{\ Q\ \}\ )$

Proof

By structural induction on formulas {P}C{Q}.

- If $\underline{\{\ P\ \}skip\{\ Q\ \}}$ then $\underline{P \Rightarrow P}$ is true [by (123.6) and (125)] and so is $\underline{P \Rightarrow Q}$ [by (22), (19.1), (123.6) and (125)]. It follows by hypothesis that $(P \Rightarrow P)$ and $(P \Rightarrow Q)$ belong to Th. Therefore the proof of {P}skip{Q} consists in applying the skip axiom (96) and the consequence rule (102).
- If $\underline{\{\ P\ \}X := E\{\ Q\ \}}$ then $\underline{P} \subseteq \{s \in S : s[X \leftarrow \underline{E}(s)] \in \underline{Q}\}$ [by (22) and (19.2)] whence $\underline{P \Rightarrow Q[X \leftarrow E]}$ is true [by (123.6) and (128)]. Therefore the proof of {P}X := E{Q} consists in applying the assignment axiom (97) and the consequence rule (102).
- If $\underline{\{\ P\ \}X := ?\{\ Q\ \}}$ then $\{s[X \leftarrow d] : s \in \underline{P} \wedge d \in D\} \subseteq \underline{Q}$ [by (22) and (19.3)] whence $\{s : \exists\, d \in D.\ s[X \leftarrow d] \in \underline{P}\} \subseteq \underline{Q}$ so that $\underline{\exists\, X.\ P \Rightarrow Q}$ is true [by (123.7), (123.6), and (125)]. The proof of {P}X := ?{Q} consists in applying the random assignment axiom (98) and the consequence rule (102).
- If $\underline{\{\ P\ \}(C_1;\ C_2)\{\ Q\ \}}$ then $\underline{P} \subseteq wlp(\underline{(C_1;\ C_2)}, \underline{Q})$ [by (152)] $= wlp(\underline{C_1}, wlp(\underline{C_2}, \underline{Q}))$ [by (151) and (19.4)]. By expressiveness let I and J be such that $\underline{I} = wlp(\underline{C_2}, \underline{Q})$ and $\underline{J} = wlp(\underline{C_1}, \underline{I})$. By induction hypothesis we can prove $\{I\}C_1\{J\}$ and $\{J\}C_2\{Q\}$. Moreover $\underline{P \Rightarrow I}$ and $\underline{Q \Rightarrow Q}$ are true so that the proof of $\{P\}(C_1;\ C_2)\{Q\}$ ends by application of the composition rule (99) and the consequence rule (102).

- If $\underline{\{\ P\ \}(B \rightarrow C_1 \lozenge C_2)\{\ Q\ \}}$ then we can prove $\{P \wedge B\}C_1\{Q\}$ and $\{P \wedge \neg B\}C_2\{Q\}$ [by (22), (19.5), (123.4), (123.3) and induction hypothesis] and conclude by the conditional rule (100).
- If $\underline{\{\ P\ \}(B * C)\{\ Q\ \}}$ then let I, J be such that $\underline{I} = wlp(\underline{(B * C)}, \underline{Q})$ and $\underline{J} = wlp(\underline{C}, \underline{I})$. We have $wlp(\underline{C}, \underline{I}) \subseteq wlp(\underline{C}, \underline{I})$ whence $\underline{\{\ wlp(\underline{C}, \underline{I})\ \}C\{\ I\ \}}$ [by (152)] so that by induction hypothesis we can prove that $\{J\}C\{I\}$. Moreover $\underline{I} = \{s \in S : \forall s' \in S. (\langle s, s'\rangle \in \underline{(B * C)}) \Rightarrow (s' \in \underline{Q})\}$ [by (151)] $= \{s \in S : \forall s' \in S. (\langle s, s'\rangle \in \underline{(\delta \lceil \neg B)} \cup (\underline{B} \rceil \underline{C}) \circ \underline{(B * C)}) \Rightarrow (s' \in \underline{Q})\}$ [by (19.7)] $= \{s \in S : (s \in \neg\underline{B}) \Rightarrow (s \in \underline{Q})\} \cup \{s \in S : \forall s' \in S. (\langle s, s'\rangle \in (\underline{B} \rceil \underline{C}) \circ \underline{(B * C)}) \Rightarrow (s' \in \underline{Q})\} = (\underline{\neg B \Rightarrow Q}) \cup \{s \in \underline{B} : \forall s' \in S. (\langle s, s'\rangle \in \underline{C} \circ \underline{(B * C)}) \Rightarrow (s' \in \underline{Q})\}$ [by (123)] $= (\underline{\neg B \Rightarrow Q}) \cup (\underline{B} \cap wlp(\underline{C} \circ \underline{(B * C)}, \underline{Q}))$ [by (151)] $= (\underline{\neg B \Rightarrow Q}) \cup (\underline{B} \cap wlp(\underline{C}, wlp(\underline{(B * C)}, \underline{Q})))$ [by (151)]. It follows that $(\underline{I \wedge B) \Rightarrow J}$. Whence applying the consequence rule (102) with $(I \wedge B) \Rightarrow J$, $\{J\}C\{I\}$ and $I \Rightarrow I$, we can prove $\{I \wedge B\}C\{I\}$ so that $\{I\}(B * C)\{I \wedge \neg B\}$ derives from the while rule (101). Finally $\underline{P \Rightarrow I}$ [by (152)] and $(\underline{I \wedge \neg B) \Rightarrow Q}$ so that the proof of $\{P\}(B * C)\{Q\}$ ends by application of the consequence rule (102).
- Observe that in the above proof Cook's expressiveness is only used to guarantee *weak expressiveness* (RODRÍGUEZ-ARTALEJO [1985]) that is that loop invariants for the while rule (101) and intermediate invariants for the composition rule (99) can be expressed in $\mathbb{Pre}$. ■

7.4.2.4.3 Expressiveness à la Cook and its equivalence with Clarke's notion of expressiveness

The original notion of *expressiveness* is due to COOK [1978] and was expressed in term of the predicate transformer $slp(\underline{C}, \underline{P})$ (that is the set of all final states C can reach when started in a state satisfying P, such that $\underline{\{\ p\ \}C\{\ slp(\underline{C}, p)\ \}}$ and $\underline{\{\ p\ \}C\{\ q\ \}} \Rightarrow slp(\underline{C}, p) \subseteq q$):

DEFINITION DIJKSTRA [1976] *Strongest liberal postcondition* (157)

$$slp(r, p) = \{s \in S : \exists s' \in p. \langle s', s\rangle \in r\}$$

DEFINITION COOK [1978] *Expressiveness à la Cook* (158)

The interpretation I is said to be *expressive* for the languages $\mathbb{Com}$ and $\mathbb{Pre}$ if and only if $\forall C \in \mathbb{Com}. \forall P \in \mathbb{Pre}. \exists J \in \mathbb{Pre}. \underline{J} = slp(\underline{C}, \underline{P})$.

However expressiveness à la Cook is equivalent to expressiveness à la Clarke. To show this we first observe that:

LEMMA *Relationships between wlp and slp* (159)

$\underline{\{\ p\ \}C\{\ q\ \}} \Leftrightarrow (slp(\underline{C}, p) \subseteq q)$, $wlp(r, q) = \neg slp(r^{-1}, \neg q)$ and $slp(r, p) = \neg wlp(r^{-1}, \neg p)$

Proof

$\underline{\{\ p\ \}C\{\ q\ \}} \Leftrightarrow (p \rceil \underline{C} \subseteq S \times q) \Leftrightarrow [\forall s, s' \in S. ((s \in p) \wedge (<s, s'> \in \underline{C})) \Rightarrow (s' \in q)] \Leftrightarrow (slp(\underline{C}, p) \subseteq q)$. Moreover $\neg slp(r^{-1}, \neg q) = \{s \in S : \neg(\exists\ s'.\ s' \in \neg q \wedge <s', s> \in r^{-1})\}$ [by (157)] $= \{s \in S : \forall\ s'.\ s' \in q \vee <s, s'> \notin r\} = wlp(r, q)$ [by (151)]. Finally $slp(r, p) = \neg\neg slp(r^{-1-1}, \neg\neg p) = \neg wlp(r^{-1}, \neg p)$. ■

LEMMA *Semantic inversion* (160)

if $\{X_1,\ldots,X_n\} = Free(C)$, $\{x_1,\ldots,x_n\} \cap Free(C) = \emptyset$, $\{x_1,\ldots,x_n\} \cap Free(P) = \emptyset$, $\underline{Q} = slp(\underline{C}, \underline{X_1 = x_1 \wedge \ldots \wedge X_n = x_n})$ and $Q' = (\exists\ X_1.\ \ldots\ \exists\ X_n.\ Q \wedge P)[x_1 \leftarrow X_1]\ \ldots\ [x_n \leftarrow X_n]$ then $\underline{Q'} = slp(\underline{C}^{-1}, \underline{P})$.

Proof

- We let n = 1 for simplicity. We first prove that $(\exists\ s''.\ s''(X) = s''(x) \wedge <s'', s[X \leftarrow d][x \leftarrow s(X)]> \in \underline{C}) \Leftrightarrow (<s, s[X \leftarrow d]> \in \underline{C})$. If $<s'', s[X \leftarrow d][x \leftarrow s(X)]> \in \underline{C}$ then $s''(y) = s[X \leftarrow d][x \leftarrow s(X)](y)$ for $y \notin Free(C) = \{X\}$ since execution of C does not modify the value of variables not appearing in C. It follows that $s''(y) = s(y)$ for $y \notin \{x, X\}$ and $s''(x) = s(X)$. Whence $s''(X) = s''(x)$ implies $s''(X) = s(X)$ that is $s''(y) = s(y)$ for $y \neq x$ and $s''(x) = s(X)$ so that $s'' = s[x \leftarrow s(X)]$. It follows that $<s[x \leftarrow s(X)], s[X \leftarrow d][x \leftarrow s(X)]> \in \underline{C}$ whence that $<s[x \leftarrow d'], s[X \leftarrow d][x \leftarrow d']> \in \underline{C}$ for all $d' \in D$ since $x \notin Free(C)$. For $d' = s(x)$ we conclude that $<s, s[X \leftarrow d]> \in \underline{C}$. Reciprocally if $<s, s[X \leftarrow d]> \in \underline{C}$ then $<s[x \leftarrow d'], s[X \leftarrow d][x \leftarrow d']> \in \underline{C}$ for all $d' \in D$ since $x \notin Free(C)$ so that for $d' = s(X)$ we get $<s[x \leftarrow s(X)], s[X \leftarrow d][x \leftarrow s(X)]> \in \underline{C}$ that is $<s'', s[X \leftarrow d][x \leftarrow s(X)]> \in \underline{C}$ with $s'' = s[x \leftarrow s(X)]$ so that $s''(X) = s''(x)$ since $x \neq X$.
- $I[\![(\exists\ X.\ Q \wedge P)[x \leftarrow X]]\!] = \{s \in S : s[x \leftarrow s(X)] \in I[\![(\exists\ X.\ Q \wedge P)]\!]\}$ [by (128), and (122.1)] $= \{s \in S : \exists\ d \in D.\ s[x \leftarrow s(X)][X \leftarrow d] \in \underline{Q} \cap \underline{P}\}$ [by (123.7) and (123.4)] $= \{s \in S : \exists\ d \in D.\ s[x \leftarrow s(X)][X \leftarrow d] \in slp(\underline{C}, \underline{X=x}) \cap \underline{P}\}$ [by hypothesis of (160)] $= \{s \in S : \exists\ d \in D. \exists\ s''.\ s''(X) = s''(x) \wedge <s'', s[X \leftarrow d][x \leftarrow s(X)]> \in \underline{C} \wedge s[x \leftarrow s(X)][X \leftarrow d] \in \underline{P}\}$ [by (157), (123.1), (122.1) and (121) since $X \neq x$] $= \{s \in S : \exists\ d \in D. <s, s[X \leftarrow d]> \in \underline{C} \wedge s[x \leftarrow s(X)][X \leftarrow d] \in \underline{P}\}$ [by the above argument] $= \{s \in S : \exists\ d \in D. <s, s[X \leftarrow d]> \in \underline{C} \wedge s[X \leftarrow d] \in \underline{P}\}$ [since $X \neq x$ implies $s[x \leftarrow s(X)][X \leftarrow d] = s[X \leftarrow d][x \leftarrow s(X)]$ and $x \notin Free(P)$ so that $s[X \leftarrow d][x \leftarrow d'] \in \underline{P}$ implies $s[X \leftarrow d] \in \underline{P}$] $= \{s \in S : \exists\ s'.<s, s'> \in \underline{C} \wedge s' \in \underline{P}\}$ [since $<s, s'> \in \underline{C}$ implies $s(y) = s'(y)$ when $y \notin Free(C)$ so that $s'=s[X \leftarrow d]$ where $d = s'(X)$] $= slp(\underline{C}^{-1}, \underline{P})$ [by (157)]. ■

THEOREM CLARKE [1977], JOSKO [1983], OLDEROG [1980] [1983] *Equivalent definitions of expressiveness* (161)

Expressiveness à la Cook is equivalent to expressiveness à la Clarke: (153) ⇔ (158)

Proof

If $\forall$ C $\in$ $\mathbb{Com}$. $\forall$ P $\in$ $\mathbb{Pre}$. $\exists$ J $\in$ $\mathbb{Pre}$. $\underline{J} = slp(\underline{C}, \underline{P})$ then if $\{X_1, \ldots, X_n\} = Free(C)$, $\{x_1, \ldots, x_n\} \cap Free(C) = \emptyset$, $\{x_1, \ldots, x_n\} \cap Free(P) = \emptyset$, $\underline{Q} = slp(\underline{C}, \underline{X_1 = x_1 \wedge \ldots \wedge X_n = x_n})$ and $Q' = (\forall X_1. \ldots \forall X_n. Q \Rightarrow \neg P)[x_1 \leftarrow X_1] \ldots [x_n \leftarrow X_n]$ then $\underline{\neg Q'} = \neg slp(\underline{C}^{-1}, \underline{\neg P})$ [by (160)] $= wlp(\underline{C}, \underline{P})$ [by (159)] is expressible in $\mathbb{Pre}$.

The same way if $\underline{R} = wlp(\underline{C}, \underline{X_1 = x_1 \wedge \ldots \wedge X_n = x_n})$ and $R' = (\exists X_1. \ldots \exists X_n. Q \wedge \neg P)[x_1 \leftarrow X_1] \ldots [x_n \leftarrow X_n]$ then $\underline{\neg R'} = slp(\underline{C}, \underline{P})$. ■

7.4.2.4.4 Relative completeness of Hoare logic for arithmetical while-programs and nonstandard interpretations

Hoare logic is relatively complete for while-programs applied to arithmetic:

THEOREM COOK [1978] *Relative completeness of Hoare logic for arithmetical while-programs* (162)

$\mathbb{H} \cup Th(\mathbb{N})$ is relatively complete for the standard interpretation I_{PE} of Peano arithmetic PE <{0}, {Su, Pr, +, *}, {<}, #> on the domain $\mathbb{N}$ of natural numbers where $Th(\mathbb{N}) = \{P \in \mathbb{Pre}_{PE} : I_{PE}[\![P]\!] = tt\}$ is the number theory.

Proof

By (155), I_{PE} is expressive for $\mathbb{Com}_{PE}$ and $\mathbb{Pre}_{PE}$ so that, by relative completeness (156), the Hoare logic $\mathbb{H} \cup Th(\mathbb{N})$ is relatively complete for I_{PE}. ■

HAREL [1979] has pointed out that any interpretation I can be expanded to an interpretation with a complete Hoare logic by expanding it to an arithmetical universe (but this expansion may increase the degree of undecidability of the theory of I). A simpler expansion when $\mathbb{Hcf}_{tt}(I)$ is recursive is proposed in BERGSTRA, CHMIELINSKA & TIURYN [1982b].

In (162) the facts about arithmetic one needs in a program correctness proof are given by the oracle $Th(\mathbb{N})$. BERGSTRA & TUCKER [1983] use instead Peano's first-order axiomatization of arithmetic (KLEENE [1967], § 38; JOHNSTONE [1987], § 3). Second-order Peano arithmetic PE^2 over the basis $\mathbb{Cte} = \{0, 1\}$, $\mathbb{Fun} = \{+, *\}$ and $\mathbb{Rel} = \emptyset$ can be formalized by the following axioms (axioms (163.4) to (163.7) are not strictly necessary since addition and multiplication can be defined the same way as $(x \leq y)$ is defined by $(\exists z. (x + z) = y)$):

DEFINITION *Second-order Peano arithmetic PE^2* (163)

$\forall$ x. $\neg$(x + 1 = 0) (.1)

$\forall$ x. $\forall$ y. (x + 1 = y + 1) $\Rightarrow$ (x=y) (.2)

$\forall$ x. $\neg$(x = 0) $\Rightarrow$ $\exists$ y. (x = y + 1) (.3)

$\forall$ x. (x + 0 = x) (.4)

$\forall$ x. $\forall$ y. (x + (y + 1)) = ((x + y) + 1) (.5)

$\forall$ x. (x * 0 = 0) (.6)

$\forall$ x. $\forall$ y. (x * (y + 1)) = ((x * y) + y) (.7)

$\forall$ P. (P[x $\leftarrow$ 0] $\wedge$ $\forall$ x. P $\Rightarrow$ P[x $\leftarrow$ x + 1]) $\Rightarrow$ $\forall$ x. P (.8^2)

The last axiom (163.8^2) states that if a property P is true for 0 and is true for the successor x + 1 of x whenever it is true for x then it is true for all x. Since P ranges over all subsets of $\mathbb{N}$, the second-order axiom (163.8^2) describes properties of $|\mathcal{P}(\mathbb{N})| = \aleph_1$ subsets of $\mathbb{N}$. To stay in the realm of first-order logic, one can define first-order Peano arithmetic PE^1 which consists of axioms (163.1) to (163.7) plus the axiom scheme:

DEFINITION *First-order Peano arithmetic PE^1* (164)

$\forall$ x. $\neg$(x + 1 = 0) (.1)

… …

$\forall$ x. $\forall$ y. (x * (y + 1)) = ((x * y) + y) (.7)

For all P $\in \mathbb{Pre}_{PE}$.

(P[x $\leftarrow$ 0] $\wedge$ $\forall$ x. P $\Rightarrow$ P[x $\leftarrow$ x']) $\Rightarrow$ $\forall$ x. P (.8^1)

There are $|\mathbb{N}| = \aleph_0 = \omega$ predicates P (the proof uses an enumeration of $\mathbb{Pre}_{PE}$ by Gödel numbers, see (143)) and $|\mathbb{N}| \neq |\mathcal{P}(\mathbb{N})|$ whence (164.8^1) describes less subsets of $\mathbb{N}$ than (163.8^2) (The proof that $|\mathbb{N}| \neq |\mathcal{P}(\mathbb{N})|$ is by reductio ad absurdum using Cantor diagonal argument : if $|\mathcal{P}(\mathbb{N})| = |\mathbb{N}|$ then $\mathcal{P}(\mathbb{N})$ is of the form $\{s_j : j \in \mathbb{N}\}$ where $s_j \subseteq \mathbb{N}$ for all $j \in \mathbb{N}$. Then the set $\{i \in \mathbb{N} : i \notin s_i\}$ would be some element s_k of $\mathcal{P}(\mathbb{N})$ whence a contradiction since either $k \in s_k$ and $s_k = \{i \in \mathbb{N} : i \notin s_i\}$ implies $k \notin s_k$ or $k \notin s_k$ and $s_k = \{i \in \mathbb{N} : i \notin s_i\}$ implies $k \in s_k$). Since PE^1 imposes less constrains on its interpretations than PE^2, PE^1 can have nonstandard interpretations that are disallowed by PE^2. Such nonstandard models of PL^1 (SKOLEM [1934]; BOOLOS & JEFFREY [1974], § 17 or KLEENE [1967], § 53) consists of the naturals followed by infinitely many blocks isomorphic to $\mathbb{Z}$: 0, 1, 2, … …… …-2' -1' 0' 1' 2… …… …-2" -1" 0" 1" 2"… ……, without least nor greatest block and between any two blocks lies a third. It follows that $\mathbb{N}$ is not first-order axiomatizable (although it is by PE^2 since $\mathbb{N}$ is the only model of PE^2) in the sense that there are true facts that can be proved by PE^2 but not by PE^1 (using again the diagonalization argument : if P^i is the predicate with *Free*(P^i) = x and Gödel number i then Q such that $\forall$ i $\in \mathbb{N}$. Q[x $\leftarrow$ i] = $\neg P^i$[x $\leftarrow$ i] is not one of them). So why not use second-order logics ? Essentially because PE^1 deals with finite sets of integers (as in pure arithmetic) whereas PE^2 deals with infinite sets of integers (as in mathematical analysis) and $\mathcal{P}(\mathbb{N})$ is much more complicated to understand than $\mathbb{N}$ (COHEN [1966] proved that there are infinitely many different ways to conceive of $\mathcal{P}(\mathbb{N})$ from the same $\mathbb{N}$). The same way Hoare logic deals with finite sets of variables and terminating programs i.e. finite execution traces and BERGSTRA & TUCKER [1983] have shown that Hoare logic for while-programs is essentially first order : the strongest postcondition calculus can be represented in Peano arithmetic PE^1 (because $slp(\underline{C}, \underline{P})$ can be expressed by a predicate SLP(C, P) of PE^1, see the proof of (155) and (159)) so that Hoare logic over PE^1 is equivalent to PE^1 itself (because {P}C{Q} is equivalent to (SLP(C, P) $\Rightarrow$ Q) by (159)). The comparison of Hoare-style reasoning about programs to reasoning about programs with first order rendering of predicate transformers is pursued by LEIVANT [1985].

7.4.2.4.5 On the unnecessity of expressiveness

Expressiveness is sufficient to obtain relative completeness but it is not necessary: BERGSTRA & TUCKER [1981] have shown that Hoare logic can be complete for an inexpressive interpretations I whose first-order theory has some expressive model (i.e. interpretation I' with the same first-order theory $\{P \in \mathbb{Pre} : I'[\![P]\!] = tt\}$). This point is illustrated by the following:

THEOREM BERGSTRA & TUCKER [1982a] *Unnecessity of expressiveness* (165)
Hoare logic $\mathbb{H} \cup Th(\mathbb{N})$ is relatively complete for any model I of Peano arithmetic (such that $\forall\ P \in Th(\mathbb{N}).\ I[\![P]\!] = tt$ where $Th(\mathbb{N}) = \{P \in \mathbb{Pre}_{PE} : I_{PE}[\![P]\!] = tt\}$) but I is not expressive for $\mathbb{Pre}_{PE}$ and $\mathbb{Com}_{PE}$ when I is not the standard model I_{PE} of arithmetic.

Proof

By (162), the Hoare logic $\mathbb{H} \cup Th(\mathbb{N})$ is relatively complete for I_{PE}. Since any $P \in \mathbb{Pre}_{PE}$ is true for the standard interpretation I_{PE} if and only if it is true for the nonstandard interpretation I, $\mathbb{H} \cup Th(\mathbb{N})$ is also relatively complete for I.

Let C = (X := Y; (($\neg$(X=0) * X := Pr(X)); X := Y)). Execution of C for the nonstandard interpretation I terminates only if the initial value of Y is standard. It follows that $slp(C, S) = \{s \in S : (s(X) = s(Y)) \wedge s(Y) \in \mathbb{N}\}$. Now I is not expressive for $\mathbb{Pre}_{PE}$ and $\mathbb{Com}_{PE}$ since otherwise there is a $P \in \mathbb{Pre}_{PE}$ such that $\underline{P} = slp(C, S)$, so that $\exists$ Y. P is true of X only if X is a standard natural number in contradiction with the fact that no predicate of $\mathbb{Pre}_{PE}$ can be used to distinguish among standard and nonstandard numbers. ■

BERGSTRA & TIURYN [1983] have identified and studied two necessary (but not sufficient) conditions that an interpretation I must satisfy if a sound Hoare logic is to be complete for this given I: first they prove that the first order theory of I must be *PC-compact* that is each asserted program which is true in all models of the theory is true in all models of a finite subset of the theory (if $\mathrm{Th} = \{P \in \mathbb{P}\mathrm{re} : I[\![P]\!] = \mathrm{tt}\}$ then $\forall\ H \in \mathbb{H}\mathrm{cf}_{\mathrm{tt}}[\![\mathrm{Th}]\!].\ \exists\ \mathrm{Th}' \subseteq \mathrm{Th}.\ (|\ \mathrm{Th}'\ | \in \mathbb{N}) \wedge (H \in \mathbb{H}\mathrm{cf}_{\mathrm{tt}}[\![\mathrm{Th}']\!])$ where $\mathbb{H}\mathrm{cf}_{\mathrm{tt}}[\![T]\!] = \{H \in \mathbb{H}\mathrm{cf}: \forall\ I'.\ (\forall\ P \in T.\ I'[\![P]\!] = \mathrm{tt}\ ) \Rightarrow (I'[\![H]\!] = \mathrm{tt})\}$). Secondly they prove that the partial correctness theory $\mathbb{H}\mathrm{cf}_{\mathrm{tt}}(I)$ must be decidable relative to its first order theory Th (as shown in (148)).

From a practical point of view, the incompleteness results about Hoare logic are not restrictive for hand-made proofs, just as Gödel's incompleteness theorems do not prevent mathematicians to make proofs. Only the semantic counterpart of Hoare logic matters and it is complete in the sense of (80). As far as expressiveness is concerned, the limited power of first order logic can always be overcome using infinitary logics since (76) is expressible in $L_{\omega_1\omega}$ (which allows infinite formulae $\wedge\ \Phi$ and $\vee\ \Phi$ when Φ is a countable set of formulae, BARWISE [1977]) as noticed in ENGELER [1968] [1975] and BACK [1980] [1981] (but then the finitary nature of proofs in ordinary first order logic $L_{\omega\omega}$ is lost). Also the use of a given theory Th corresponds to the common mathematical practice to accept certain notions and structures as basic and work axiomatically from there on. However when considering more complicated programming language features, Hoare logic turns out to be incomplete for intrinsic reasons.

7.4.2.5 Clarke's characterization problem

CLARKE [1977] has shown that some programming languages have no sound and relatively complete Hoare logic. The formal argument is first that if a programming language possesses a relatively complete and sound Hoare logic then the halting problem for finite interpretations must be decidable and second that Algol-like (NAUR [1960]) or Pascal-like (WIRTH [1971]) languages have an undecidable halting problem for finite interpretations with $|\ D\ | \geq 2$. The intuitive reason is that names in predicates $P, Q \in \mathbb{P}\mathrm{re}$ and in commands $C \in \mathbb{C}\mathrm{om}$ are used in a similar way: all considered objects, at a given instant of time, are given different names. Hence variables of P, Q and C can be interpreted in exactly the same way by means of states (120). But when considering Algol or Pascal-like languages, the naming conventions in P, Q and C are totally different. For example objects deeply buried in the run-time stack cannot be accessed by their name although they can be modified using procedure calls ! Such Algol or Pascal-like languages are more precisely characterized by the following:

DEFINITION CLARKE [1977] *Clarke's languages* (166)

A Clarke's language $\mathbb{L}$ is a programming language allowing procedures (with a finite number of local variables and parameters taking a finite number of values, without sharing via aliases) and the following features :

- procedures as parameters of procedure calls (without self-application); (i)
- recursion; (ii)
- static scoping; (iii)
- use of global variables in procedure bodies; (iv)
- nested internal procedures as parameters of procedure calls. (v)

A Clarke language $\mathbb{L}_j$ is obtained by disallowing feature (j).

The non-existence of Hoare logic for Clarke's languages (and other variants of (166), see CLARKE [1977] and LEIVANT & FERNANDO [1987]) introduces the *characterization problem* (CLARKE [1984]): what criteria guarantee that a programming language has a sound and relatively complete Hoare logic ? First we prove the non-existence of Hoare logics for Clarke languages and next review the literature on the characterization problem.

7.4.2.5.1 Languages with a relatively complete and sound Hoare logic have a decidable halting problem for finite interpretations

LEMMA CLARKE [1977] *Decidability of the halting problem ...* (167)

If $\mathbb{C}\mathrm{om}$ has a sound and relatively complete Hoare logic then the halting problem must be decidable for all interpretations *I* on a finite Herbrand definable domain *D*.

Proof

Let be given some particular finite interpretation *I*. There is a decision procedure to verify that $\mathrm{P} \in \mathrm{Th}$ that is $I[\![\mathrm{P}]\!] = \mathrm{tt}$: we just have to check using truth tables that P holds for the finitely many possible combinations of values of the free variables of P. Moreover since *D* is finite and Herbrand definable, $\mathbb{P}\mathrm{re}$ is expressive with respect to $\mathbb{C}\mathrm{om}$ and *I*: any subset of *D* can be represented as a finite disjunction of terms representing its elements. Then by the soundness theorem (129) and relative completeness (156) we have $\underline{\{\ \mathrm{true}\ \}\mathrm{C}\{\ \mathrm{false}\ \}} \Leftrightarrow\ \vdash_{\mathrm{Th} \cup \mathbb{H}} \{\ \mathrm{true}\ \}\mathrm{C}\{\ \mathrm{false}\ \}$ where true = (x = x) and false = ¬(x = x). Since Th is recursive, it follows from (143) that $\mathbb{H}\mathrm{cf}_{\mathrm{pr}}(\mathrm{Th})$ is recursively enumerable whence so is $\{\mathrm{C} : \vdash_{\mathrm{Th} \cup \mathbb{H}} \{\ \mathrm{true}\ \}\mathrm{C}\{\ \mathrm{false}\ \}\ \} = \{\mathrm{C} : \underline{\{\ \mathrm{true}\ \}\mathrm{C}\{\ \mathrm{false}\ \}}\ \}$ so that the non-halting problem is semi-decidable. We conclude that the halting problem cannot be undecidable (see (144)). ■

7.4.2.5.2 The halting problem for finite interpretations is undecidable for Clarke's languages

The halting problem is decidable for while-programs on finite interpretations (we may test for termination (at least theoretically) by watching the execution trace of the program to see if a state is repeated, JONES & MUCHNICK [1977]). For recursion one might expect that the program could be viewed as a type of push-down automaton for which the halting problem is also decidable (COOK [1971], JONES & MUCHNICK [1978]). However this is not true for Clarke's languages:

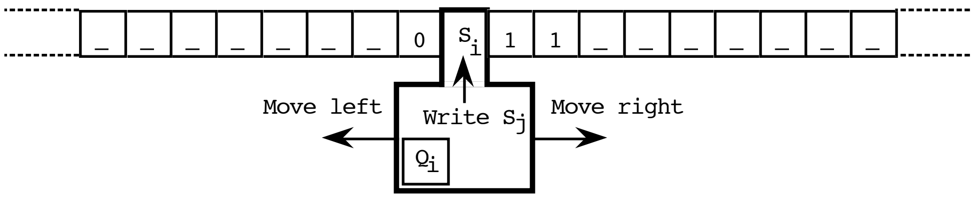
LEMMA CLARKE [1977], JONES & MUCHNICK [1978] *Undecidability of the halting problem...* (168)

Clarke's languages have an undecidable halting problem for finite interpretations with $| D | \geq 2$.

Proof

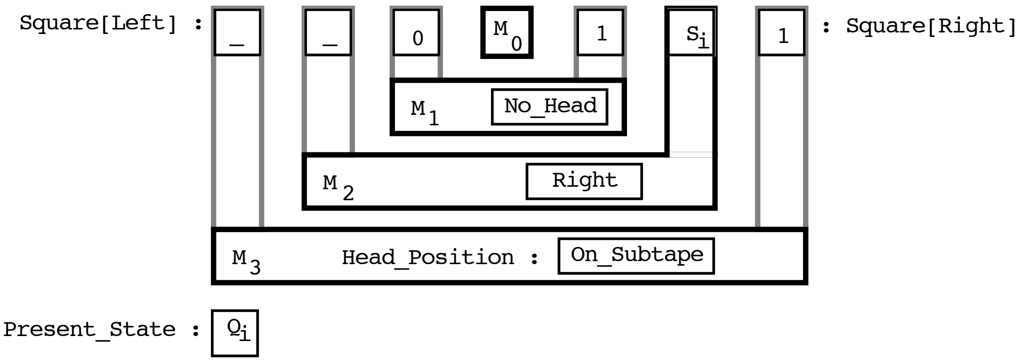
The proofs of JONES & MUCHNICK [1978] (modified in CLARKE [1977]) consist in showing that such languages can be used to simulate a queue machine which have an undecidable halting problem. Clarke's languages can also simulate the more well-known Turing machines (TURING [1936] [1937]; ENDERTON [1977], § 2; BOOLOS & JEFFREY [1974], § 5; KLEENE [1967], § 41; ROGERS [1977], § 1.5). Since, by *Church thesis* (i.e. formally unprovable mathematical assertion), all functions intuitively computable algorithmically are computable by Turing machines (CHURCH [1936], KLEENE [1936], TURING [1937]), it follows that all computable functions are programmable in Clarke's languages with $| D | \in \mathbb{N}^+ - \{1\}$. Hence by (144), the halting problem is undecidable. A similar result was previously obtained by LANGMAACK [1973], where it is shown that the pure procedure mechanism of Algol 60 can simulate any Turing machine.

A Turing machine has a finite number of internal states $Q_0, \ldots, Q_n$. It can read and write symbols chosen in a finite alphabet $S_0, \ldots, S_m$ (containing the blank symbol '_') on a potentially infinite tape marked off into squares by means of a head which can also be moved left or right, one square at a time:



An instruction has the form $M(Q_i, S_i, S_j, D, Q_j)$ where D is 'Left' or 'Right'. This instruction is executable if the machine is in the configuration $<Q_i, S_i>$, that is its internal state is Q_i and the symbol scanned under the head is S_i. Its execution consists in overwriting the square under the head by symbol S_j, in moving the head on the tape one square of the present square in the direction indicated by D and in changing the internal state into Q_j. A program consists of a finite number of instructions $M(Q_i, S_i, S_j, D, Q_j)$, i = 1, …, l. Its execution consists in repeatedly executing any one of the executable instructions. The execution of the program halts when no instruction is executable. Initially the tape contains finitely many non-blank symbols.

By induction on the number of steps, it follows that only finitely many squares can be non-blank at any time during execution of the program. Therefore Turing machines can be built-up recursively from a finite number of simpler identical machines $M_1, \ldots, M_n$ consisting only of two squares:



Machine M_0 is empty. The internal state of each machine consists of two squares (Square[Left] and Square[Right]), an indication of whether the head of the Turing machine is on its left square (Head_Position = Left), on its right square (Head_Position = Right), on a square of one of the machines $M_{i-1}, \ldots, M_1$ (Head_Position = On_Subtape), or on a square of one of the machines $M_{i+1}, \ldots, M_n$ (Head_Position = No_Head) and an indication of whether machine M_{i-1} is empty (Is_Empty_Subtape = true). We could have used only Boolean variables as done by the Pascal compiler. To execute the program of the Turing machine, machine M_n has access to the current state Q_i stored in the global variable Present_State (using feature (166.iv) of Clarke's languages). Execution of an instruction of the Turing machine (such as $M(Q_i, S_i, S_j, \text{Left}, Q_j)$ when the head is on the left square of machine M_n (i.e. Head_Position = Left) containing S_i (i.e. Square[Left] = S_i)) may require to extend the tape by one square on side D (D = Left in the example). In this case, machine M_n (currently simulated by procedure Turing_Machine) assigns S_j to its D square, No_Head to its Head_Position, Q_j to Present_State and creates a new machine M_{n+1} (by a recursive call to procedure Turing_Machine, using feature (166.ii) of Clarke's languages). This machine M_{n+1} has two blank squares and Head_Position = Initial_Head_Position = D. M_{n+1} is now in charge of executing the program of the Turing machine. To do this, machine M_{n+1} can ask the cooperation of machine M_n (hence recursively of machines $M_{n-1}, \ldots, M_1$) using functions and procedures local to M_n and passed to procedure Turing_Machine upon creation of machine M_{n+1} (using features (166.i) and (166.v) of Clarke's languages). These functions and procedures can be used by M_{n+1} to read (Scanned_Symbol_On_Ends_Of_Subtape) or write (Write_On_Ends_Of_Subtape) the squares of M_n and to read (Head_Position_On_Ends_Of_Subtape) or write (Set_Head_Position_On_Ends_Of_Subtape) the Head_Position of M_n. Procedure M_On_Subtape can be used by machine M_{n+1} to execute an instruction $M(Q_i, S_i, S_j, D, Q_j)$ of the Turing machine when M_{n+1} knows that the head of the Turing machine is not on the squares of machine M_n (by calling Head_Position_On_Ends_Of_Subtape) so that after execution of this instruction, the head will remain on the subtape represented by machines M_n, …, M_1. It follows that in order to simulate the Turing machine, machine M_{n+1} has just to take care of head moves from its squares to those of machine M_n. For example when the head is on the left square of machine M_{n+1} (i.e. Head_Position = Left) and reads S_i (i.e. Square[Left] = S_i), execution of $M(Q_i, S_i, S_j, \text{Right}, Q_j)$ consists in writing Sj in this left square, in changing Head_Position to On_Subtape, in changing the Head_Position of machine M_n to Left (by calling Set_Head_Position_On_Ends_Of_Subtape(Left)) and in going to the next state by assignment of Q_j to Present_State. When n = 0, the Head_Position of machine M_1 is simply changed to Right. Details are given in the following Pascal program (using the static scope execution rule (166.iii) which states that procedure calls are interpreted in the environment of the procedure's declaration rather than in the environment of the procedure call, thus allowing to access values normally buried deeply in the run-time stack):

```
program Simulate_Turing_Machine;

     const Blank = '_';
     type
      State_Type = 0..107; Symbol_Type = char;
      Head_Position_Type = (Left, Right, On_Subtape, No_Head); Side_Type = Left..Right;

     function Opposite(D : Side_Type) : Side_Type;
      { Opposite(Left) = Right and Opposite(Right) = Left. }
     begin {Opposite}
      case D of
       Left  : Opposite := Right;
       Right : Opposite := Left;
      end;
     end; {Opposite}

     var Present_State : State_Type;
          { Present state of the Turing machine. }
        Stopped : Boolean;
          { True only if the Turing machine must halt (initially false). }
        Configuration_Found : Boolean;
          { To check no invalid configuration (Present_State, scanned symbol) is found. }

     procedure Turing_Machine
      (Initial_Head_Position : Side_Type;
       Is_Empty_Subtape : Boolean;
       function Scanned_Symbol_On_Ends_Of_Subtape(D: Side_Type) : Symbol_Type;
       procedure Write_On_Ends_Of_Subtape(D: Side_Type; WS : Symbol_Type);
       function Head_Position_On_Ends_Of_Subtape : Head_Position_Type;
       procedure Set_Head_Position_On_Ends_Of_Subtape(P : Head_Position_Type);
       procedure Dump_Subtape;
```

```
procedure M_On_Subtape(Q : State_Type; S : Symbol_Type;
          WS : Symbol_Type; D : Side_Type; NQ : State_Type));
var
  Square : array [Side_Type] of Symbol_Type;
  Head_Position : Head_Position_Type;

 { An infinite tape is represented by its  finite  non-blank  part  as  a  quadruple }
 { <Square[Left],  subtape,  Square[Right],  Head_Position>, where the Head_Position }
 { equals D if the head is on 'Square[D]' where D is 'Left' or 'Right', 'On_Subtape' }
 { if the head is on the subtape or else 'No_Head' when the  head  is  outside  that }
 { part  of  the  whole tape. Is_Empty_Subtape is true if and only if the subtape is }
 { empty. When the subtape is not empty, it can  be  manipulated  by  functions  and }
 { procedures, similar to the ones explained below for manipulating the tape.        }
 { A call of 'Turing_Machine' extends the subtape by two additional blank squares on }
 { its  left  and  right  ends.  The  head  of  the  machine  is set on one of these }
 { additional squares as specified by Initial_Head_Position. }

 function  Scanned_Symbol_On_Ends_Of_Tape(D: Side_Type) : Symbol_Type;
  { Returns the symbol written on the square on the D end of the tape. }
 begin Scanned_Symbol_On_Ends_Of_Tape := Square[D]; end;

 procedure Write_On_Ends_Of_Tape(D : Side_Type; WS : Symbol_Type);
  { Writes WS on the square on the D end of the tape. }
 begin Square[D] := WS; end;

 function Head_Position_On_Ends_Of_Tape : Head_Position_Type;
  { To  check  if  the  head of the machine is on the left end of the tape (when the }
  { returned value is Left) or on its right end (when the returned value  is  Right) }
  { or  whether  it  is on the subtape delimited by the extreme squares (On_Subtape) }
  { or if it is outside the part of the ideal  infinite  tape  represented  by  that }
  { finite tape (No_Head). }
 begin Head_Position_On_Ends_Of_Tape := Head_Position; end;

 procedure Set_Head_Position_On_Ends_Of_Tape(P : Head_Position_Type);
  { Sets the Head_Position of the tape to P. }
 begin Head_Position := P; end;

 procedure Dump_Tape;
  { Dump all tape marking the scanned symbol under the head between square brackets. }
  procedure Dump_Square(D : Side_Type);
  begin {Dump_Square}
   if (Head_Position = D) then write('[') else write(' ');
   if (Square[D] = Blank) then write('_') else write(Square[D]);
   if (Head_Position = D) then write(']');
  end; {Dump_Square}
 begin Dump_Square(Left); Dump_Subtape; Dump_Square(Right); end;

 procedure M(Q : State_Type; S, WS : Symbol_Type; D : Side_Type; NQ : State_Type);
  { Whenever  the  machine  (which  is  not  stopped) comes to state Q (that is the }
  { instruction labeled Q) while scanning under  the  head  a  square  where  S  is }
  { written, (set Configuration_Found to true), overwrite this square with WS, move }
  { the  head  in the direction indicated by D one square of the present square and }
  { proceed to instruction labeled NQ. }
 begin {M}
  if (not Configuration_Found) and (not Stopped) and (Present_State = Q) then
  begin
   if (Head_Position = Opposite(D)) then begin
    if (Square[Opposite(D)] = S) then begin
     Configuration_Found := true;
     Square[Opposite(D)] := WS;
     if Is_Empty_Subtape then Head_Position := D
     else begin { Move the head on the Opposite(D) end of the subtape }
      Set_Head_Position_On_Ends_Of_Subtape(Opposite(D));
      Head_Position := On_Subtape;
     end;
     Present_State := NQ; { Go to next state. }
    end;
```

```
    end else if (Head_Position = D) then begin
     if (Square[D] = S) then begin
      Configuration_Found := true;
      Square[D] := WS; Head_Position := No_Head;
      { From now on, the continuation of the simulation of the Turing machine }
      { is  delegated  to  the  next call of procedure 'Turing_Machine' which }
      { extends the non empty tape by two new blank squares on its  ends  and }
      { moves the head of the machine on the square on the D end. }
      Present_State := NQ; { Go to next state. }
      Turing_Machine(D, false, Scanned_Symbol_On_Ends_Of_Tape,
                     Write_On_Ends_Of_Tape, Head_Position_On_Ends_Of_Tape,
                     Set_Head_Position_On_Ends_Of_Tape, Dump_Tape, M);
     end;
    end else if (Head_Position = On_Subtape) then begin
     if (Head_Position_On_Ends_Of_Subtape = D)
          and (Scanned_Symbol_On_Ends_Of_Subtape(D) = S) then begin
      Configuration_Found := true;
      Write_On_Ends_Of_Subtape(D, WS);
      { The head leaves the subtape for the D square }
      Set_Head_Position_On_Ends_Of_Subtape(No_Head); Head_Position := D;
      Present_State := NQ; { Go to next state. }
    end else { The move of the head on the subtape will remain on that subtape. }
     M_On_Subtape(Q, S, WS, D, NQ);
    end;
   end;
 end; {M}

begin {Turing_Machine}
 Head_Position := Initial_Head_Position;
 Square[Left] := Blank; Square[Right] := Blank;
 while (not Stopped) do begin
  { Execute one instruction of the Turing machine }
  Configuration_Found := false;
  {****************************************************************************}
 { Turing machine of ENDERTON [1977], p. 532. (This  machine  computes  x + y.  The }
  { arguments  x and y of the + function are respectively represented by a string of }
  { 1's of length x and y. The arguments are separated by a single blank.  The  tape }
  { is  otherwise blank. The head is initially on the leftmost non-blank symbol. The }
 { result is a string of 1's of length x + y. }
 {  M(State, Scan,  Write, Move,    Next state) }
    M(0,     '1',   '1',   Right,   0);  { Pass over x. }
    M(0,     Blank, '1',   Right,   1);  { Fill blank square between x and y }
    M(1,     '1',   '1',   Right,   1);  { Pass over y. }
    M(1,     Blank, Blank, Left,    2);  { Move to end of y; }
    M(2,     '1',   Blank, Left,    3);  { Erase a 1 at end of y. }
    M(3,     '1',   '1',   Left,    3);  { Back up to leftmost 1 of x + y. }
    M(3,     Blank, Blank, Right,   4);  { Halt. }
  {****************************************************************************}
  { Initialize the tape to compute x + y with x = 2 and y = 3 }
    M(100,   Blank, '1',   Left,  101); { Write y }
    M(101,   Blank, '1',   Left,  102);
    M(102,   Blank, '1',   Left,  103);
    M(103,   Blank, Blank, Left,  104); { Write blank between x and y }
    M(104,   Blank, '1',   Left,  105); { Write x }
    M(105,   Blank, '1',   Left,  106);
    M(106,   Blank, Blank, Right, 107); { Move head on leftmost 1 of x }
  {****************************************************************************}
  if Present_State = 107 then begin
   { Dump the initial tape and start the computation. }
   Dump_Tape; writeln; Present_State := 0;
  end else if (not Configuration_Found) then begin
   { Dump the final tape and halt the computation. }
   Dump_Tape; writeln; Configuration_Found := true; Stopped := true;
  end;
 end;
end; {Turing_Machine}

function  Scanned_Symbol_On_Ends_Of_Empty_Tape(D: Side_Type) : Symbol_Type;
 begin Scanned_Symbol_On_Ends_Of_Empty_Tape := '?'; end;
procedure Write_On_Ends_Of_Empty_Tape(D: Side_Type; WS : Symbol_Type); begin end;
function  Head_Position_On_Ends_Of_Empty_Tape : Head_Position_Type;
 begin Head_Position_On_Ends_Of_Empty_Tape := No_Head; end;
procedure Set_Head_Position_On_Empty_Tape(P : Head_Position_Type); begin end;
```

```
     procedure Move_On_Empty_Tape(D : Side_Type; Q : State_Type); begin end;
     procedure Dump_Empty_Tape; begin end;
     procedure M_On_Empty_Tape(Q : State_Type; S : Symbol_Type;
               WS : Symbol_Type; D : Side_Type; NQ : State_Type); begin end;

begin {Simulate_Turing_Machine}
     Present_State := 100; Stopped := false;
     Turing_Machine(Right, true, Scanned_Symbol_On_Ends_Of_Empty_Tape,
               Write_On_Ends_Of_Empty_Tape, Head_Position_On_Ends_Of_Empty_Tape,
               Set_Head_Position_On_Empty_Tape, Dump_Empty_Tape, M_On_Empty_Tape);
end. {Simulate_Turing_Machine}
```

Execution of the above program leads to the following initial and final configurations of the Turing machine :

```
 _[1] 1 _ 1 1 1 _ _ _ _ _
 _[1] 1 1 1 1 _ _ _ _ _ _
```

The program can be easily modified to simulate any Turing machine. ■

7.4.2.5.3 Languages with no sound and relatively complete Hoare logic

THEOREM CLARKE [1977] *Non-existence of Hoare logics for Clarke's languages* (169)
The Hoare logic for Clarke's languages (166) is not relatively complete in the class of all expressive interpretations.

Proof

By (167) and (168) there exists no sound and relatively complete Hoare logic for languages with features (166) since for finite domains *D*, $\mathbb{Pre}$ can be enriched by finitely many constant symbols denoting elements of *D* so that *D* is Herbrand definable. ■

First order logic is not expressive for Clarke's languages because their control structure is very complex. LEIVANT & FERNANDO [1987] gives another proof of (169) using lambda calculus (for a variant of Clarke's languages). They also exhibit a programming language whose control structure is trivial (the language consists of the single program C = (y := x; (¬(y = 0) * y := x + y)) where (0, +) is a torsion-free Abelian group) and yet for which no relatively complete logic exists in the sense of Cook. The idea is that the notion of torsion-free group ($\forall$ n ≥ 1. $\forall$ x. (x ≠ 0) ⇒ (n.x ≠ 0), where 1.x is x and (n + 1).x abbreviates (n.x) + x) is not finitely axiomatizable in first-order logic (BARWISE [1977], proposition 2.2) but is completely captured by {x ≠ 0}C{false}. In this case the poverty of the language is precisely what permits certain interpretations to be expressive, interpretations which would not be expressive had the program constructs been used more freely. This result is not in contradiction with relative completeness (156) which holds for $\mathbb{Com}$ as defined by (1) and (13).

LIPTON [1977] proved a form of converse of (167) further extended by CLARKE, GERMAN & HALPERN [1984], GERMAN & HALPERN [1983] and URZYCZYN [1983] who showed that for a deterministic acceptable programming language $\mathbb{Com}$ (see the long definition in CLARKE, GERMAN & HALPERN [1983] or GRABOWSKI [1984] and know that almost all Algol-like programming languages are acceptable (CRASEMANN & LANGMAACK [1983])) with recursion, the relative completeness of Hoare logic in the class of expressive and Herbrand definable interpretations is equivalent to the condition that the halting problem for the programming language must be decidable for finite interpretations. GRABOWSKI [1984] proved that the requirement of Herbrand definability can be dropped in the case of partial correctness. The result of CLARKE, GERMAN & HALPERN [1983] also holds for total correctness but GRABOWSKI [1985] and GRABOWSKI & HUNGAR [1988] proved that it cannot essentially be strengthened.

7.4.2.6 Nonstandard semantics and logical completeness

Observe that soundness (129) is proved for all interpretations I satisfying all theorems of Th whereas completeness in the sense of Cook (156) is relative to a given expressive interpretation I satisfying all theorems of Th and not for all interpretations I' with theory Th (i.e. such that Th is exactly the subset of formulae of $\mathbb{Pre}$ which are true for I'). However this second understanding fits better with FLOYD [1967a] and HOARE [1969] original idea that Hoare logic defines the semantics of the program where Th is the specification of the operations invoked in the program. It follows that the lack of a general completeness theorem for a sound Hoare logic implies that the operational semantics of the programming language is not the semantics about which the logic is reasoning. This remark has motivated two rather different perspectives on completeness theorems: the first (ANDRÉKA & NÉMETI [1978], GERGELY & ÚRY [1978], ANDRÉKA, NÉMETI & SAIN [1979] [1981] [1982], BERGSTRA & TUCKER [1984], BIRÓ [1981], CSIRMAZ [1980] [1981a] [1981b], GERGELY & ÚRY [1980], HORTALÁ-GONZÁLEZ & RODRÍGUEZ-ARTALEJO [1985], MAKOWSKY & SAIN [1986], NÉMETI [1980], SAIN [1985]) consists in considering “explicit time semantics” or “nonstandard semantics” which can permit of transfinite execution traces, the second (BERGSTRA & TUCKER [1982b] [1982c]) consists in considering another notion of completeness called *logical completeness* such that $\mathbb{H} \cup$ Th is logically complete if and only if any partial correctness formula H which is valid in <u>all</u> models of the specification Th is provable in $\mathbb{H} \cup$ Th (that is $\bigcap_I \mathbb{Hcf}_{tt}(I) \subseteq \mathbb{Hcf}_{pr}(\mathrm{Th})$) whereas CSIRMAZ & HART [1986] only consider finite models. Another variant of the notion of incompleteness is studied in RODRÍGUEZ-ARTALEJO [1985]. Soundness and completeness of Hoare logic is studied from an algebraic point of view in WECHLER [1983].

8. Complements on Hoare logic

8.1 Data structures

The assignment axiom (97) is correct for simple variables but cannot be used to handle all data structures. For example, considering one dimensional arrays, we could deduce $\{1 = 1\}$ T[T[2]] := 1 $\{T[T[2]] = 1\}$ from the assignment axiom (97) and since $(T[1] = 2 \wedge T[2] = 2) \Rightarrow (1 = 1)$ we deduce that $\{T[1] = 2 \wedge T[2] = 2\}$ T[T[2]] := 1 $\{T[T[2]] = 1\}$ from the consequence rule (102) but this is not correct.

- One way to handle arrays correctly is to understand the value of an array T as a function T : *dom* T → *rng* T where *dom* T is the domain of its indexes and *rng* T is the domain of its elements and to consider assignments to an element as a modification of the whole array (McCARTHY [1962], HOARE & WIRTH [1973], MANNA & WALDINGER [1981]). For example from $\{T = t \wedge (t(2) = 2) \Rightarrow (t(1) = 1)\}$ we derive that after assignment T[T[2]] := 1 we have $\{T = t[t(2) \leftarrow 1] \wedge (t(2) = 2) \Rightarrow (t(1) = 1)\}$ so that $T[T[2]] = t[t(2) \leftarrow 1](t[t(2) \leftarrow 1](2)) = t[t(2) \leftarrow 1](t(2) = 2 \rightarrow 1 \lozenge t(2)) = (t(2) = 2 \rightarrow t(1) \lozenge 1) = 1$.
- Another way to handle arrays correctly is to understand them as a collection of simple variables with possible aliases (IGARASHI, LONDON & LUCKHAM [1975]). For example, DE BAKKER [1980] suggests the following assignment axiom for subscripted variables:

$$\{P[T[E_1] \leftarrow E_2]\}\ T[E_1] := E_2\ \{P\} \tag{170}$$

with a refinement of substitution such that:

$$T[I][T[E_1] \leftarrow E_2] = (I = E_1 \rightarrow E_2 \lozenge T[I]) \tag{171}$$

and the case when an arbitrary expression stands for I is not handled. By way of example, we prove that $\{(T[2] = 2) \Rightarrow (T[1] = 1)\}$ T[T[2]] := 1 $\{T[T[2]] = 1\}$. We have $(T[T[2]] = 1)[T[T[2]] \leftarrow 1] = (\exists I.\ T[I] = 1 \wedge T[2] = I)[T[T[2]] \leftarrow 1] = (\exists I.\ (I = T[2] \rightarrow 1 \lozenge T[I]) = 1 \wedge (2 = T[2] \rightarrow 1 \lozenge T[2]) = I)$. Now the last formula is implied by $(T[2] = 2) \Rightarrow (T[1] = 1)$ since if $T[2] = 2$ holds then we choose $I = 1$ else we choose $I = T[2]$. Thus by the consequence rule and axiom (170) we get the desired result.

- Axioms of assignment applicable to multi-dimensional arrays or pointers to linked data structures are given in DEMBINSKI & SCHWARTZ [1976], CARTWRIGHT & OPPEN [1978] [1981], GRIES [1978], GRIES & LEVIN [1980], JANSSEN & VAN EMDE BOAS [1977], KOWALTOWSKI

[1977], LUCKHAM & SUZUKI [1979], MANNA & WALDINGER [1981], MORRIS [1982], PRATT [1976], SCHWARTZ & BERRY [1979]. One can also consult HOARE [1972a], COOK & OPPEN [1975], OPPEN & COOK [1975], JONES [1980], TIURYN [1985] and HOARE, HE JIFENG & SANDERS [1987] on proving partial correctness properties of programs with user-defined data types and BURSTALL [1972] and NELSON [1983] for the special case of linear lists.

8.2 Procedures

First we define the syntax and relational semantics of recursive parameterless procedures. Then we consider partial correctness proofs based upon computation induction (a generalization of Scott induction) which leads to Hoare's recursion rule. Since this only rule is not complete, we consider Park's fixpoint induction, which, using auxiliary variables, can be indirectly transcribed into Hoare's rule of adaptation. Then these rules are generalized for value-result parameters and numerous examples of application are provided. References to the literature are given for variable parameters and procedures as parameters.

8.2.1 Recursive parameterless procedures

8.2.1.1 Syntax and relational semantics of a parameterless procedural language

Let us now consider programs of the form Pn :: C_1; C_2 which consists of a command C_2 calling a single recursive parameterless procedure Pn with body C_1:

DEFINITION *Syntax of a parameterless procedural language* (172)

Pn : $\mathbb{Proc}$ *Procedure names* (.1)

Pg : $\mathbb{Prog}$ *Programs* (.2)

Pg ::= Pn :: C_1; C_2

C : $\mathbb{Com}$ *Commands* (.3)

C ::= skip | X := E | X := ? | (C_1; C_2) | (B → C_1 ◊ C_2) | (B * C) | Pn

The relational semantics $\underline{\text{Pn}}$ of a procedure Pn is a relationship between states such that a call of Pn in state s leads (if execution of such a call does terminate) to a state s' such that <s, s'> ∈ $\underline{\text{Pn}}$. When a command C contains a procedure call, the relational semantics of the command can only be defined if the semantics of the procedure is known. Therefore, we define the semantics $\underline{\text{C}}$ of a command C as a function of the semantics r of the procedure Pn :: C_1. The definition of $\underline{\text{C}}$(r) = $I⟦C⟧$(r) is similar to (19) but for the fact that we must define the effect of a procedure call $\underline{\text{Pn}}$ (r) = r and that of a program: $\underline{\text{Pn} :: C_1; C_2} = \underline{C_2}(r)$ where r is the semantics of procedure Pn. If procedure Pn is not recursive then its semantics is simply $r = \underline{C_1}(\emptyset)$. If it is recursive, the definition is circular since $r = \underline{C_1}(r)$. To avoid paradoxes, we must prove that this equation has a solution and, if it is not unique, we must specify which one is to be considered. To do this, observe that $\mathcal{P}(S \times S)$ is a complete lattice for the partial ordering $\subseteq$ where $\emptyset$ is the infimum, $S \times S$ is the supremum, $\cup$ is the least upper bound and $\cap$ is the greatest lower bound. Also $\underline{C_1}$ is monotone i.e. $r \subseteq r' \Rightarrow \underline{C_1}(r) \subseteq \underline{C_1}(r')$. Therefore, according to TARSKI [1955], $\underline{C_1}$ has a least fixed point $\mathit{lfp}\,\underline{C_1} = \cap\{r \in \mathcal{P}(S \times S) : \underline{C_1}(r) \subseteq r\}$ such that $\mathit{lfp}\,\underline{C_1} = \underline{C_1}(\mathit{lfp}\,\underline{C_1})$ and $r = \underline{C_1}(r) \Rightarrow (\mathit{lfp}\,\underline{C_1} \subseteq r)$. The semantics of procedure Pn is chosen to be $\mathit{lfp}\,\underline{C_1}$ (this could be justified with respect to an operational semantics as in (19), see DE BAKKER [1980, Ch. 5]). We get the following:

DEFINITION after SCOTT & DE BAKKER [1969], DE BAKKER & DE ROEVER [1972]

Relational semantics (173)

$I : \mathbb{Com} \rightarrow (\mathcal{P}(S \times S) \rightarrow \mathcal{P}(S \times S))$ (.1)

$\underline{\text{skip}}\,(r) = \{\langle s, s\rangle : s \in S\}$ (.2)

$\underline{X := E}\,(r) = \{\langle s, s[X \leftarrow \underline{E}(s)]\rangle : s \in S\}$ (.3)

$\underline{X := ?}\,(r) = \{\langle s, s[X \leftarrow d]\rangle : s \in S \wedge d \in D\}$ (.4)

$\underline{(C_1 ; C_2)}\,(r) = \underline{C_1}(r) \circ \underline{C_2}(r)$ (.5)

$\underline{(B \rightarrow C_1 \Diamond C_2)}\,(r) = (\underline{B} \rceil \underline{C_1}(r)) \cup (\neg\underline{B} \rceil \underline{C_2}(r))$ (.6)

$\underline{(B * C)}\,(r) = (\underline{B} \rceil \underline{C}(r))^* \lceil \neg\underline{B}$ (.7)

$\underline{\text{Pn}}\,(r) = r$ (.8)

$\underline{\text{Pn} :: C_1; C_2} = \underline{C_2}(\mathit{lfp}\,\underline{C_1})$ (.9)

(For a version of (173) taking termination into account see HITCHCOCK & PARK [1973], DE BAKKER [1976], DE ROEVER [1976], PLOTKIN [1976], MAJSTER-CEDERBAUM [1980] and APT & PLOTKIN [1986]. A semantics of nondeterministic recursive programs based upon fixpoints of relations on functions rather than fixpoints of functions on relations is presented in PLAISTED [1986]).

The recursive programming language (172) with its relational semantics (173) is strictly more powerful than the iterative language (1) with its semantics (19) since, for example, when the data space D is finite,the first may use an unbounded storage space while the second may not (the comparison is pursued in KFOURY [1983], KFOURY & URZYCZYN [1985]).

8.2.1.2 The recursion rule based upon computational induction

To prove partial correctness, we need, in addition of theorems (76), means of proving $\{p\}\mathit{lfp}\,\underline{C}_1\{q\}$ or, more generally, of proving a property $P(\mathit{lfp}\,F)$ of the least fixpoint $\mathit{lfp}\,F$ of a monotone function F on a complete lattice L:

LEMMA SCOTT & DE BAKKER [1969] *Computation induction* (174)

If $\langle L, \leq, \cup, \cap, \bot\rangle$ is a complete lattice and $F : L \to L$ is monotone then $[P(\bot) \wedge \forall X.\ P(X) \Rightarrow P(F(X)) \wedge \forall \alpha.\ \forall X \in \alpha \to L.\ (\forall \beta < \alpha.\ P(X_\beta)) \Rightarrow P(\cup_{\beta < \alpha} X_\beta)] \Rightarrow P(\mathit{lfp}\,F)$

Proof

$\mathit{lfp}\,F$ is one of the elements of the transfinite sequence $X_0 = \bot, \ldots, X_\alpha = F(X_{\alpha-1})$ for successor ordinals $\alpha, \ldots, X_\alpha = \cup_{\beta < \alpha} X_\beta$ for limit ordinals α (see COUSOT & COUSOT [1979]). Therefore we can prove that $P(\mathit{lfp}\,F)$ holds by proving $\forall \alpha.\ P(X_\alpha)$, which, by transfinite induction, is implied by $P(\bot) \wedge \forall X.\ P(X) \Rightarrow P(F(X)) \wedge \forall \alpha.\ \forall X.\ (\forall \beta < \alpha.\ P(X_\beta)) \Rightarrow P(\cup_{\beta < \alpha} X_\beta)$. ■

Computation induction is a generalization of *Scott induction* (also called *computational induction* in MANNA, NESS & VUILLEMIN [1972]) which corresponds to the particular case when F is upper-continuous (if so $\mathit{lfp}\,F = X^\omega$ where $\omega = |\mathbb{N}|$) and P is *admissible* (i.e. $\forall X \in \mathbb{N} \to L.\ (\forall n \in \mathbb{N}.\ P(X_n)) \Rightarrow P(\cup_{n \in \mathbb{N}} X_n)$) so that $(P(\bot) \wedge \forall X.\ P(X) \Rightarrow P(F(X)) \Rightarrow P(\mathit{lfp}\,F)$. When specialized to the partial correctness proof $\{p\}\mathit{lfp}\,\underline{C}_1\{q\}$ of a recursive procedure Pn :: C_1, computation induction leads to the following theorem where the assumption $\forall \langle s, s'\rangle \in r.\ \forall v \notin \mathit{Var}(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ \langle s[v \leftarrow d], s'[v \leftarrow d]\rangle \in r$ states that the relational semantics r of C_1 is without side-effects, more precisely that the variables v not appearing in C_1 cannot be modified and can have any value during execution of C_1:

THEOREM *Partial correctness proof of procedures by computation induction* (175)

$[\forall r \in \mathcal{P}(S^2).$

$\quad(\forall \langle s, s'\rangle \in r.\ \forall v \notin \mathit{Var}(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ \langle s[v \leftarrow d], s'[v \leftarrow d]\rangle \in r)$

$\quad\Rightarrow (\{p\}r\{q\} \Rightarrow \{p\}\underline{C}_1(r)\{q\})]$

$\Rightarrow \{p\}\mathit{lfp}\,\underline{C}_1\{q\}$

Proof

We prove $(\forall \langle s, s'\rangle \in \mathit{lfp}\,\underline{C}_1.\ \forall v \notin \mathit{Var}(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ \langle s[v \leftarrow d], s'[v \leftarrow d]\rangle \in \mathit{lfp}\,\underline{C}_1 \wedge \{p\}\mathit{lfp}\,\underline{C}_1\{q\})$ by computation induction (174). $(\forall \langle s, s'\rangle \in \emptyset.\ \forall v \notin \mathit{Var}(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ \langle s[v \leftarrow d], s'[v \leftarrow d]\rangle \in \emptyset \wedge \{p\}\emptyset\{q\})$ is obviously true. If, by induction hypothesis, $\forall \langle s, s'\rangle \in r.\ \forall v \notin \mathit{Var}(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ \langle s[v \leftarrow d], s'[v \leftarrow d]\rangle \in r \wedge \{p\}r\{q\}$ is true then $\{p\}\underline{C}_1(r)\{q\}$ holds by hypothesis of theorem (175) and we can prove $\forall <s, s'> \in \underline{C_1}(r).\ \forall v \notin Var(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ <s[v \leftarrow d], s'[v \leftarrow d]> \in \underline{C_1}(r)$ by structural induction on the syntax (172) of C_1. Finally $\forall \alpha.\ \forall r.\ (\forall \beta < \alpha.\ (\forall <s, s'> \in r_\beta.\ \forall v \notin Var(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ <s[v \leftarrow d], s'[v \leftarrow d]> \in r_\beta) \wedge (p \rceil r_\beta \subseteq S \times q)) \Rightarrow ((\forall <s, s'> \in \cup_{\beta < \alpha} r_\beta.\ \forall v \notin Var(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ <s[v \leftarrow d], s'[v \leftarrow d]> \in \cup_{\beta < \alpha} r_\beta) \wedge p \rceil (\cup_{\beta < \alpha} r_\beta) \subseteq S \times q)$ is again obviously true. ■

Computation induction (175) can be directly translated into Hoare logic by the following *recursion rule* due to HOARE [1971b]:

Recursive procedure Pn :: C_1; :

$$\frac{\{P\}Pn\{Q\} \vdash \{P\}C_1\{Q\}}{\{P\}Pn\{Q\}} \qquad \textit{Recursion rule} \qquad (176)$$

This rule of inference in the sense of PRAWITZ [1965] means that "$H_1, \ldots, H_n$" is a formal proof of H_n using:

$$\mathbb{H} \cup \left\{ \frac{H \vdash H'}{H''} \right\}$$

if and only if "$H, H_1, \ldots, H_n$" is a formal proof of H_n (as defined at (94)) using:

$$\mathbb{H} \cup \left\{ \frac{H'}{H''} \right\}$$

Formally, the metarule (176) can be avoided by transforming the proof system into in ordinary one as shown by APT [1981a] and AMERICA & DE BOER [1989].

Example *Partial correctness proof by computation induction* (177)

The following program terminates with X = n and Y = n ! when initially X = n ≥ 0:

(178)
```
procedure F;
begin
  if X = 0 then Y := 1
           else begin X := X - 1; F; X := X + 1; Y := Y * X; end;
end;
F;
```

Partial correctness "{ true } F { Y = X ! }" can be proved using (176) as follows:

(a) $\{ \text{true} \}\ F\ \{ Y = X\,! \}$ by induction hypothesis

(b) $\{ \text{true} \wedge X = 0 \}\ Y := 1\ \{ Y = X\,! \}$ by (97), (102)

(c) $\{ Y * X = X\,! \}\ Y := Y * X\ \{ Y = X\,! \}$ by (97)

(d) $\{ Y * (X + 1) = (X + 1)\,! \}\ X := X + 1\ \{ Y * X = X \}$ by (97)

(e) $\{ Y = X\,! \} \Rightarrow \{ Y * (X + 1) = (X + 1)\,! \}$ from Th

(f) $\{ Y = X\,! \}\ X := X + 1\ \{ Y * X = X\,! \}$ by e, d, (102)

(g) $\{ \text{true} \}\ X =: X - 1\ \{ \text{true} \}$ by (97)

(h) $\{ \text{true} \wedge \neg(X = 0) \}\ (((X := X - 1; F); X := X + 1); Y := Y * X)\ \{ Y * X = X\,! \}$ by g, a, f, c, (99), (102)

(i) $\{ \text{true} \}\ (X = 0 \rightarrow Y := 1 \lozenge (((X := X - 1; F); X := X - 1); Y := Y * X))\ \{ Y = X\,! \}$ by b, h, (100)

(j) $\{ \text{true} \}\ F\ \{ Y = X\,! \}$ by a, i, (175)

Observe that the recursion rule (176) is powerful enough to prove "$\{\text{true}\}\ F\ \{Y = X\,!\}$" because this conclusion (j) can be used as induction hypothesis (a). This is not the case for proving "$\{X = n\}\ F\ \{X = n \wedge Y = n\,!\}$" since then we need induction hypothesis "$\{X = n - 1\}\ F\ \{X = n - 1 \wedge Y = (n - 1)\,!\}$" which cannot be directly derived from the conclusion "$\{X = n\}\ F\ \{X = n \wedge Y = n\,!\}$" using the consequence rule (102). Hence a proof method using (175) or (176) only is not relatively complete (APT [1981a]). ■

Various proof rules, known as *copy-rule induction*, have been proposed by GORELICK [1975], CLARKE [1977], LANGMAACK & OLDEROG [1980], APT [1981], OLDEROG [1981] [1983b] [1983c], TRAKHTENBROT, HALPERN & MEYER [1983] to extend the recursion rule (176) for higher-order procedure calls.

8.2.1.3 The rule of adaptation based upon fixpoint induction

Since the proof method (175) based upon computation induction is not complete, we come back to the problem of proving a property $P(\mathit{lfp}\ F)$ of the least fixpoint $\mathit{lfp}\ F$ of a monotone function F on a complete lattice L. When $P(\mathit{lfp}\ F)$ is of the form $X \cap \mathit{lfp}\ F \leq Y$, which is the case for $\underline{\{p\}}\mathit{lfp}\ \underline{C_1}\underline{\{q\}}$, we can use fixpoint induction:

LEMMA PARK [1969] *Fixpoint induction* (179)

If $\langle L, \leq, \cup, \cap, \perp\rangle$ is a complete lattice and $F : L \rightarrow L$ is monotone then

$$(X \cap \mathit{lfp}\ F \leq Y) \Leftrightarrow (\exists Z \in L.\ F(Z) \leq Z \wedge X \cap Z \leq Y)$$

Proof

For $\Rightarrow$ we can choose $Z = \mathit{lfp}\ F$ so that $F(Z) = Z$ and for $\Leftarrow$ we have $(F(Z) \leq Z) \Rightarrow (\mathit{lfp}\ F \leq Z)$ since $\mathit{lfp}\ F = \cap\{Z : F(Z) \leq Z\}$ by TARSKI [1955] whence $X \cap \mathit{lfp}\ F \leq X \cap Z \leq Y$. ■

When specialized to the partial correctness proof $\underline{\{p\}}\mathit{lfp}\ \underline{C_1}\underline{\{q\}}$ of a recursive procedure $Pn :: C_1$, fixpoint induction (179) leads to the following proof method (a version of which is used in COURCELLE [1985] to establish the partial correctness of clausal programs):

THEOREM after PARK [1969], MANNA & PNUELI [1970] *Partial correctness proof of* (180)

- $(\exists\, r \in \mathcal{P}(S^2).\ \underline{C_1}(r) \subseteq r \wedge \underline{\{}p\underline{\}}r\underline{\{}q\underline{\}}) \Rightarrow \underline{\{}p\underline{\}}\mathit{lfp}\ \underline{C_1}\underline{\{}q\underline{\}}$ (.1)
- $\underline{\{}p\underline{\}}\mathit{lfp}\ \underline{C_1}\underline{\{}q\underline{\}} \Rightarrow (\exists\, r \in \mathcal{P}(S^2).\ \underline{C_1}(r) \subseteq r \wedge \underline{\{}p\underline{\}}r\underline{\{}q\underline{\}}$ (.2)
  $\wedge\ \forall <s, s'> \in r.\ \forall v \notin \mathit{Var}(C).\ (s(v) = s'(v))$
  $\wedge\ \forall d \in D.\ <s[v \leftarrow d],\ s'[v \leftarrow d]> \in r)$

Proof

We have $(\exists\, r \in \mathcal{P}(S^2).\ \underline{C_1}(r) \subseteq r \wedge \underline{\{}p\underline{\}}r\underline{\{}q\underline{\}}) \Rightarrow (\exists\, r \in \mathcal{P}(S^2).\ \underline{C_1}(r) \subseteq r \wedge (p \times S) \cap r \subseteq S \times q) \Rightarrow ((p \times S) \cap (\mathit{lfp}\ \underline{C_1}) \subseteq S \times q) \Rightarrow (p \rceil (\mathit{lfp}\ \underline{C_1}) \subseteq S \times q) \Rightarrow \underline{\{}p\underline{\}}\mathit{lfp}\ \underline{C_1}\underline{\{}q\underline{\}}$. Reciprocally, if $\underline{\{}p\underline{\}}\mathit{lfp}\ \underline{C_1}\underline{\{}q\underline{\}}$ then obviously $\underline{C_1}(r) \subseteq r \wedge \underline{\{}p\underline{\}}r\underline{\{}q\underline{\}}$ for $r = \mathit{lfp}\ \underline{C_1}$. Moreover we can prove $P(\mathit{lfp}\ \underline{C_1})$ where $P(r) = (\forall <s, s'> \in r.\ \forall v \notin \mathit{Var}(C).\ (s(v) = s'(v)) \wedge \forall d \in D.\ <s[v \leftarrow d],\ s'[v \leftarrow d]> \in r)$ by computation induction (174). $P(\emptyset)$ is obvious. Assuming $P(r)$ we can prove $P(\underline{C_1}(r))$ by structural induction on the syntax of C_1. For example $P(\underline{Pn}(r)) = P(r)$ [by (173.8)] = tt [by induction hypothesis]. $P(\underline{X := E}(r)) = \forall v \notin \{X\} \cup \mathit{Var}(E).\ s(v) = s[X \leftarrow \underline{E}(s)](v) \wedge \forall d \in D.\ <s[v \leftarrow d], s[X \leftarrow \underline{E}(s)][v \leftarrow d]> \in \{<s'', s''[X \leftarrow \underline{E}(s'')]> : s'' \in S\}$ is true, etc. Finally $\forall \alpha.\ \forall r.\ (\forall \beta < \alpha.\ P(r_\beta)) \Rightarrow P(\cup_{\beta < \alpha} r_\beta)$ is obvious. ■

Example *Partial correctness proof by fixpoint induction I* (181)

Partial correctness "{ X = n } F { X = n ∧ Y = n ! }" of program (178) can be proved using (180.1) as follows:

(a)	$r = \{<s, s'> : s'(X) = s(X) \wedge s'(Y) = s(X)\,!\}$	by definition of r
(b)	$\underline{Y := 1}(r) \subseteq \{<s, s[Y \leftarrow 0\,!]> : s \in S\}$	by (173.2)
(c)	$\underline{X = 0} \rceil \underline{Y := 1}(r) \subseteq r$	by b, a
(d)	$\underline{X := X - 1}(r) \subseteq \{<s, s[X \leftarrow s(X) - 1]> : s \in S\}$	by (173.2)
(e)	$\underline{F}(r) \subseteq \{<s, s'> : s'(X) = s(X) \wedge s'(Y) = s(X)\,!\}$	by (173.8), a
(f)	$(\underline{X := X - 1; F})(r) \subseteq \{<s, s'> : s'(X) = s(X) - 1 \wedge s'(Y) = (s(X) - 1)\,!\}$	by d, e, (173.5)
(g)	$\underline{X := X + 1}(r) \subseteq \{<s, s[X \leftarrow s(X) + 1]> : s \in S\}$	by (173.2)
(h)	$((\underline{X := X - 1; F); X := X + 1})(r) \subseteq \{<s, s'> : s'(X) = s(X) \wedge s'(Y) = (s(X) - 1)\,!\}$	by f, g, (173.5)
(i)	$\underline{Y := Y * X}(r) \subseteq \{<s, s[Y \leftarrow s(Y) * s(X)]> : s \in S\}$	by (173.2)
(i)	$\neg\underline{X = 0} \rceil (((\underline{X := X - 1; F); X := X + 1); Y := Y * X})(r) \subseteq r$	by f, g, (173.5)
(j)	$\underline{(X = 0 \to Y := 1 \lozenge (((X := X - 1; F); X := X - 1); Y := Y * X))}(r) \subseteq r$	by c, j, (173.6)
(k)	$\underline{\{}\{s : s(X) = n\}\underline{\}}\ r\ \underline{\{}\{s : s(X) = n \wedge s(Y) = n\,!\}\underline{\}}$	by a, (22)
(l)	$\underline{\{}\{s : s(X) = n\}\underline{\}}\ \mathit{lfp}\ \underline{(X = 0 \to Y := 1 \lozenge (((X := X - 1; F);}$ $\underline{X := X - 1); Y := Y * X))}\ \underline{\{}\{s : s(X) = n \wedge s(Y) = n\,!\}\underline{\}}$	by j, k, (178.1)

■

Theorem (180) is not directly expressible in Hoare logic since ($\exists$ r $\in \mathcal{P}(S^2)$. $\underline{C_1}$(r) $\subseteq$ r $\wedge$ $\underline{\{}$p$\underline{\}}$r$\underline{\{}$q$\underline{\}}$) does not use only formulae of the style p' $\subseteq$ q' and $\underline{\{}$p'$\underline{\}}\underline{C'}\underline{\{}q'\underline{\}}$. To enforce this, we can let $S'=S^2$, p' = {<s, s> : s $\in S$}, $\underline{C_1'}$ = {<<$\underline{s}$, s>, <$\underline{s}$, s'>> : <s, s'> $\in \underline{C_1}$(r)} and q' = r so that $\underline{\{}$p'$\underline{\}}\underline{C_1'}\underline{\{}q'\underline{\}}$ = $\underline{\{}${<s, s'> : s $\in S$}$\underline{\}}$ {<<$\underline{s}$, s>, <$\underline{s}$, s'>> : <s, s'> $\in \underline{C_1}$(r)} $\underline{\{}$r$\underline{\}}$ = (p' $\rceil$ $\underline{C_1'} \subseteq S \times$ q') = ({<<$\underline{s}$, $\underline{s}$>, <$\underline{s}$,s>> : <s, s> $\in \underline{C_1}$(r)} $\subseteq$ {<<s_0, s_1>, <$\underline{s}$, s>> : <$\underline{s}$, s> $\in$ r}) = $\underline{C_1}$(r) $\subseteq$ r. In this translation the relationship r between states before and after the procedure call is expressed using predicates upon states but the state space S has been changed into S^2 (as in MANNA & PNUELI [1974]). To remain in the spirit of traditional Hoare logic, it is better to use logical auxiliary variables not appearing in the program to memorize the value of the programming variables before the procedure call (the importance of these auxiliary variables in correctness proofs for recursive procedures was first realized by GREIBACH [1975] and GORELICK [1975] and later thoroughly investigated in GALLIER [1978] [1981], APT, BERGSTRA & MEERTENS [1979], APT & MEERTENS [1980], MEYER & HALPERN [1980] and APT [1981b]). Let $\check{X}$ = <X_1, …, X_n> be the vector of variables $Var(C_1)$ = {X_1, …, X_n} appearing in command C_1. Following OLDEROG [1983], we write $\check{X} \parallel \check{x}$ whenever $\check{X}$ = <X_1, …, X_n>, $\check{x}$ = <x_1, …, x_m>, m = n and {X_1, …, X_n} $\cap$ {x_1, …, x_m} = ø and extend all notations to vectors, for example $\check{X} = \check{x}$ means $X_1 = x_1 \wedge \ldots \wedge X_n = x_m$, s($\check{X}$) stands for <s($X_1$),…,s($X_n$)>, s[$\check{X} \leftarrow \check{x}$] means s[$X_1 \leftarrow x_1$]…[$X_n \leftarrow x_n$], {$\check{X}$} is {$X_1$, …, X_n}, etc. Then we let p' = {s $\in S$: s($\check{X}$) = s($\check{x}$)} and q' = {s' $\in S$: <s'[$\check{X} \leftarrow$ s'($\check{x}$)], s'> $\in$ r} where {$\check{X}$} = $Var(C_1)$ and $\check{X} \parallel \check{x}$ so that $\underline{\{}$p'$\underline{\}}\underline{C_1}(r)\underline{\{}q'\underline{\}}$ implies $\underline{C_1}$(r) $\subseteq$ r. More precisely, we have:

DEFINITION OLDEROG [1983] *Non-interference* (182)

$\check{X} \parallel \check{x}$ = [$\check{X}$ = <X_1, …, X_n> $\wedge$ $\check{x}$ = <x_1, …, x_m> $\wedge$ m = n $\wedge$ {X_1, …, X_n} $\cap$ {x_1, …, x_m} = ø]

THEOREM *Partial correctness proof of procedures by fixpoint induction II* (183)

$\exists$ r $\in \mathcal{P}(S^2)$. $\underline{C_1}$(r) $\subseteq$ r $\wedge$ $\underline{\{}$p$\underline{\}}$r$\underline{\{}$q$\underline{\}}$

$\Leftrightarrow$

$\exists$ p', q' $\in \mathcal{P}(S)$.
$\forall$ r' $\in \mathcal{P}(S^2)$.
($\forall$ <s, s'> $\in$ r'. $\forall$ v $\notin Var(C_1)$. s(v) = s'(v)
$\wedge$ $\forall$ d $\in D$. <s[v $\leftarrow$ d], s'[v $\leftarrow$ d]> $\in$ r')
$\Rightarrow$ ($\underline{\{}$p'$\underline{\}}$r'$\underline{\{}$q'$\underline{\}}$ $\Rightarrow$ $\underline{\{}$p'$\underline{\}}\underline{C_1}(r')\underline{\{}q'\underline{\}}$)
$\wedge$ $\forall$ s $\in$ p. $\forall \check{d} \in \check{D}$. ($\forall$ s' $\in S$. s'[$\check{X} \leftarrow$ s($\check{X}$)] $\in$ p'
$\Rightarrow$ s'[$\check{X} \leftarrow \check{d}$] $\in$ q') $\Rightarrow$ s[$\check{X} \leftarrow \check{d}$] $\in$ q

where {$\check{X}$} = $Var(C_1)$ and $\check{X} \parallel \check{x}$

(The assertion $\forall$ <s, s'> $\in$ r'. $\forall$ v $\notin Var(C_1)$. s(v) = s'(v) $\wedge$ $\forall$ d $\in D$. <s[v $\leftarrow$ d], s'[v $\leftarrow$ d]> $\in$ r' states that the relational semantics r' of C_1 is without side-effects, more precisely that the variables v not appearing in C_1 cannot be modified and can have any value during execution of C_1. We have $\{p'\}\mathit{lfp}\,\underline{C_1}\{q'\}$ so that in the assertion $\forall s \in p.\ \forall \check{d} \in \check{D}.\ (\forall s' \in S.\ s'[\check{X} \leftarrow s(\check{X})] \in p' \Rightarrow s'[\check{X} \leftarrow \check{d}] \in q') \Rightarrow s[\check{X} \leftarrow \check{d}] \in q$, the assumption $\forall s' \in S.\ s'[\check{X} \leftarrow s(\check{X})] \in p' \Rightarrow s'[\check{X} \leftarrow \check{d}] \in q'$ states that $<p', q'>$ is the specification of the procedure Pn :: C_1. More precisely any terminating execution of C_1 started with initial values $s(\check{X})$ of the variables $\check{X}$ satisfying p' must terminate with final values $\check{d}$ of $\check{X}$ satisfying q' and this whatever the possible values s'(v) of the variables $v \notin \{\check{X}\}$ not appearing in the procedure body C_1 may be. The assertion states that any execution of Pn started with values $s(\check{X})$ of $\check{X}$ and terminated with values $\check{d}$ of $\check{X}$ satisfying specification $<p', q'>$ must satisfy the postcondition q whenever the precondition p is satisfied.)

Proof

- For $\Rightarrow$, assume $\underline{C_1}(r) \subseteq r \wedge \{p\}r\{q\}$ then $\{p\}\mathit{lfp}\,\underline{C_1}\{q\}$ by (180.1) so that by (180.2) we can assume that $\forall <s, s'> \in r.\ \forall v \notin \mathit{Var}(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ <s[v \leftarrow d], s'[v \leftarrow d]> \in r$. Let $p' = \{s \in S : s(\check{X}) = s(\check{x})\}$ and $q' = \{s' \in S : <s'[\check{X} \leftarrow s'(\check{x})], s'> \in r\}$ where $\{\check{X}\} = \mathit{Var}(C_1)$ and $\check{X} \parallel \check{x}$.

(A) Assuming $\forall <s, s'> \in r'.\ \forall v \notin \mathit{Var}(C_1).\ s(v) = s'(v) \wedge \forall d \in D.\ <s[v \leftarrow d], s'[v \leftarrow d]> \in r'$ and $\{p'\}r'\{q'\}$ we first prove $\{p'\}\underline{C_1}(r')\{q'\}$.

(a) We first prove that $p' \rceil \underline{C_1}(r') \subseteq p' \rceil \underline{C_1}(r)$. We have $(p' \rceil r' \subseteq S \times q')$ [by (124)] so that $\forall s, s' \in S.\ (s(\check{X}) = s(\check{x}) \wedge <s, s'> \in r') \Rightarrow <s'[\check{X} \leftarrow s'(\check{x})], s'> \in r$. It follows that if $<s, s'> \in r'$ then $<s[\check{x} \leftarrow s(\check{X})], s'[\check{x} \leftarrow s(\check{X})]> \in r'$ [since $\forall v \notin \mathit{Var}(C_1).\ \forall d \in D.\ <s[v \leftarrow d], s'[v \leftarrow d]> \in r'$] in which case $s[\check{x} \leftarrow s(\check{X})](X) = s[\check{x} \leftarrow s(\check{X})](\check{x}) = s(\check{X})$ [by (121) since $\check{X} \parallel \check{x}$ hence $\{\check{X}\} \cap \{\check{x}\} = \emptyset$] so that $<s'[\check{x} \leftarrow s(\check{X})][\check{X} \leftarrow s'[\check{x} \leftarrow s(\check{X})](\check{x})], s'[\check{x} \leftarrow s(\check{X})]> \in r$ and, after simplification by (121), $<s'[\check{x} \leftarrow s(\check{X})][\check{X} \leftarrow s(\check{X})], s'[\check{x} \leftarrow s(\check{X})]> \in r$ hence $<s, s'> \in r$ since $\forall v \notin \mathit{Var}(C_1).\ s'(v) = s(v)$. We conclude $r' \subseteq r$, whence by monotony $\underline{C_1}(r') \subseteq \underline{C_1}(r)$ so that $p' \rceil \underline{C_1}(r') \subseteq p' \rceil \underline{C_1}(r)$.

(b) Then we prove that $\{p'\}\underline{C_1}(r)\{q'\}$ is true. We have $\{p'\}r\{q'\} = (p' \rceil r \subseteq S \times q')$ [by (124)] $= (\forall <s, s'> \in r : s(\check{X}) = s(\check{x}) \Rightarrow <s'[\check{X} \leftarrow s'(\check{x})], s'> \in r\ )$ which is true since $<s, s'> \in r$ implies $<s'[\check{X} \leftarrow s(\check{X})], s'> \in r$ [since $\forall v \notin \mathit{Var}(C_1).\ \forall d \in D.\ <s[v \leftarrow d], s'[v \leftarrow d]> \in r$] hence $<s'[\check{X} \leftarrow s(\check{x})], s'> \in r$ [since $s(\check{X}) = s(\check{x})$]. We conclude $\{p'\}\underline{C_1}(r)\{q'\}$ since $\underline{C_1}(r) \subseteq r$.

(c) Finally, $\{p'\}\underline{C_1}(r')\{q'\}$ is true since (b) implies $(p' \rceil \underline{C_1}(r) \subseteq S \times q')$, whence $p' \rceil \underline{C_1}(r') \subseteq S \times q'$ by (a) so that $\{p'\}\underline{C_1}(r')\{q'\}$ holds.

(B) Let be given $s \in p$ and $\check{d} \in \check{D}$. Assuming $(\forall s' \in S.\ s'[\check{X} \leftarrow s(\check{X})] \in p' \Rightarrow s'[\check{X} \leftarrow \check{d}] \in q')$ we prove that $s[\check{X} \leftarrow \check{d}] \in q$. We have $s[\check{x} \leftarrow s(\check{X})](\check{X}) = s[\check{x} \leftarrow s(\check{X})](\check{x}) = s(\check{X})$ [by (121) since $\check{X} \parallel \check{x}$ hence $\{\check{X}\} \cap \{\check{x}\} = \emptyset$] so that $s[\check{X} \leftarrow s(\check{X})] \in p'$. By definition of p', this implies $s(\check{X}) = s[\check{X} \leftarrow s(\check{X})](\check{X}) = s[\check{X} \leftarrow s(\check{X})](\check{x}) = s(\check{x})$ [since $\check{X} \parallel \check{x}$]. Also $s[\check{X} \leftarrow s(\check{X})] \in p'$ implies $s[\check{X} \leftarrow \check{d}] \in q'$ that is $<s[\check{X} \leftarrow \check{d}][\check{X} \leftarrow s[\check{X} \leftarrow \check{d}](\check{x})], s[\check{X} \leftarrow \check{d}]> \in r$, after simplification by

(121), $<s[\check{X} \leftarrow s(\check{x})], s[\check{X} \leftarrow \check{d}]> \in r$ hence $<s[\check{X} \leftarrow s(\check{X})], s[\check{X} \leftarrow \check{d}]> \in r$ [since $s(\check{X}) = s(\check{x})$] so that $<s, s[\check{X} \leftarrow \check{d}]> \in r$ [by (121)] whence $s[\check{X} \leftarrow \check{d}] \in q$ [since $s \in p$ and $\underline{\{}p\underline{\}}r\underline{\{}q\underline{\}}$].

- For $\Leftarrow$, we have $\forall r' \in \mathcal{P}(S^2). (\forall <s, s'> \in r'. \forall v \notin Var(C_1). s(v) = s'(v) \wedge \forall d \in D. <s[v \leftarrow d], s'[v \leftarrow d]> \in r') \Rightarrow (\underline{\{}p'\underline{\}}r'\underline{\{}q'\underline{\}} \Rightarrow \underline{\{}p'\underline{\}}\underline{C}_1(r')\underline{\{}q'\underline{\}})$ and (175) which imply $\underline{\{}p'\underline{\}}\mathit{lfp}\ \underline{C}_1\underline{\{}q'\underline{\}}$. Hence by (180.2) there exists $r \in \mathcal{P}(S^2)$ such that $\underline{C}_1(r) \subseteq r \wedge \underline{\{}p'\underline{\}}r\underline{\{}q'\underline{\}} \wedge \forall <s, s'> \in r. \forall x \notin Var(C_1). (s(x) = s'(x)) \wedge \forall d \in D. <s[x \leftarrow d], s'[x \leftarrow d]> \in r$.

  To prove $\underline{\{}p\underline{\}}r\underline{\{}q\underline{\}}$ we assume that $s \in p$ and $<s, s'> \in r$ so that $s' = s[\check{X} \leftarrow s'(\check{X})]$ since $\forall x \notin \{\check{X}\}. s(x) = s'(x)$. Also $\forall s'' \in S. <s''[\check{X} \leftarrow s(\check{X})], s''[\check{X} \leftarrow s'(\check{X})]> \in r$ since $\forall x \notin \{\check{X}\}. \forall d \in D. <s[x \leftarrow d], s'[x \leftarrow d]> \in r$. Hence $\underline{\{}p'\underline{\}}r\underline{\{}q'\underline{\}}$ implies $\forall s'' \in S. s''[\check{X} \leftarrow s(\check{X})] \in p' \Rightarrow s''[\check{X} \leftarrow s'(\check{X})] \in q'$. Then $(\forall s'' \in S. s''[\check{X} \leftarrow s(\check{X})] \in p' \Rightarrow s''[\check{X} \leftarrow \check{d}] \in q')$ where $\check{d} = s'(\check{X})$ implies $s[\check{X} \leftarrow \check{d}] \in q$ that is $s[\check{X} \leftarrow s'(\check{X})] \in q$ so that, in conclusion, $s' \in q$. ■

Theorem (183) can be directly transcribed in Hoare logic using the recursion rule (176) and the following rule of adaptation (due to MORRIS [19??] and OLDEROG [1983a]) :

$$\frac{\{P'\}C\{Q'\}}{\{\forall \check{x}. (\forall \check{y}. (P' \Rightarrow Q'[\check{X} \leftarrow \check{x}])) \Rightarrow Q[\check{X} \leftarrow \check{x}]\}C\{Q\}} \quad \textit{Rule of adaptation} \qquad (184)$$

where $\check{X} \parallel \check{x}$ with $\{\check{x}\} \cap Free(P', Q', C, Q) = \emptyset$ and $\{\check{y}\} = Free(P', Q') - \{\check{X}\}$

A first version of this rule of adaptation was introduced by HOARE [1971b] and slightly extended in IGARASHI, LONDON & LUCKHAM [1975] and ERNST [1977]. OLDEROG [1983a] proved that it is sound and relatively complete and can replace the substitution rule I, the invariance rule and the elimination rule of APT [1981a]. Although HOARE [1971b] adaptation rule is relatively complete, MORRIS [19??] and OLDEROG [1983a] showed that it does not give the best possible precondition and proposed a strengthened version. An incorrect version of HOARE [1971b] adaptation rule was designed by GUTTAG, HORNING & LONDON [1978] and LONDON, GUTTAG, HORNING & LAMPSON [1978]. The error was pointed out by GRIES & LEVIN [1980] who proposed a sound version. A slightly simpler version of the adaptation rule of GRIES & LEVIN [1980] and GRIES [1981] was proposed by MARTIN [1983] but BIJLSMA, WILTINK & MATTHEWS [1986] showed that both versions were equivalent. As pointed out by OLDEROG [1983a] and BIJLSMA, WILTINK & MATTHEWS [1989] the rule of GRIES & LEVIN [1980] and GRIES [1981] is relatively incomplete. They proposed a sound and relatively complete variant. Another sound and relatively complete variant was designed by CARTWRIGHT & OPPEN [1978] [1981].

Example APT [1981a] *Recursive factorial procedure with global variables* (185)

To prove partial correctness { X = n } F { X = n ∧ Y = n ! } of procedure F of program (178) we use the recursion rule (176). Assuming {X = n} F {X = n ∧ Y = n !} by induction hypothesis, we must prove {X = n - 1} F {X = n - 1 ∧ Y = (n - 1) !} for the recursive call of F. By the rule of adaptation (184) where $\breve{X}$ = <X, Y>, $\breve{x}$ = <x, y> and $\breve{y}$ = <n> we derive {∀ x. ∀ y. (∀ n. ((X = n) ⇒ (x = n ∧ y = n !))) ⇒ (x = n - 1 ∧ y = (n - 1) !)} F {X = n - 1 ∧ Y = (n - 1) !}. By the consequence rule (102), it remains to show that (X = n - 1) ⇒ (∀ x. ∀ y. (∀ n'. ((X = n') ⇒ (x = n' ∧ y = n' !))) ⇒ (x = n - 1 ∧ y = (n - 1) !)) which is obviously true. Then the correctness proof of procedure F can be completed as shown by the following proof outline:

```
procedure F;
begin
  { X = n }
  if X = 0 then
     { X = n ∧ X = 0 } Y := 1 { X = n ∧ Y = n ! }
  else begin
     { X = n ∧ X ≠ 0 } X := X - 1; { X = n - 1 }
     F;
     { X = n - 1 ∧ Y = (n - 1) ! } X := X + 1; { X = n ∧ Y = (n - 1) ! } Y := Y * X;
  end;
  { X = n ∧ Y = n ! }
end;
{ X = y } F; { X = y ∧ Y = y ! }
```

For the main call of procedure F it remains to prove {X = y} F {X = y ∧ Y = y !}. Since (X = y) ⇒ (∀ x'. ∀ y'. (∀ n. ((X = n) ⇒ (x' = n ∧ y' = n !)) ⇒ (x' = y ∧ y' = y !))) this follows from the rule of adaptation (184) and the consequence rule (102). ■

Example DE BAKKER & MEERTENS [1975] *Showing the usefulness of auxiliary variables* (186)

The following program computes $2^{n_0} - 1$ when n_0 is positive:

```
procedure F;
begin
  { N = n ≥ 0 ∧ S = s }
  if N > 0 then begin
     { N = n > 0 ∧ S = s } N := N - 1; { N = n - 1 ≥ 0 ∧ S = s }
     F;
     { N = n - 1 ≥ 0 ∧ S = s + 2^(n-1) - 1 } S := S + 1; { N = n - 1 ≥ 0 ∧ S = s + 2^(n-1) }
     F;
     { N = n - 1 ≥ 0 ∧ S = s + 2^(n-1) + 2^(n-1) - 1 } N := N + 1;
  end;
  { N = n ≥ 0 ∧ S = s + 2^n - 1 }
end;
{ N = n0 ≥ 0 ∧ S = 0 } F; { N = n0 ≥ 0 ∧ S = 2^n0 - 1 }
```

Its partial correctness proof uses the recursion rule (176) with induction hypothesis $\{ N = n \geq 0 \wedge S = s \} F \{ N = n \geq 0 \wedge S = s + 2^n - 1 \}$ and the adaptation and consequence rules (184) and (102) for the main and recursive calls of F. For example $\{ N = n - 1 \geq 0 \wedge S = s \} F \{ N = n - 1 \geq 0 \wedge S = s + 2^{n-1} - 1 \}$ follows from the fact that $(N = n - 1 \geq 0 \wedge S = s)$ implies $(\forall\ n', s'. (\forall\ \underline{n}, \underline{s}. ((N = \underline{n} \geq 0 \wedge S = \underline{s}) \Rightarrow (n' = \underline{n} \geq 0 \wedge s' = \underline{s} + 2^{\underline{n}} - 1)) \Rightarrow (n' = n - 1 \geq 0 \wedge s' = s + 2^{n-1} - 1))$. The proof proposed in DE BAKKER & MEERTENS [1975] uses an infinite pattern of assertions (one for each program step) because in their proof system one cannot use logical variables (n, s) to relate the different values of the program variables (N, S) at different stages of the computation so that the invariants have to express the current value of N and S in terms of their initial values n_0 and 0 and of a representation of the computation history (see GREIBACH [1975], GORELICK [1975], GALLIER [1978] [1981], APT, BERGSTRA & MEERTENS [1979], APT & MEERTENS [1980], MEYER & HALPERN [1980] and APT [1981b]). ■

8.2.1.4 Hoare-like deductive systems with context-dependent conditions

Hoare deduction systems are Hilbert-style systems (BARWISE [1977]) where proofs proceed by induction on the syntax of programs. Since this syntax is specified by a context-free grammar (AHO, SETHI & ULLMAN [1986]) it is not directly possible to express context-dependent conditions such as the correspondence between procedure calls and procedure declarations (specifying which procedure bodies are associated with procedure names). In fact a formula {P}C{Q} is often relative to declarations of named objects which can be represented by a mapping from names to objects called an *environment* (SCOTT & STRACHEY [1972]). Therefore, most often, Hoare correctness formulae have the form $<\rho \mid \{P\}C\{Q\}>$ where ρ is an environment associating procedure bodies to procedure names (and informally all procedure names occurring free in C are declared in ρ and there are no procedure names occurring free in ρ). The axioms and rules of inference propagate the environment from declarations to calls:

(187)

$$\frac{< [Pn \leftarrow C_1] \mid \{P\}\ C_2\ \{Q\} >}{\{P\}\ Pn\ ::\ C_1;\ C_2\ \{Q\}}$$ *Rule of programs* (.1)

$< \rho \mid \{P\}\ \text{skip}\ \{P\} >$ *Skip axiom* (.2)

8.2.2 Value-result parameters

Let us now consider a procedural language with value-result parameters:

DEFINITION *Syntax of a procedural language with value and/or result parameters* (188)

Pn : $\mathbb{P}\mathrm{roc}$ *Procedure names* (.1)

Pg : $\mathbb{P}\mathrm{rog}$ *Programs* (.2)

Pg ::= Pn (? U ?! V ! W) :: C_1; C_2

C : $\mathbb{C}\mathrm{om}$ *Commands* (.3)

C ::= skip | X := E | X := ? | (C_1; C_2) | (B → C1 ◊ C2) | (B * C) | Pn (E, Y, Z)

U, V and W are distinct formal parameters of recursive procedure Pn with body C_1 which are respectively passed by value, value-result and result. Therefore a call Pn (E, Y, Z) with expression E and variables Y and Z as actual parameters consists in creating <u>new</u> local variables named U, V, W within C_1, respectively initialized to the value of E, to the value of Y and to any value d of *D*, then in executing the body C_1 of procedure Pn (which may modify the values of local variables U, V, W as well as that of the global variables (but not the ones named U, V and W which are hidden by their local homonyms)), and finally in assigning the values of the local variables V and W to the variables Y and Z (in that order) and then in destroying the local variables named U, V, W:

DEFINITION after DE ROEVER [1974] *Relational semantics of value and/or result parameter passing* (189)

- $\underline{\mathrm{Pn\ (E, Y, Z)}}$ (r) = (.1)

$$\{\langle s,\ s'[U \leftarrow s(U)][V \leftarrow s(V)][W \leftarrow s(W)][Y \leftarrow s'(V)][Z \leftarrow s'(W)]\rangle :$$
$$\exists\, d \in D.\ \langle s[U \leftarrow \underline{E}(s)][V \leftarrow s(Y)][W \leftarrow d],\ s'\rangle \in r\}$$

- $\underline{\mathrm{Pn\ (?\ U\ ?!\ V\ !\ W) :: C_1; C_2}} = \underline{C_2}(\mathit{lfp}\ \underline{C_1})$ (.2)

Example (190)

Let Pg be Pn(? A ?! B ! C) :: C_1; C_2 where C_1 is (B := A + B; (A := A + 1; (C := A + 1; D := C + 1))) and C_2 is (C_3; Pn(A + 1, B, B)) with C_3 = (A := 1; (B := 10; C := 100)). By (173.3) and (173.5), the relational semantics of the procedure body is $\mathit{lfp}\ \underline{C_1} = \underline{C_1}(\emptyset) = \{\langle s,\ s[A \leftarrow s(A) + 1]\ [B \leftarrow s(A) + s(B)]\ \ [C \leftarrow s(A) + 2]\ \ [D \leftarrow s(A) + 3]\rangle : s \in S\}$. By (189.1), the semantics of the procedure call is $\underline{\mathrm{Pn(A + 1, B, B)}} = \{\langle s,\ s'[A \leftarrow s(A)]\ \ [B \leftarrow s(B)]\ \ [C \leftarrow s(C)]\ \ [B \leftarrow s'(B)]\ \ [B \leftarrow s'(C)]\rangle : \exists\, d \in D.\ \langle s[A \leftarrow \underline{A + 1}(s)]\ \ [B \leftarrow s(B)]\ \ [C \leftarrow d],\ s'\rangle \in \mathit{lfp}\ \underline{C_1}\} = \{\langle s,\ s[D \leftarrow s(A) + 4]\ [B \leftarrow s(A) + 3]\rangle : s \in S\}$. The semantics of the initialization part of the program body is $\underline{C_3}(\mathit{lfp}\ \underline{C_1}) = \{\langle s,\ s[A \leftarrow 1]\ [B \leftarrow 10]\ [C \leftarrow 100]\rangle : s \in S\}$ so that the semantics of the program is $\underline{\mathrm{Pg}} = \underline{C_2}(\mathit{lfp}\ \underline{C_1}) = \{\langle s,\ s[A \leftarrow 1]\ [B \leftarrow 10]\ [C \leftarrow 100]\ [D \leftarrow (s[A{\leftarrow}1][B{\leftarrow}10][C{\leftarrow}100](A))$

$+ 4] [B \leftarrow (s[A\leftarrow 1][B\leftarrow 10][C\leftarrow 100](A)) + 3]> : s \in \mathcal{S}\} = \{<s, s[A \leftarrow 1] [C \leftarrow 100] [D \leftarrow 5] [B \leftarrow 4] > : s \in \mathcal{S}\}$. ■

This semantics leads to an analog of theorem (183) and then to the following proof rules (the rule of adaptation (192) could be simplified as in GRIES & LEVIN [1980], GRIES [1981], MARTIN [1983] and BIJLSMA, MATTHEWS & WILTINK [1989] by restricting the use of global variables, the assignments to value parameters, or assuming that value-result and result parameters are different):

Recursive procedure Pn (? U ?! V ! W) :: C_1;

Recursion rule (191)

$$\frac{\{P\}\ \mathbf{Pn\ (?\ U\ ?!\ V\ !\ W)}\ \{Q\} \vdash \{P\}\mathbf{C_1}\{Q\}}{\{P\}\ \mathbf{Pn\ (?\ U\ ?!\ V\ !\ W)}\ \{Q\}}$$

Rule of adaptation (192)

$$\frac{\{P'\}\quad \mathbf{Pn(?U,\quad ?!V,\quad !W)}\quad \{Q'\}}{\begin{array}{l}\{\forall\, \breve{x}.\ \forall\, “u, “v, “w, “y, “z, v', w'.\\ \quad ((U = “u) \wedge (V = “v) \wedge (W = “w) \wedge (Y = “y) \wedge (Z = “z) \wedge\\ \quad (\forall\, \breve{m}.\\ \quad\quad ((\exists ‘w.P'[U \leftarrow E[U\leftarrow “u][V\leftarrow “v][W\leftarrow “w][Y\leftarrow “y][Z\leftarrow “z]][V\leftarrow “v][W\leftarrow ‘w][Y\leftarrow “y]\\ [Z\leftarrow “z])\\ \quad\quad\quad \Rightarrow (\exists\ u', y', z'.\ Q'[\breve{X} \leftarrow \breve{x}]\ [U \leftarrow u']\ [V \leftarrow v']\ [W \leftarrow w']\ [Y \leftarrow y']\ [Z \leftarrow z']))))\\ \quad\quad \Rightarrow Q\ [Z \leftarrow w']\ [Y \leftarrow v']\ [\breve{X} \leftarrow \breve{x}]\ [U \leftarrow “u]\ [V \leftarrow “v]\ [W \leftarrow “w]\}\\ \mathbf{Pn(E,\ Y,\ Z)}\\ \{Q\}\end{array}}$$

where:

- $\breve{X} = Var(E, C_1) - \{U, V, W, Y, Z\}$ is the value of the global variables (not used as actual result parameters and not homonyms of actual or formal parameters) before the call Pn(E, Y, Z),
- $\breve{x}$ such that $\breve{X} \parallel \breve{x}$ are fresh variables denoting the value of $\breve{X}$ after the call Pn(E, Y, Z),
- “u, “v, “w, “y, “z are fresh variables denoting the values of the global variables U, V, W, Y, Z before the call,
- ‘w is a fresh variable denoting the undetermined initial value $d \in D$ of formal result parameter W when execution of the procedure body C_1 is started,
- u’, v’, w’ are the values of the local formal parameters U, V and W after execution of the procedure body C_1 and before the result parameters passing,

- $\{m\check{}\} = Free(P', Q') - (\{X\check{}\} \cup \{U, V, W, Y, Z\})$ is the set of logical variables used for the specification of the procedure body C_1,
- y', z' are the values of the global variables Y and Z after execution of the procedure body C_1 and before the result parameters passing,
- all fresh variables are distinct and do not appear in $Free(P', Q', C_1, E, Q) \cup \{U, V, W, Y, Z\}$.

When reading rule (192) it must be remembered that substitution is left to right that is to say $P[X \leftarrow x][Y \leftarrow y]$ is $P'[Y \leftarrow y]$ where $P' = P[X \leftarrow x]$.

Example MARTIN [1983] *Once used to show the inconsistency of erroneous procedure-* (193)
call proof rules

Assuming {P'} Pn(! Z) {Q'} where P' = true and Q' = ((Z = 1) ∨ (Z = 2)), we can determine P such that {P} Pn(C) {Q} holds with Q = (C = 2) by the rule of adaptation (192):

$$\begin{aligned} P &= \forall \text{“}w, \text{“}z, w'. ((Z = \text{“}w) \wedge (C = \text{“}z) \wedge ((\exists \text{‘}w.\ P'[Z \leftarrow \text{‘}w][C \leftarrow \text{“}z]) \\ &\qquad\qquad \Rightarrow (\exists z'.\ Q'[Z \leftarrow w'][C \leftarrow z']))) \Rightarrow Q\ [C \leftarrow w']\ [Z \leftarrow \text{“}w] \\ &= \forall \text{“}w, \text{“}z, w'. ((Z = \text{“}w) \wedge (C = \text{“}z) \wedge ((w' = 1) \vee (w' = 2))) \Rightarrow (w' = 2) \\ &= \text{false}. \end{aligned}$$

MARTIN [1983] gives David Gries the credit for this example who used it to demonstrate the inconsistency of a number of procedure-call proof rules. The Euclid proof rule (LONDON, GUTTAG, HORNING, LAMPSON, MITCHELL & POPEK [1978]), for example, gives P = true. ■

Example BIJLSMA, MATTHEWS & WILTINK [1989] *On the use of auxiliary variables* (194)

Consider a procedure Pn(? X ! Z) for rounding the real number X to a nearby integer Z (such as Z := floor(X), Z := ceil(X) or Z = round(X)). The specification {P'}Pn{Q'} where $P' = (m \le X \le m + 1)$ and $Q' = (Z = m \vee Z = m + 1)$ states that Z is obtained from X by rounding any non-integer X either up or down to an integer, and by using X itself if X happens to be an integer. Now consider the call Pn(A, C) with postcondition Q = (C = 0). The corresponding precondition P such that {P} Pn(A, C) {Q} holds is given by the rule of adaptation (192) as:

$$\begin{aligned} P &= \forall \text{“}u, \text{“}w, \text{“}z, w'. \\ &\quad ((X = \text{“}u) \wedge (Z = \text{“}w) \wedge (C = \text{“}z) \wedge \\ &\quad\ (\forall m. \\ &\qquad ((\exists \text{‘}w.\ P'[X \leftarrow A[X \leftarrow \text{“}u][Z \leftarrow \text{“}w][C \leftarrow \text{“}z]]\ [Z \leftarrow \text{‘}w]\ [C \leftarrow \text{“}z]) \\ &\qquad\quad \Rightarrow (\exists u', y', z'.\ Q'[X \leftarrow u']\ [Z \leftarrow w']\ [C \leftarrow z'])))) \\ &\quad\ \Rightarrow Q\ [C \leftarrow w']\ [X \leftarrow \text{“}u]\ [Z \leftarrow \text{“}w] \\ &= \forall w'. (\forall m. ((m \le A \le m + 1) \Rightarrow (w' = m \vee w' = m + 1))) \Rightarrow (w' = 0) \\ &= (A = 0). \end{aligned}$$

This example was designed by BIJLSMA, MATTHEWS & WILTINK [1989] to show that the precondition in GRIES [1981, Th. 12.4.1 p. 161]'S rule (which would be P = false) is not the weakest that can be inferred solely from the procedure's specification in cases when the specification involves auxiliary logical variables. ■

Example *Homonym formal and actual parameters* (195)

Let Pg be Pn(? A ?! B ! C) :: C_1; C_2 where C_1 is (B := A + B; (A := A + 1; (C := A + 1; D := C + 1))) and C_2 is (C_3; Pn(A + 1,B, B)) with C_3 = (A := 1; (B := 10; C := 100)). By (97), the composition rule (99) and the consequence rule (102) we have {A = a ∧ B = b ∧ C = c} C_1 {A = a + 1 ∧ B = a + b ∧ C = a + 2 ∧ D = a + 3} whence by the recursion rule (191 we derive {P'} Pn (? A ?! B ! C) {Q'} where P' is (A = a ∧ B = b ∧ C = c) and Q' is (A = a + 1 ∧ B = a + b ∧ C = a + 2 ∧ D = a + 3). Let Q be (A = a ∧ B = b ∧ C = c ∧ D = d). In order to derive the corresponding precondition P by the rule of adaptation (192) we observe that X˘ = <D>, m˘ = <a, b, c> whence:

P = (∀ x. ∀ “u, “v, “w, “y, “z, v’, w’.
((A = “u) ∧ (B = “v) ∧ (C = “w) ∧ (B = “y) ∧ (B = “z) ∧
(∀ a, b, c.
((∃‘w. P'[A ← E[A←“u][B←“v][C←“w][B←“y][B←“z]] [B←“v] [C←‘w] [B←“y] [B←“z])
⇒ (∃ u’, y’, z’. Q'[D ← x] [A ← u’] [B ← v’] [C ← w’] [B ← y’] [B ← z’]))))
⇒ Q[B ← w’] [B ← v’] [D ← x] [A ← “u] [B ← “v] [C ← “w])
= (A = a = b - 3 = d - 4 ∧ C = c). ■

Example *Recursive factorial procedure with value-result parameters* (196)

Let us prove the partial correctness of the following program:

```
procedure F(?X !Y);
begin
   {X = n}
   if X = 0 then
      {X = n ∧ X = 0} Y := 1
   else begin
      {X = n ∧ X ≠ 0} F(X - 1, Y); {X = n ∧ Y = (n - 1) !} Y := Y * X;
   end;
   {X = n ∧ Y = n !}
end;
{X = y} F(X, Y); {X = y ∧ Y = y !}
```

To prove partial correctness {X = n} F(?X !Y) {X = n ∧ Y = n !} of procedure F, we use the recursion rule (191). Assuming {X = n} F(?X !Y) {X = n ∧ Y = n !} by induction hypothesis, we must prove {X = n ∧ X ≠ 0} F(X -1, Y) {X = n ∧ Y = (n - 1) !} for the recursive call of F. By the consequence rule (102) and rule of adaptation

(192) (where {x˘} = {x˘} = ø and m˘ = n, P' = (X = n), Q' = (X = n ∧ Y = n !) and Q = (X = n ∧ Y = (n - 1) !)) we must show that:

(X = n ∧ X ≠ 0) ⇒
  (∀ “u, “w, “z, w’.
    ((X = “u) ∧ (Y = “w) ∧ (Y = “z)
    ∧ (∀ n.((∃‘w. P'[X ← (X - 1)[X←“u][Y←“w][Y←“z]] [Y ← ‘w] [Y ← “z])
        ⇒ (∃ u’, z’. Q'[X ← u’] [Y ← w’] [Y ← z’]))))
  ⇒ Q [Y ← w’] [X ← “u] [Y ← “w])

that is, after simplification:

(X = n ∧ X ≠ 0) ⇒ (∀ “u, w’. ((X = “u) ∧ (w’ = (“u - 1) !)) ⇒ (“u = n ∧ w’ = (n - 1) !))

which is obvious. Then the correctness proof of procedure F can be completed as shown by the above proof outline. Knowing that {X = n} F(?X !Y) {X = n ∧ Y = n !} holds, it remains to prove {X = y} F(X, Y) {X = y ∧ Y = y !} for the main call of procedure F. This follows from the rule of adaptation (192) and the consequence rule (102) since: (X = y) ⇒ (∀ “u, “w, w’. ((X = “u) ∧ (Y = “w) ∧ (∀ n. ((∃ ‘w. (X = n)[X ← “u] [Y ← ‘w]) ⇒ (∃ u’. (X = n ∧ Y = n !)[X ← u’] [Y ← w’])))) ⇒ (X = y ∧ Y = y !)[Y ← w’] [X ← “u]). ■

8.2.3 Complements on variable parameters and procedures as parameters

APT [1981a]'s survey on Hoare logic is mainly concerned with procedures and parameter mechanisms. More recent progresses concerning procedures have been reported in LANGMAACK & OLDEROG [1980] and OLDEROG [1983b]. In particular, for a treatment of variable (reference, address,…) parameters, see HOARE [1971b], IGARASHI, LONDON & LUCKHAM [1975], ERNST [1977], SCHWARTZ [1979], DE BAKKER [1980, Ch. 9], GRIES & LEVIN [1980], APT [1981a], CARTWRIGHT & OPPEN [1981], GERMAN, CLARKE & HALPERN [1983] [1988], SIEBER [1985]. A separation of procedural abstraction and parameter passing is attempted is MORGAN [1988a] [1988b] [1988c].

CLARKE [1977]'s theorem (169) put a borderline to sound and relatively complete Hoare logics for programs with procedures. CLARKE [1977] also claimed that the languages $\mathbb{L}_j$ (as defined at (166)) do have a sound and relatively complete Hoare logic. For j ≠ 4 these claims were either proved in CLARKE [1977] or later established by OLDEROG [1981]. These languages $\mathbb{L}_j$, j ≠ 4 are easy to axiomatize since for each program, there is a bound on the number of procedure environments (associating a procedure body to a procedure name) that can be reached during execution (OLDEROG [1983b], DAMM & JOSKO [1983a] [1983b]). It follows that each of the procedure environments can be treated as a separate case. The case of $\mathbb{L}_4$ was more difficult because it can give rise to non-homomorphic chains of procedure environments that grow arbitrarily long.

Example (197)

In the following program written in an $\mathbb{L}_4$ subset of Pascal:

```
program L4;
     procedure P (X : integer; procedure Q (Z : integer));
       procedure L (X : integer); begin Q(X - 1) end;
     begin if X > 0 then P(X - 1, L) else Q(X) end;

     procedure M;
       var N : integer;
       procedure R (X : integer); begin writeln(X) end;
     begin write('N = '); readln(N); P(N, R) end; {M}
begin M end.
```

the main procedure M reads the value n of N and calls P(n, L^n) with L^n = R which recursively calls P(n - 1, L^{n-1}), P(n - 2, L^{n-2}), … , P(1, L^1) and P(0, L^0) thus creating a chain of procedures **procedure** L^{i-1} (X : integer); **begin** L^i(X - 1) **end**; for i = n, … , 1 so that execution of P(0, L^0) consists in calling L^0(0) whence L^1(-1), L^2(-2), …, L^{n-1}(-n+1) and L^n(-n) that is R(-n) which finally prints -n. ■

The first axiomatizations of Clarke's language $\mathbb{L}_4$ proposed by DAMM & JOSKO [1983a] [1983b] and OLDEROG [1983c] [1984] used higher-order assertion languages to make assertions about this unbounded number of procedure environments (i.e. state-transformation): all concepts for Hoare logic are lifted from programs transforming states into states to programs transforming state-transformations into state-transformations. This leads to completeness not relative to the first-order theory of the interpretation. GERMAN, CLARKE & HALPERN [1983] [1988] were the first to propose a relative completeness proof for $\mathbb{L}_4$ with a first-order oracle and COOK [1978]'s notion of expressiveness. However the logic extends Hoare logic by allowing quantifiers over first-order variables (∀ x. H, …) and other logical connectives (H ⇒ H',…) to be used on the level of Hoare formulae (an idea going back to HAREL, PNUELI & STAVI [1977] and also exploited in SIEBER [1985]) but is relatively complete only for Hoare correctness formulae H. Later GOERDT [1987] proposed an indirect axiomatic proof method for $\mathbb{L}_4$ which involves the translation of programs into finitely typed lambda calculus and then the application of GOERDT [1985]'s Hoare calculus. This calculus was shown to be relatively complete (without Herbrand definability hypothesis) in GOERDT [1988]. Hoare logic has also been extended to other types of languages with higher-order concepts in ERNST, NAVLAKHA & OGDEN [1982], DE BAKKER, KLOP & MEYER [1982], GOERDT [1985] [1987] [1988], HALPERN [1984], LANGMAACK [1983], TRAKHTENBROT, HALPERN & MEYER [1983].

8.3 Undefinedness

When the evaluation of expressions in the programming language can lead to runtime errors, it is necessary to deal with partially defined functions. Hoare logic can be easily extended when the domain of these partial functions is given (for example in the case of array bounds outside their declared range) or can be easily defined (for example in the case of dangling pointers) as shown by SITES [1974], GERMAN [1978] [1981], COLEMAN & HUGUES [1979] and BLIKLE [1981]. Contrary to mathematicians, the computer scientists are often confronted with the problem of proving properties about partial objects without the knowledge of their domain. In this case first-order 2-valued logic is inadequate as pointed out by ASHCROFT, CLINT & HOARE [1972] who have discovered an error in CLINT & HOARE [1976] for nonterminating functions. Alternatives consist in introducing an extra element $\perp$ in the data domain which is forced by the axiomatization to behave like a "divergent" or "undefined" data object (CARTWRIGHT [1984]) or, more generally, in using a 3-valued logic covering undefinedness in program proofs as proposed by BARRINGER, CHENG & JONES [1984] and studied by HOOGEWIJS [1987].

8.4 Aliasing and side effects

The assignment axiom (97) is incorrect when aliases (i.e. distinct identifiers designating a shared storage location) are allowed. One can hide the source of the problem by prohibiting interference between identifiers in the programming language or in the proofs (REYNOLDS [1978] [1989], MASON [1987]) or consider augmented versions of Hoare logic which allows aliasing between variables (JANSSEN & VAN EMDE BOAS [1977], SCHWARTZ [1979], CARTWRIGHT & OPPEN [1981], OLDEROG [1981], LANGMAACK [1983], TRAKHTENBROT, HALPERN & MEYER [1983]).

The same way the assignment axiom (97), conditional (100) and while (101) rules are incorrect when the evaluation of expressions can have side effects (MANNA & WALDINGER [1981]). A number of Hoare-like axiomatizations of expression languages or expressions with side effects (CUNNINGHAM & GILFORD [1976], KOWALTOWSKI [1977], PRITCHARD [1977], SCHWARTZ & BERRY [1979]) modify Hoare logic by introducing Hoare correctness formulae for expressions $\{P\}E\{Q\}$ where the precondition P and postcondition Q explicitly depend upon a distinguished variable standing for the value returned by the evaluation of expression E. Alternatives consist in considering other logics explicitly referring to the state of computation (MANNA & WALDINGER [1981]), in using the value of a programming language expression as the underlying primitive (BOEHM [1985]) or in transforming the program into procedural form (LANGMAACK [1983]).

8.5 Block structured local variables

If we extend the syntax (188) of programs with block structured local variables:

$$C ::= (\textbf{var } X;\ C_1) \tag{198}$$

local variable X is like a bound variable of a predicate (see (116)) so that blocks satisfy the property that systematic replacement of the local variable X by some fresh variable Y preserves the meaning of the block: (**var** X; C) is equivalent to (**var** Y; $C[X \leftarrow Y]$), provided that Y does not occur free in C. For example, according to this *static scope* rule, (**var** X; ((**var** X; C_1); C_2)) is equivalent to (**var** Y; ((**var** Z; $C_1[X \leftarrow Z]$); $C_2[X \leftarrow Y]$)) where $Y, Z \notin Var(C_1, C_2)$. This leads to the following rule (HOARE [1971b], HOARE & WIRTH [1973], APT [1978] [1981a], DE BAKKER [1980]) where the renaming of X for Y is performed to distinguish between the occurrences of local X in C and possible free occurrences of nonlocal X in P or Q:

$$\frac{\{P\}\ C[X \leftarrow Y]\ \{Q\}}{\{P\}\ (\textbf{var } X;\ C)\ \{Q\}} \qquad \textit{Block rule} \tag{199}$$

where $Y \notin Free(P, Q) \wedge ((X = Y) \vee Y \notin Free(C))$

Observe that rule (199) may necessitate alteration $C[X \leftarrow Y]$ of the program text C, but this can be avoided (naïve solutions such as DONAHUE [1976] fail to treat the scope rule properly, see the discussion in APT [1981a] and FOKKINGA [1978], OLDEROG [1981] [1983] for further details). Observe also that rule (199) does not take the possibility of running out of new storage locations into account (TRAKHTENBROT, HALPERN & MEYER [1983]). There are other difficulties in defining the semantics of such blocks (see DE BAKKER [1980, Ch. 6], TRAKHTENBROT, HALPERN & MEYER [1983], HALPERN, MEYER & TRAKHTENBROT [1984], MEYER & SIEBER [1988], for a discussion). For example, mapping local variables into a global memory using a stack discipline is an overspecification, since then, for example, {true} ((**var** X; X := 0); (**var** Z; Y := Z)) {Y = 0} would hold because X and Z are allocated at the same address. Although this phenomenon can be observed in a number of implementations, it is contrary to the specification of block-structured languages where the initial value of local variables is usually *undetermined* so that, as shown by (199), (**var** X; C) should be equivalent to (**var** X; (X := ?; C)), see WIRTH [1971] for example. However this introduces unbounded nondeterminism (see the corresponding difficulties in paragraph § 8.10.2). Another way to cope with uninitialized local variables (APT [1981a]) is to use an extra uninitialized value $\omega \in D$ so that (**var** X; C) is equivalent to (**var** X; (X := ω; C)). But then one can prove {true} ((**var** X; Y := X); (**var** X; Z := X)) {X = Y}, which is not true for most implementations where the initial value of local variables is undetermined. DE BAKKER [1980] introduces simple conditions which guarantee that variables are initialized before being used. One can also force initialization upon declaration with the syntax (**var** X := E; C_1) (and use union types to allow for an initial value denoting logical uninitialization).

8.6 Goto statements

Goto's with static labels cause no problem with FLOYD [1967a] partial correctness proof method. For each label L, if $\{P_i, i = 1, \ldots, n\}$ is the set of preconditions of the statements **goto** L in the program (including the postcondition of the command sequentially preceding the command labeled L) and Q is the precondition of the command labeled L then FLOYD [1967a]'s verification condition is simply $(\cup_{i = 1,\ldots,n} P_i) \Rightarrow Q$. The difficulty with Hoare logic is to express this verification conditional compositionally, by induction on the syntax of programs (O'DONNELL [1982]). Various solutions have been proposed by CLINT & HOARE [1972], DONAHUE [1976], WANG [1976], KOWALTOWSKI [1977], DE BRUIN [1981] and LIFSCHITZ [1984]. An inconsistency problem with CLINT & HOARE [1972], DONAHUE [1976] and KOWALTOWSKI [1977] goto rules is noticed by ARBIB & ALAGIC [1979] and O'DONNELL [1982]. Scope problems with jumps out of procedures have not been explicitly dealt with (but for weakened forms of procedure escapes, FOKKINGA [1978]).

Techniques similar to that used in theorem (169) have also been used to obtain incompleteness results for programming languages that include label variables with retention (CLARKE [1977] [1984]).

8.7 Functions and expressions (with side effects)

Functions are not called in order to change states - the realm that assertions can capture - but to return a value. In the proof rule proposed by CLINT & HOARE [1972] the result returned by the function call f(x) (in the computer science sense) is simply denoted in predicates by f(x) (in the mathematical sense). This excludes side-effects in functions (since for example the mathematical identity f(x) + f(x) = 2 f(x) is not valid whenever a call to f increments x by 1). Moreover ASHCROFT, CLINT & HOARE [1976] have noticed that the proof rule of CLINT & HOARE [1972] yields an inconsistency whenever a defined function fails to halt for some possible argument, even if the value of the function is never computed for that argument, as explained in O'DONNELL [1982]. Hence the problem is again the one of undefinedness which is not correctly handle in predicates (see 8.3).

The problem can be circumvented by reduction of expression evaluation to execution of a sequence of assignment statements (DE BAKKER, KLOP & MEYER [1982]), by transforming the programmer-declared functions into procedures (LANGMAACK [1983]) or by modifying Hoare logic by explicitly introducing one or more symbols to denote expression values (CUNNINGHAM & GILFORD [1976], KOWALTOWSKI [1977], PRITCHARD [1977], SCHWARTZ [1979]) or by introducing primitive notations to make explicit assertions about the value of programming language expressions (BOEHM [1982] [1985], SOKOLOWSKI [1984]).

8.8 Coroutines

CLINT [1973] extended Hoare logic to simple coroutines (block-body and single coroutine combination) based upon the semi-coroutine concept of Simula 67 (WANG & DAHL [1971], DAHL & NYGAARD [1966]). It was further extended by PRITCHARD [1976] to multiple coroutines and DAHL [1975] to multiple dynamic instances of coroutines. They use auxiliary variables to accumulate the computation and communication history in stacks of arbitrary size. CLARKE [1980] showed that, in the case of simple static coroutines, history variables are useless and that auxiliary variables of bounded size simulating program counters are enough. CLINT [1981] argues that history variables, although not necessary, can help for clarity and ease of verification.

Techniques similar to that used in theorem (169) have also been used to obtain incompleteness results for programming languages that include coroutines with local recursive procedures that can access global variables (CLARKE [1977] [1984]).

8.9 Parallel programs

The controversy on the usefulness of program verification (DE MILLO, LIPTON & PERLIS [1979]) can hardly be extended to parallel programs because their correctness which is often very intricate cannot be checked by non-reproducible and non-exhaustive tests. Therefore, clear programming notations as well as correctness proofs are indispensable, at least when designing the underlying basic algorithms (DIJKSTRA [1968], ANDREWS [1981]).

The evolution of Hoare logic for parallel programs is tightly coupled with the slow emergence of clear notations for expressing process synchronization and communication to which C. A. R. Hoare largely contributed, from shared variables (HOARE [1972c] [1975], OWICKI & GRIES [1976a] [1976b], ...), then monitors (HOARE [1974], HOWARD [1976], OWICKI [1978]) and finally synchronous message passing (HOARE [1978b], APT, FRANCEZ & DE ROEVER [1980], LEVIN & GRIES [1981], …).

We discuss proof methods for parallel programs with shared variables and briefly survey the case of synchronous message passing in HOARE [1978b]'s communicating sequential processes CSP.

8.9.1 Operational semantics of parallel programs with shared variables

We consider (a simplified version of) parallel programs with shared global variables as introduced by OWICKI [1975] and OWICKI & GRIES [1976a]:

DEFINITION *Syntax of parallel programs* (200)

. Pp : $\mathbb{Papr}$ *Parallel programs* (.1)

Pp ::= $[C_1 \parallel C_2 \parallel \ldots \parallel C_n]$ $n \geq 2$

. C : $\mathbb{Com}$ *Sequential commands* (.2)

C ::= skip | X := E | X := ? | $(C_1; C_2)$ | $(B \rightarrow C_1 \lozenge C_2)$

| (B * C) | (B ¿ C) | √

Execution of a program "$[C_1 \parallel C_2 \parallel \ldots \parallel C_n]$" consists in executing processes C_1, C_2, … and C_n in parallel. These commands act upon implicitly declared shared global variables. Evaluation of a Boolean expression B, execution of assignments "X := E" or "X := ?" and execution of await commands "(B ¿ C)" are *atomic* or *indivisible actions* that is no concurrent action can modify the value of the variables involved in this action. When a process attempts to execute an await command "(B ¿ C)", it is delayed until the condition B is true. Then the command C is executed as an indivisible action. Evaluation of B is part of the indivisible action so that another process may not change variables so as to make B false after B has been evaluated but before C begins execution. Upon termination of "(B ¿ C)", parallel processing continues. If two or more processes are waiting for the same condition B, any one of them may be allowed to proceed when B becomes true, while the others continue waiting. The order in which processes are scheduled is indifferent, for example a *weak fairness hypothesis* would be that no nonterminated hence permanently enabled process C_k can be indefinitely delayed. A *strong fairness hypothesis* would be that no process can be indefinitely delayed if condition B can be infinitely often evaluated to true while that process is waiting, see LAMPORT [1980a], LEHMANN, PNUELI & STAVI [1981], MANNA & PNUELI [1984] [1989] and FRANCEZ [1986] for more details on fairness hypotheses. OWICKI [1975] and OWICKI & GRIES [1976a] assume that await commands (B ¿ C) cannot be imbricated since this could lead to deadlocks (as in "[(true ¿ (false ¿ skip)) ∥ skip]"). Execution of a program "$[C_1 \parallel C_2 \parallel \ldots \parallel C_n]$" is terminated when all processes have finished their execution. We use the empty command "√" to denote termination.

To define the operational semantics of parallel programs, we introduce control states:

DEFINITION *Labels designating control states* (201)

if Pp is $[C_1 \parallel C_2 \parallel \ldots \parallel C_n]$ then

- . $\mathbb{At}[\![(B \text{ ¿ } C)]\!] = \{(B \text{ ¿ } C)\}$, $\mathbb{In}[\![(B \text{ ¿ } C)]\!] = \emptyset$, $\mathbb{After}[\![(B \text{ ¿ } C)]\!] = \{\surd\}$ (.1)
- . $\mathbb{Lab}[\![Pp]\!] = \mathbb{Lab}[\![C_1]\!] \times \ldots \times \mathbb{Lab}[\![C_n]\!]$ (.2)
- . $\mathbb{PpLa} = \cup\{\mathbb{Lab}[\![Pp]\!] : Pp \in \mathbb{Papr}\}$ (.3)
- . $\mathbb{At}[\![Pp]\!] = \mathbb{At}[\![C_1]\!] \times \ldots \times \mathbb{At}[\![C_n]\!]$ (.4)
- . $\mathbb{After}[\![Pp]\!] = \mathbb{After}[\![C_1]\!] \times \ldots \times \mathbb{After}[\![C_n]\!]$ (.5)
- . $\mathbb{In}[\![Pp]\!] = \mathbb{Lab}[\![Pp]\!] - \mathbb{At}[\![Pp]\!] - \mathbb{After}[\![Pp]\!]$ (.6)

The *operational semantics* of parallel programs is defined by interleaved executions of atomic actions as defined by (13.1) to (13.6):

DEFINITION *Operational semantics of parallel programs (compositional presentation)* (202)

d : D	*Data*		(.1)
s : $S = \mathbb{Var} \to D$		*States*	(.2)
γ : $\Gamma = (S \times \mathbb{Com}) \cup (S \times \mathbb{PpLa})$		*Configurations*	(.3)
op : $(\mathbb{Com} \cup \mathbb{Papr}) \to \mathcal{P}(\Gamma \times \Gamma)$		*Operational transition relation*	(.4)

$op[\![(B \text{ ¿ } C)]\!] = \{\langle\langle s, (B \text{ ¿ } C)\rangle, s'\rangle : s \in \underline{B} \wedge \langle\langle s, C\rangle, s'\rangle \in op[\![C]\!]^*\}$ (.5)

$op[\![\surd]\!] = \emptyset$ (.6)

$op[\![[C_1 \parallel C_2 \parallel \ldots \parallel C_n]]\!] =$ (.7)

$\{\langle\langle s, L\rangle, \langle s', L[k \leftarrow L'_k]\rangle\rangle : k \in \{1, \ldots, n\} \wedge \langle\langle s, L_k\rangle, \langle s', L'_k\rangle\rangle \in op[\![C_k]\!]\}$

$\cup \quad \{\langle\langle s, L\rangle, \langle s', L[k \leftarrow \surd]\rangle\rangle : k \in \{1, \ldots, n\} \wedge \langle\langle s, L_k\rangle, s'\rangle \in op[\![C_k]\!]\}$

Example (203)

We illustrate the proof methods for parallel programs on the following program Pp which increments the value of variable X by a + b where a and b are given integer constants:

(204)

```
[ X := X + a || X := X + b ]
 |          | |           |
 L11      L12 L21        L22
```

Its operational semantics is (for simplicity we write $\langle s, \langle L_1, \ldots, L_n\rangle\rangle$ as $\langle s, L_1, \ldots, L_n\rangle$):

$op[\![Pp]\!] = \{<<s, L_{11}, L_{21}>, <s[X \leftarrow s(X) + a], L_{12}, L_{21}>> : s \in S\}$
$\cup \{<<s, L_{11}, L_{22}>, <s[X \leftarrow s(X) + a], L_{12}, L_{22}>> : s \in S\}$
$\cup \{<<s, L_{11}, L_{21}>, <s[X \leftarrow s(X) + b], L_{11}, L_{22}>> : s \in S\}$
$\cup \{<<s, L_{12}, L_{21}>, <s[X \leftarrow s(X) + b], L_{12}, L_{22}>> : s \in S\}$

where $L_{11} = (X := X + 1)^0$, $L_{12} = \surd^0$, $L_{21} = (X := X + 1)^1$ and $L_{22} = \surd^1$. This program is partially correct $\underline{\{p\}Pp\{q\}}$ for the specification $p = \{s \in S : s(X) = s(x)\}$ and $q = \{s \in S : s(X) = s(x) + a + b\}$. ■

By (44), the operational semantics (202) can also be given a stepwise presentation:

LEMMA *Operational semantics of parallel programs (stepwise presentation)* (205)

$op[\![[C_1 \parallel C_2 \parallel \ldots \parallel C_n]]\!]$ (.1)
$= \{<<s, L>, <s', L[k \leftarrow L'_k]>> : k \in \{1, \ldots, n\}$
$\wedge L_k \notin \mathbb{A}\mathrm{fter}[\![C_k]\!] \wedge s' \in \mathbb{N}\mathrm{extS}[\![C_k]\!]<s, L_k> \wedge L'_k \in \mathbb{N}\mathrm{extL}[\![C_k]\!]<s, L_k>\}$

where:

$\mathbb{S}\mathrm{tep}[\![C]\!][(\ldots(((B ¿ C'); C_1); C_2)\ldots; C_n)] = (B ¿ C')$ (.2)

$\mathbb{S}\mathrm{ucc}[\![C]\!][(\ldots(((B ¿ C'); C_1); C_2)\ldots; C_n)] = (\ldots(C_1; C_2)\ldots; C_n)$ (.3)

if $\mathbb{S}\mathrm{tep}[\![C]\!][L]$ is $(B ¿ C')$ then

$\mathbb{N}\mathrm{extS}[\![C]\!]<s, L> = \{s' : s \in \underline{B} \wedge <<s, C'>, s'> \in op[\![C]\!]^*\}$ (.4)

$\mathbb{N}\mathrm{extL}[\![C]\!]<s, L> = \{\mathbb{S}\mathrm{ucc}[\![C]\!][L] : s \in \underline{B}\}$ (.5)

Example *Stepwise operational semantics of parallel program (204)* (206)

The stepwise presentation of the operational semantics of program $Pp = [C_1 \parallel C_2]$ defined at (204) is specified by $C_1 = (X := X + a)$, $C_2 = (X := X + b)$, $\mathbb{A}\mathrm{t}[\![C_1]\!] = \{L_{11}\}$, $\mathbb{I}\mathrm{n}[\![C_1]\!] = \emptyset$, $\mathbb{A}\mathrm{fter}[\![C_1]\!] = \{L_{12}\}$, $\mathbb{L}\mathrm{ab}[\![C_1]\!] = \{L_{11}, L_{12}\}$, $\mathbb{A}\mathrm{t}[\![C_2]\!] = \{L_{21}\}$, $\mathbb{I}\mathrm{n}[\![C_2]\!] = \emptyset$, $\mathbb{A}\mathrm{fter}[\![C_2]\!] = \{L_{22}\}$, $\mathbb{L}\mathrm{ab}[\![C_2]\!] = \{L_{21}, L_{22}\}$, $\mathbb{N}\mathrm{extS}[\![C_1]\!]<s, L_{11}> = \{s[X \leftarrow s(X) + a]\}$, $\mathbb{N}\mathrm{extL}[\![C_1]\!]<s, L_{11}> = \{L_{12}\}$, $\mathbb{N}\mathrm{extS}[\![C_2]\!]<s, L_{21}> = \{s[X \leftarrow s(X) + b]\}$, $\mathbb{N}\mathrm{extL}[\![C_2]\!]<s, L_{21}> = \{L_{22}\}$. ■

8.9.2 À la Floyd proof methods for parallel programs with shared variables

We follow the style of presentation of COUSOT & COUSOT [1984]. From a semantical point of view, proofs of partial correctness $\underline{\{p\}Pp\{q\}}$ consist in discovering local invariants $I \in \mathbb{L}\mathrm{inv}[\![Pp]\!]$ attached to control points and in proving that they satisfy local verification conditions $lvc[\![Pp]\!][p, q](I)$ expressing that local invariants must remain true after execution of atomic actions. Hence without complementary hypotheses on the set

$\mathbb{L}\mathrm{inv}[\![\mathrm{Pp}]\!]$ of local invariants we can assume that à la Floyd proof methods are of the form:

$$[\exists\, I \in \mathbb{L}\mathrm{inv}[\![\mathrm{Pp}]\!].\ \mathrm{lvc}[\![\mathrm{Pp}]\!][p,\ q](I)] \tag{207}$$

Up to a connection (α, γ) between local invariants $I \in \mathbb{L}\mathrm{inv}[\![\mathrm{Pp}]\!]$ and global invariants $i \in \mathcal{P}(\Gamma)$:

$$\alpha \in \mathcal{P}(\Gamma) \rightarrow \mathbb{L}\mathrm{inv}[\![\mathrm{Pp}]\!]) \tag{208}$$
$$\gamma \in \mathbb{L}\mathrm{inv}[\![\mathrm{Pp}]\!] \rightarrow \mathcal{P}(\Gamma) \tag{209}$$

this proof method (207) consists in applying induction principle (29.1):

$$[\exists\, i \in \mathcal{P}(\Gamma).\ \mathrm{gvc}[\![\mathrm{Pp}]\!][p, q](i)] \tag{210}$$

This induction principle involves a single global invariant i on configurations (i.e. memory and control states), which is true for initial configurations satisfying p, remains true after each program step and implies q for final configurations:

$$\begin{aligned}
& & & (211)\\
\mathrm{gvc}[\![\mathrm{Pp}]\!][p, q](i) = \quad & (\forall\, s \in p.\ \langle s, C_1, \ldots C_n\rangle \in i) & & (.1)\\
& \wedge\ (i \rceil \mathit{op}[\![\mathrm{Pp}]\!] \subseteq \Gamma \times i) & & (.2)\\
& \wedge\ (\forall \langle s, \surd, \ldots, \surd\rangle \in i.\ s \in q) & & (.3)
\end{aligned}$$

By theorem (29.1) this induction principle (210) is semantically sound ($\Leftarrow$) and complete ($\Rightarrow$):

$$\underline{\{p\}\mathrm{Pp}\{q\}} \Leftrightarrow [\exists\, i \in \mathcal{P}(\Gamma).\ \mathrm{gvc}[\![\mathrm{Pp}]\!][p, q](i)] \tag{212}$$

The connection $i = \gamma(I)$ and $I = \alpha(i)$ between (207) and (210) expresses the fact that the global invariant i can be decomposed into local assertions I attached to control points. It induces a logical connection between local and global verification conditions:

$$\forall\, I \in \mathbb{L}\mathrm{inv}[\![\mathrm{Pp}]\!].\ \mathrm{lvc}[\![\mathrm{Pp}]\!][p, q](I) \Rightarrow \mathrm{gvc}[\![\mathrm{Pp}]\!][p, q](\gamma(I)) \tag{213}$$
$$\forall\, i \in \mathcal{P}(\Gamma).\ \mathrm{gvc}[\![\mathrm{Pp}]\!][p, q](i) \Rightarrow \mathrm{lvc}[\![\mathrm{Pp}]\!][p, q](\alpha(i)) \tag{214}$$

which, together with (212), ensures the soundness and semantical completeness of (207):

$$\underline{\{p\}\mathrm{Pp}\{q\}} \Leftrightarrow [\exists\, I \in \mathbb{L}\mathrm{inv}[\![\mathrm{Pp}]\!].\ \mathrm{lvc}[\![\mathrm{Pp}]\!][p,\ q](I)] \tag{215}$$

This approach can also be used to formally construct proof methods by symbolic calculus: Given gvc⟦Pp⟧[p, q], α and γ , we can let lvc⟦Pp⟧[p, q](I) be gvc⟦Pp⟧[p, q](γ(I)) (so that the proof method is sound by construction) and check semantical completeness by showing that gvc⟦Pp⟧[p, q](i) $\Rightarrow$ lvc⟦Pp⟧[p, q](α(i)) = gvc⟦Pp⟧[p, q](γ(α(i))) (which is often obvious because gvc⟦Pp⟧[p, q] is monotone and (α, γ) is a Galois connection so that $\forall$ i $\in \mathcal{P}(\Gamma)$. i $\Rightarrow \gamma(\alpha(i))$). For example, this point of view was applied to the design of a partial correctness proof method for CSP programs in COUSOT & COUSOT [1980].

We now explain a few partial correctness proof methods for parallel programs following these guidelines.

8.9.2.1 Using a single global invariant

ASHCROFT [1975] partial correctness proof method and the one of KELLER [1976] (illustrated by BABICH [1979]) consist in directly applying induction principle (29.1).

Example (216)

To prove partial correctness of (204) by (210), we can use the global invariant:

$$\begin{aligned} i = \; & \{<s, L_{11}, L_{21}> : s(X) = s(x)\} \cup \{<s, L_{12}, L_{21}> : s(X) = s(x) + a\} \\ & \cup \{<s, L_{11}, L_{22}> : s(X) = s(x) + b\} \cup \{<s, L_{12}, L_{22}>: s(X) = s(x) + a + b\} \end{aligned}$$

and show that (for all s, s' $\in S$):

$$\begin{aligned} & (s \in p) \Rightarrow (<s, L_{11}, L_{21}> \in i) \\ & (<s, L_{11}, L_{21}> \in i \wedge s' = s[X \leftarrow s(X) + a]) \Rightarrow (<s', L_{12}, L_{21}> \in i) \\ & (<s, L_{11}, L_{22}> \in i \wedge s' = s[X \leftarrow s(X) + a]) \Rightarrow (<s', L_{12}, L_{22}> \in i) \\ & (<s, L_{11}, L_{21}> \in i \wedge s' = s[X \leftarrow s(X) + b]) \Rightarrow (<s', L_{11}, L_{22}> \in i) \\ & (<s, L_{12}, L_{21}> \in i \wedge s' = s[X \leftarrow s(X) + b]) \Rightarrow (<s', L_{12}, L_{22}> \in i) \\ & (<s, L_{12}, L_{22}> \in i) \Rightarrow (s \in q) \end{aligned}$$

■

In Ashcroft-Keller method, $\mathbb{Linv}_{AK}$⟦Pp⟧ is chosen as $\mathcal{P}(\Gamma)$ and (α_{AK}, γ_{AK}) is identity. According to the operational semantics (205), the verification condition (i ⌉ *op*⟦Pp⟧ $\subseteq \Gamma \times$ i) of (210) can be decomposed into simpler verification conditions corresponding to each atomic action. Otherwise stated (i ⌉ *op*⟦Pp⟧ $\subseteq \Gamma \times$ i) is equivalent to:

$$\forall \text{ s, s'} \in S.\ \forall \text{ L} \in \mathbb{Lab}⟦\text{Pp}⟧.\ \forall \text{ k} \in \{1, \ldots, n\}. \tag{217}$$

$(\langle s, L\rangle \in i \wedge L_k \notin \mathrm{After}[C_k] \wedge s' \in \mathrm{NextS}[C_k]\langle s, L_k\rangle \wedge L'_k \in \mathrm{NextL}[C_k]\langle s, L_k\rangle)$
$\quad \Rightarrow (\langle s', L[k \leftarrow L'_k]\rangle \in i)$

This means that starting with i true of $\langle s, L_1, \ldots, L_k, \ldots, L_n\rangle$ and executing any atomic action labeled L_k of any process C_k of the program (that is a transition in that process C_k from configuration $\langle s, L_k\rangle$ to configuration $\langle s', L'_k\rangle$) leaves i true of $\langle s', L_1, \ldots, L'_k, \ldots, L_n\rangle$.

The difficulty with this method is that for large programs the single global invariant i tends to be unmanageable without being decomposed into simpler assertions.

8.9.2.2 Using an invariant on memory states for each control state

Early attempts towards the decomposition of the global invariant such as ASHCROFT & MANNA [1970] and LEVITT [1972] involves the transformation of the parallel program into an equivalent nondeterministic one upon which Floyd-Naur's partial correctness proof method is applied. As observed by KELLER [1976], this simply consists in using induction principle (210) with local invariants on memory states attached to each control state.

Example *Partial correctness proof of program (204) by Ashcroft & Manna's method* (218)

To prove partial correctness of parallel program (204) by (220), we can use the local invariants:

$I\langle L_{11}, L_{21}\rangle = \{s \in S : s(X) = s(x)\}$
$I\langle L_{11}, L_{22}\rangle = \{s \in S : s(X) = s(x) + b\}$
$I\langle L_{12}, L_{21}\rangle = \{s \in S : s(X) = s(x) + a\}$
$I\langle L_{12}, L_{22}\rangle = \{s \in S : s(X) = s(x) + a + b\}$

and show that (for all s, s' $\in S$):

Initialization (220.1):
$(s \in p) \Rightarrow (s \in I[L_{11} \| L_{21}])$
Induction (220.2):
$(s \in I\langle L_{11}, L_{21}\rangle \wedge s' = s[X \leftarrow s(X) + a]) \Rightarrow (s' \in I\langle L_{12}, L_{21}\rangle)$
$(s \in I\langle L_{11}, L_{22}\rangle \wedge s' = s[X \leftarrow s(X) + a]) \Rightarrow (s' \in I\langle L_{12}, L_{22}\rangle)$
$(s \in I\langle L_{11}, L_{21}\rangle \wedge s' = s[X \leftarrow s(X) + b]) \Rightarrow (s' \in I\langle L_{11}, L_{22}\rangle)$
$(s \in I\langle L_{12}, L_{21}\rangle \wedge s' = s[X \leftarrow s(X) + b]) \Rightarrow (s' \in I\langle L_{12}, L_{22}\rangle)$
Finalization (220.3):
$(s \in I\langle L_{12}, L_{22}\rangle) \Rightarrow (s \in q)$

■

To formally construct this proof method, we exactly follow the development of paragraph § 5.2 for Floyd-Naur method.Then we introduce local invariants and their connection with the global invariant of (210):

DEFINITION *Connection between local and global invariants* (219)

. $\mathbb{Linv}_{AM}[\![Pp]\!] = \mathbb{Lab}[\![Pp]\!] \rightarrow \mathbb{Ass}$ (where $\mathbb{Ass} = \mathcal{P}(S)$) (.1)

. $\alpha_{AM} : \mathcal{P}(\Gamma) \rightarrow \mathbb{Linv}[\![Pp]\!], \alpha(i)(L) = \{s : <s, L> \in i\}$ (.2)

. $\gamma_{AM} : \mathbb{Linv}[\![Pp]\!] \rightarrow \mathcal{P}(\Gamma), \gamma(I) = \{<s, L> : L \in \mathbb{Lab}[\![Pp]\!] \wedge s \in I(L)\}$ (.3)

Then we derive local verification conditions by $lvc[\![Pp]\!][p, q](I) = gvc[\![Pp]\!][p, q](\gamma_{AM}(I))$:

THEOREM *Ashcroft & Manna partial correctness proof method for parallel programs* (220)

Pp is $[C_1 \parallel C_2 \parallel \ldots \parallel C_n]$ then $\underline{\{p\}Pp\{q\}}$ holds if and only if their exists $I \in \mathbb{Lab}[\![Pp]\!] \rightarrow \mathbb{Ass}$
such that:

. if $L \in \mathbb{At}[\![Pp]\!]$ then $p \subseteq I(L)$ (.1)

. if $s, s' \in S$, $L \in \mathbb{At}[\![Pp]\!] \cup \mathbb{In}[\![Pp]\!]$, $k \in \{1, \ldots, n\}$ then (.2)

$(s \in I(L) \wedge L_k \notin \mathbb{After}[\![C_k]\!] \wedge s' \in \mathbb{NextS}[\![C_k]\!]<s, L_k> \wedge L'_k \in \mathbb{NextL}[\![C_k]\!]<s, L_k>)$
$\Rightarrow (s' \in I(L[k \leftarrow L'_k])$

. if $L \in \mathbb{After}[\![Pp]\!]$ then $I(L) \subseteq q$ (.3)

Otherwise stated, starting with $I<L_1, \ldots, L_k, \ldots, L_n>$ true of s and executing any atomic action labeled L_k of any process C_k of the program (that is a transition in that process C_k from configuration $<s, L_k>$ to configuration $<s', L'_k>$) leaves $I<L_1, \ldots, L'_k, \ldots, L_n>$ true of s'. These local verification conditions can further be detailed as in (45) for each possible kind of atomic action. In particular when the atomic action $\mathbb{Step}[\![C_k]\!][L_k]$ labeled L_k is an await command (B ¿ C) with next label $L'_k = \mathbb{Succ}[\![C_k]\!][L_k]$ we must prove $\underline{\{}\ I(L) \cap \underline{B}\ \underline{\}C\{}\ I(L[k \leftarrow L'_k]\ \underline{\}}$ to which Floyd's stepwise partial correctness proof method (45) is directly applicable.

As shown by the success of Floyd-Naur partial correctness proof method, this approach is very well suited for sequential programs. However for parallel programs $Pp = [C_1 \parallel C_2 \parallel \ldots \parallel C_n]$ the number of local invariants I(L), $L \in \mathbb{Lab}[\![Pp]\!]$ grows as the number $|\ \mathbb{Lab}[\![Pp]\!]\ |$ of control states that is as the product $|\mathbb{Lab}[\![C_1]\!]| . |\mathbb{Lab}[\![C_2]\!]| . \ldots . |\mathbb{Lab}[\![C_n]\!]|$ of the sizes of the processes $C_1, C_2, \ldots, C_n$. Apart for trivial programs this exponential explosion is rapidly unmanageable.

8.9.2.3 Using an invariant on memory states for each program point

In paragraph § 8.9.2.2, Floyd-Naur partial correctness proof method was generalized to parallel programs by using an invariant on memory states for each control state $<L_1, \dots, L_n> \in \mathbb{Lab}[\![Pp]\!]$. For sequential programs this is equivalent to the use of an invariant $I(L) \in \mathbb{Ass}$ on memory states for each program point $L \in \mathbb{Prpt}[\![Pp]\!]$. These two points of view do not coincide for parallel programs. So, as first suggested by ASHCROFT [1975], Floyd-Naur proof method can also be generalized to parallel programs using an invariant on memory states for each program point. This consists in defining:

DEFINITION *Connection between local and global invariants* (221)

. $\mathbb{Prpt}[\![[C_1 \| \dots \| C_n]]\!] = \cup\{\mathbb{Lab}[\![C_k]\!] : k \in \{1,\dots, n\}\}$ (.1)

. $\mathbb{Linv}_A[\![Pp]\!] = \mathbb{Prpt}[\![Pp]\!] \to \mathbb{Ass}$ (.2)

. $\alpha_A : \mathcal{P}(\Gamma) \to \mathbb{Linv}_A[\![Pp]\!]$, $\alpha_A(i)(l) = \{s : \exists\, L \in \mathbb{Lab}[\![Pp]\!].\ <s, L[k \leftarrow l]> \in i\}$ (.3)

. $\gamma_A : \mathbb{Linv}_A[\![Pp]\!] \to \mathcal{P}(\Gamma)$, $\gamma_A(I) = \{<s, L> : \forall\, j \in \{1, \dots, n\}.\ s \in I(L_j)\}$ (.4)

The induction step $(\gamma_A(I) \rceil\, op[\![Pp]\!] \subseteq \Gamma \times \gamma_A(I))$ of the corresponding verification conditions $lvc[\![Pp]\!][p, q](I) = gvc[\![Pp]\!][p, q](\gamma_A(I))$ can be, following OWICKI & GRIES [1976a], decomposed into a sequential proof and a proof of interference freedom (also called "monotony condition" in LAMPORT [1977]). Sequential correctness asserts that executing any atomic action labeled L of any process C_k of the program (that is a transition of that process C_k from configuration <s, L> to configuration <s', L'>) starting with I(L) true of s makes I(L') true of s'. Interference freedom asserts that for every label L" in a different process C_m, starting with both I(L") and I(L) true of s leaves I(L") true of s':

THEOREM *Incomplete partial correctness proof method for parallel programs* (222)

if Pp is $[C_1 \| C_2 \| \dots \| C_n]$ then <u>{p}Pp{q}</u> holds if their exists $I \in \mathbb{Prpt}[\![Pp]\!] \to$
$\mathbb{Ass}$
such that :

. if $k \in \{1, \dots, n\}$ and $L \in \mathbb{At}[\![C_k]\!]$ then $p \subseteq I(L)$ (.1)

. if $s, s' \in S$, $k \in \{1, \dots, n\}$, $L \in \mathbb{At}[\![C_k]\!] \cup \mathbb{In}[\![C_k]\!]$ and $L' \in \mathbb{Lab}[\![C_k]\!]$ then (.2)
$$[s \in I(L) \wedge s' \in \mathbb{NextS}[\![C_k]\!]<s, L> \wedge L' \in \mathbb{NextL}[\![C_k]\!]<s, L>] \Rightarrow s' \in I(L')$$

. if $s, s' \in S$, $k \in \{1, \dots, n\}$, $L \in \mathbb{At}[\![C_k]\!] \cup \mathbb{In}[\![C_k]\!]$, $m \in \{1, \dots, n\} - \{k\}$ (.3)
and $L'' \in \mathbb{Lab}[\![C_m]\!]$ then
$$[s \in I(L'') \wedge s \in I(L) \wedge s' \in \mathbb{NextS}[\![C_k]\!]<s, L>] \Rightarrow s' \in I(L'')$$

. $\cap\ \{I(L) : \exists\, k \in \{1, \dots, n\}.\ L \in \mathbb{After}[\![C_k]\!]\ \} \subseteq q$ (.4)

but the reciprocal is not true.

Example *Partial correctness proof of program (204) with method (222)* (223)

Assuming a = 1 and b = 2, we can apply (222) to prove partial correctness of parallel program (204). The verification conditions are the following:

Initialization (222.1):

$p \subseteq I(L_{11})$

$p \subseteq I(L_{21})$

Sequential correctness (222.2):

$s \in I(L_{11}) \Rightarrow s[X \leftarrow s(X) + 1] \in I(L_{12})$

$s \in I(L_{21}) \Rightarrow s[X \leftarrow s(X) + 2] \in I(L_{22})$

Interference freedom (222.3):

$s \in I(L_{11}) \wedge s \in I(L_{21}) \Rightarrow s[X \leftarrow s(X) + 2] \in I(L_{11})$

$s \in I(L_{12}) \wedge s \in I(L_{21}) \Rightarrow s[X \leftarrow s(X) + 2] \in I(L_{12})$

$s \in I(L_{21}) \wedge s \in I(L_{11}) \Rightarrow s[X \leftarrow s(X) + 1] \in I(L_{21})$

$s \in I(L_{22}) \wedge s \in I(L_{11}) \Rightarrow s[X \leftarrow s(X) + 1] \in I(L_{22})$

Finalization (222.4):

$I(L_{12}) \cap I(L_{22}) \subseteq q$

They are satisfied by the following local invariants:

$I[L_{11}] = \{s \in S : s(X) = s(x) \vee s(X) = s(x) + 2\}$

$I[L_{12}] = \{s \in S : s(X) = s(x) + 1 \vee s(X) = s(x) + 3\}$

$I[L_{21}] = \{s \in S : s(X) = s(x) \vee s(X) = s(x) + 1\}$

$I[L_{22}] = \{s \in S : s(X) = s(x) + 2 \vee s(X) = s(x) + 3\}$

■

Using example (204) with a = b = 1, KELLER [1976] and OWICKI & GRIES [1976a] have shown that the corresponding proof method is semantically incomplete:

Counterexample *Incompleteness of method (222) for program (204)* (224)

Formally, the strongest global invariant $\underline{i}$ satisfying the global verification condition $gvc[\![Pp]\!][p, q]$ in (210) for program (204) is given by its fixpoint characterization (30) as:

$$\begin{aligned} \underline{i} \quad &= \quad \mathit{lfp}\ \lambda X : \mathcal{P}(\Gamma).\ \{<s, C_1, \ldots, C_n> : s \in p\} \cup \{\gamma : \exists \gamma'.\ <\gamma', \gamma> \in X \rceil op[\![Pp]\!]\} \\ = \quad & \{<s, L_{11}, L_{21}> : s(X) = s(x)\} \cup \{<s, L_{12}, L_{21}> : s(X) = s(x) + a\} \\ & \cup \{<s, L_{11}, L_{22}> : s(X) = s(x) + b\} \cup \{<s, L_{12}, L_{22}>: s(X) = s(x) + a + b\} \end{aligned}$$

It follows by monotony, that the strongest local invariants are $\underline{I} = \alpha_A(\underline{i})$, that is to say:

$\underline{I}(L_{11}) = \{s \in S : s(X) = s(x) \vee s(X) = s(x) + b\}$

$\underline{I}(L_{12}) = \{s \in S : s(X) = s(x) + a \vee s(X) = s(x) + a + b\}$

$\underline{I}(L_{21}) = \{s \in S : s(X) = s(x) \vee s(X) = s(x) + a\}$

$\underline{I}(L_{22}) = \{s \in S : s(X) = s(x) + b \vee s(X) = s(x) + a + b\}$

When a = b = 1 they are too weak to satisfy the interference freedom (222.3) and finalization (222.4) verification conditions as given at example (223) ■

8.9.2.4 Using an invariant on memory and control states for each program point

The use of an invariant on memory states for each program point is incomplete because the relationship between memory and control states is lost. This can be avoided by using local invariants $I(L_k)$ attached to program points L_k of each process C_k of the program Pp specifying the relationship between the memory state and the control state of other processes:

$$\begin{aligned}\mathbb{Linv}_L[\![Pp]\!] = \{I \in \mathbb{Prpt}[\![Pp]\!] \to \mathcal{P}(S \times \mathbb{Prpt}[\![Pp]\!]^{n-1}) :\\ \forall k \in \{1, \ldots, n\}.\ \forall L_k \in \mathbb{Lab}[\![C_k]\!].\\ I(L_k) \subseteq S \times \mathbb{Lab}[\![C_1]\!] \times \ldots \times \mathbb{Lab}[\![C_{k-1}]\!] \times \mathbb{Lab}[\![C_{k+1}]\!] \times \ldots \times \mathbb{Lab}[\![C_n]\!]\}\end{aligned} \tag{225}$$

This way of expressing the global invariant can be extirpated from NEWTON [1975] and is clear in LAMPORT [1977]. These local invariants can be written as a proof outline (as in 7.2.4 and OWICKI & GRIES [1976a]) in which a predicate P_k representing $I(L_k)$ is attached to control point L_k. Control predicates à la LAMPORT [1977] [1980b] and LAMPORT & SCHNEIDER [1984] can be used in P_k to explicitly mention the control state. Such control predicates can, to some extent, be defined in a language independent way (COUSOT & COUSOT [1989]).

Example *Proof outline for parallel program (204)*

To prove partial correctness of parallel program (204), we can use the following local invariants:

$$\begin{aligned}I[L_{11}] &= \{\langle s, L_{21}\rangle : s(X) = s(x)\} \cup \{\langle s, L_{22}\rangle : s(X) = s(x) + b\}\\ I[L_{12}] &= \{\langle s, L_{21}\rangle : s(X) = s(x) + a\} \cup \{\langle s, L_{22}\rangle : s(X) = s(x) + a + b\}\\ I[L_{21}] &= \{\langle s, L_{11}\rangle : s(X) = s(x)\} \cup \{\langle s, L_{12}\rangle : s(X) = s(x) + a\}\end{aligned} \tag{226}$$

which can be specified by the following proof outline:

$\{ X = x \}$ (227)

[L_{11} : $\{ (at(L_{21}) \wedge X = x) \vee (at(L_{22}) \wedge X = x + b) \}$

X := X + a

L_{12} : $\{ (at(L_{21}) \wedge X = x + a) \vee (at(L_{22}) \wedge X = x + a + b) \}$

|| L_{21} : $\{ (at(L_{11}) \wedge X = x) \vee (at(L_{12}) \wedge X = x + a) \}$

$\mathbf{X := X + b}$

L_{22} : { (at(L_{11}) $\wedge$ X = x + b) $\vee$ (at(L_{12}) $\wedge$ X = x + a + b) }]

{ X = x + a + b }

■

This decomposition of the global invariant can be specified by the following connection between local and global invariants (recall that $<L_1, \ldots, L_n>_{\sim k}$ is $<L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n>$):

DEFINITION *Connection between local and global invariants* (228)

. $\alpha_L : \mathcal{P}(\Gamma) \rightarrow \mathbb{Linv}_L[\![Pp]\!]$, $\alpha_L(i)(L_k) = \{<s, L_{\sim k}> : <s, L> \in i\}$ (.1)

. $\gamma_L : \mathbb{Linv}_L[\![Pp]\!] \rightarrow \mathcal{P}(\Gamma)$, $\gamma_L(I) = \{<s, L> : \exists\, k \in \{1, \ldots, n\}.\ <s, L_{\sim k}> \in I(L_k)\}$ (.2)

Observe that (α_L, γ_L) is a bijection between $\mathcal{P}(\Gamma)$ and $\mathbb{Linv}_L[\![Pp]\!]$ which ensures the soundness and semantical completeness of the derived proof method.

8.9.2.4.1 The strengthened Lamport and Owicki & Gries method

The local verification conditions $lvc[\![Pp]\!][p, q](I) = gvc[\![Pp]\!][p, q](\gamma_L(I))$ corresponding to the above definition of $\mathbb{Linv}_L[\![Pp]\!]$ and (α_L, γ_L) were first designed by COUSOT & COUSOT [1984] and later by LAMPORT [1988]. The proof method is similar to that of LAMPORT [1977] and OWICKI & GRIES [1976a] but for the fact that it is strengthened by allowing the use of the proof outline of the processes not involved in the sequential or interference freedom proof.

Example (229)

To prove partial correctness of parallel program (204), we can use local invariants (226) and check the following local verification conditions (for all s, s' $\in S$, $c_1 \in \{L_{11}, L_{12}\}$ and $c_2 \in \{L_{21}, L_{22}\}$):

- Initialization (230.1):

  The initialization step ($\forall\, s \in p.\ <s, C_1, \ldots C_n> \in \gamma_A(I)$) states that the input specification p implies the invariants attached to entry points of the processes $C_1, \ldots, C_n$ of Pp:

  . $s \in p \Rightarrow <s, L_{21}> \in I(L_{11})$

  . $s \in p \Rightarrow <s, L_{11}> \in I(L_{21})$

- Induction step:

  The induction step ($\gamma_A(I) \rceil\ op[\![Pp]\!] \subseteq \Gamma \times \gamma_A(I)$) must be checked for all atomic actions of all processes C_k of program Pp that is all transitions of that process C_k from configuration <s, L> to configuration <s', L'>. The proof can be decomposed into sequential and interference freedom proofs:

- Sequential correctness (230.2):

To prove sequential correctness we must show that starting with I(L) true of $<s, L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n>$ (as well as the assertions $I(L_j)$ attached to control points L_j of the other processes C_j, $j \in \{1, \ldots, n\} - \{k\}$ true of $<s, L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_{k-1}, L, L_{k+1}, \ldots, L_n>$) makes I(L') true of $<s', L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n>$:

  . $[<s, c_2> \in I(L_{11}) \wedge <s, L_{11}> \in I(L_{21}) \cup I(L_{22})] \Rightarrow <s[X \leftarrow s(X) + a], c_2> \in I(L_{12})$

  . $[<s, c_1> \in I(L_{21}) \wedge <s, L_{21}> \in I(L_{11}) \cup I(L_{12})] \Rightarrow <s[X \leftarrow s(X) + b], c_1> \in I(L_{22})$

- Interference freedom (230.3):

Interference freedom asserts that for every label L" in a different process C_m, $m \in \{1, \ldots, n\} - \{k\}$ starting with both I(L) true of $<s, L_1, \ldots, L_{m-1}, L'', L_{m+1}, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n>$ and I(L") true of $<s, L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_{k-1}, L, L_{k+1}, \ldots, L_n>$ (as well as the assertions $I(L_j)$ attached to control points L_j of the other processes C_j, $j \in \{1, \ldots, n\} - \{k, m\}$ true of $<s, L_1, \ldots, L_{m-1}, L'', L_{m+1}, \ldots, L_{j-1}, L_{j+1}, \ldots, L_{k-1}, L, L_{k+1}, \ldots, L_n>$) leaves I(L") true of $<s', L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_{k-1}, L', L_{k+1}, \ldots, L_n>$:

  . $[<s, c_2> \in I(L_{11}) \wedge <s, L_{11}> \in I(L_{21})] \Rightarrow <s[X \leftarrow s(X) + b], c_2> \in I(L_{11})$

  . $[<s, c_2> \in I(L_{12}) \wedge <s, L_{12}> \in I(L_{21})] \Rightarrow <s[X \leftarrow s(X) + b], c_2> \in I(L_{12})$

  . $[<s, c_1> \in I(L_{21}) \wedge <s, L_{21}> \in I(L_{11})] \Rightarrow <s[X \leftarrow s(X) + a], c_1> \in I(L_{21})$

  . $[<s, c_1> \in I(L_{22}) \wedge <s, L_{22}> \in I(L_{11})] \Rightarrow <s[X \leftarrow s(X) + a], c_1> \in I(L_{22})$

• Finalization (230.4):

The final step of the partial correctness proof $(\forall <s, [\surd \parallel \ldots \parallel \surd]> \in \gamma_A(I).\ s \in q)$ shows that final states satisfy the output specification q:

  . $[<s, L_{22}> \in I(L_{12}) \wedge <s, L_{12}> \in I(L_{22})] \Rightarrow s \in q$

■

This proof method $[\exists I \in \mathbb{L}\mathrm{inv}_L[\![Pp]\!].\ \mathrm{lvc}_{SLO}[\![Pp]\!][p, q](I)]$ directly follows from the choice $\mathrm{lvc}_{SLO}[\![Pp]\!][p, q](I) = \mathrm{gvc}[\![Pp]\!][p, q](\gamma_L(I))$ with $\mathrm{gvc}[\![Pp]\!][p, q](i)$ defined by (211) and operational semantics (205):

THEOREM COUSOT & COUSOT [1984] *Strengthened Lamport and Owicki & Gries method* (230)

$\underline{\{p\}Pp\{q\}}$ where Pp is $[C_1 \parallel C_2 \parallel \ldots \parallel C_n]$ is true if and only if their exists $I \in \mathbb{Linv}_L[\![Pp]\!]$ such that for all $k \in \{1, \ldots, n\}$, $m \in \{1, \ldots, n\} - \{k\}$ and $s, s' \in S$:

. if $l \in \mathbb{At}[\![C_k]\!]$ and $L \in \mathbb{At}[\![Pp]\!]$ then $s \in p \Rightarrow \langle s, L_{\sim k}\rangle \in I(l)$ (.1)

. if $l \in \mathbb{At}[\![C_k]\!] \cup \mathbb{In}[\![C_k]\!]$ and $L \in \mathbb{Lab}[\![Pp]\!]$ then (.2)

$$[\langle s, L_{\sim k}\rangle \in I(l) \wedge \forall j \in \{1,\ldots,n\} - \{k\}. \langle s, L[k \leftarrow l]_{\sim j}\rangle \in I(L_j) \wedge s' \in \mathbb{NextS}[\![C_k]\!]\langle s, l\rangle \wedge l' \in \mathbb{NextL}[\![C_k]\!]\langle s, l\rangle] \Rightarrow \langle s', L_{\sim k}\rangle \in I(l')$$

. if $l \in \mathbb{At}[\![C_k]\!] \cup \mathbb{In}[\![C_k]\!]$, $l'' \in \mathbb{Lab}[\![C_m]\!]$ and $L \in \mathbb{Lab}[\![Pp]\!]$ then (.3)

$$[\langle s, L[k \leftarrow l]_{\sim m}\rangle \in I(l'') \wedge \langle s, L[m \leftarrow l'']_{\sim k}\rangle \in I(l) \wedge \forall j \in \{1, \ldots, n\} - \{k, m\}. \langle s, L[m \leftarrow l''][k \leftarrow l]_{\sim m}\rangle \in I(L_j) \wedge s' \in \mathbb{NextS}[\![C_k]\!]\langle s, l\rangle \wedge l' \in \mathbb{NextL}[\![C_k]\!]\langle s, l\rangle] \Rightarrow \langle s', L[k \leftarrow l']_{\sim m}\rangle \in I(l'')$$

. if $L \in \mathbb{After}[\![Pp]\!]$ then $[\forall k \in \{1, \ldots, n\}. \langle s, L_{\sim k}\rangle \in I(L_k)] \Rightarrow s \in q$ (.4)

8.9.2.4.2 Newton's method

NEWTON [1975] proof method (although designed for a quite different definition of concurrent programs) is a weakened version of LAMPORT [1977] and OWICKI & GRIES [1976a] method in the sense that interference freeness simply consists in proving that any local invariant $I(L'')$, $L'' \in \mathbb{Lab}[\![C_m]\!]$ remains true after execution of any atomic action labeled L of any other process C_k, $k \neq m$ (and this without assuming that I(L) holds before executing this atomic action).

Example *Partial correctness proof of program (204) by Newton's method* (231)

To prove partial correctness of parallel program (204), we can use local invariants (226) and check the following local verification conditions (for all $s, s' \in S$, $c_1 \in \{L_{11}, L_{12}\}$ and $c_2 \in \{L_{21}, L_{22}\}$):

- Initialization (232.1):
  - . $s \in p \Rightarrow \langle s, L_{21}\rangle \in I(L_{11})$
  - . $s \in p \Rightarrow \langle s, L_{11}\rangle \in I(L_{21})$
- Sequential correctness (232.2):
  - . $\langle s, c_2\rangle \in I(L_{11}) \Rightarrow \langle s[X \leftarrow s(X) + a], c_2\rangle \in I(L_{12})$
  - . $\langle s, c_1\rangle \in I(L_{21}) \Rightarrow \langle s[X \leftarrow s(X) + b], c_1\rangle \in I(L_{22})$
- Interference freedom (232.3):
  - . $\langle s, c_2\rangle \in I(L_{11}) \Rightarrow \langle s[X \leftarrow s(X) + b], c_2\rangle \in I(L_{11})$
  - . $\langle s, c_2\rangle \in I(L_{12}) \Rightarrow \langle s[X \leftarrow s(X) + b], c_2\rangle \in I(L_{12})$
  - . $\langle s, c_1\rangle \in I(L_{21}) \Rightarrow \langle s[X \leftarrow s(X) + a], c_1\rangle \in I(L_{21})$
  - . $\langle s, c_1\rangle \in I(L_{22}) \Rightarrow \langle s[X \leftarrow s(X) + a], c_1\rangle \in I(L_{22})$
- Finalization (232.4):
  - . $[\langle s, L_{22}\rangle \in I(L_{12}) \wedge \langle s, L_{12}\rangle \in I(L_{22})] \Rightarrow s \in q$

■

This proof method $[\exists\, I \in \mathbb{L}\mathrm{inv}_L[\![Pp]\!].\ \mathrm{lvc}_N[\![Pp]\!][p, q](I)]$ is sound (since $\mathrm{lvc}_N[\![Pp]\!][p, q](I) \Rightarrow \mathrm{lvc}_{SLO}[\![Pp]\!][p, q](I) \Rightarrow \mathrm{gvc}[\![Pp]\!][p, q](\gamma(I))) \Rightarrow \underline{\{p\}Pp\{q\}}$) and relatively complete (since any $I \in \mathbb{L}\mathrm{inv}_L[\![Pp]\!]$ satisfying $\mathrm{lvc}_{SLO}[\![Pp]\!][p, q](I)$ can be strengthened into I' such that $I'(l) = \{<s, L_{\sim k}> \in I(l) : \forall j \in \{1,\dots,n\} - \{k\}.\ <s, L[k \leftarrow l]_{\sim j} \in I(L_j)\}$ satisfying $\mathrm{lvc}_N[\![Pp]\!][p, q](I')$):

THEOREM *Newton's method* (232)

$\underline{\{p\}Pp\{q\}}$ where Pp is $[C_1 \parallel C_2 \parallel \dots \parallel C_n]$ is true if and only if their exists $I \in \mathbb{L}\mathrm{inv}_L[\![Pp]\!]$ such that for all $k \in \{1, \dots, n\}$, $m \in \{1, \dots, n\} - \{k\}$ and $s, s' \in S$:

. if $l \in \mathbb{A}\mathrm{t}[\![C_k]\!]$ and $L \in \mathbb{A}\mathrm{t}[\![Pp]\!]$ then $s \in p \Rightarrow <s, L_{\sim k}> \in I(l)$ (.1)

. if $l \in \mathbb{A}\mathrm{t}[\![C_k]\!] \cup \mathbb{I}\mathrm{n}[\![C_k]\!]$ and $L \in \mathbb{L}\mathrm{ab}[\![Pp]\!]$ then (.2)

$[<s, L_{\sim k}> \in I(l) \wedge s' \in \mathbb{N}\mathrm{extS}[\![C_k]\!]<s, l> \wedge l' \in \mathbb{N}\mathrm{extL}[\![C_k]\!]<s, l>]$

$\Rightarrow <s', L_{\sim k}> \in I(l')$

. if $l \in \mathbb{A}\mathrm{t}[\![C_k]\!] \cup \mathbb{I}\mathrm{n}[\![C_k]\!]$, $l'' \in \mathbb{L}\mathrm{ab}[\![C_m]\!]$ and $L \in \mathbb{L}\mathrm{ab}[\![Pp]\!]$ then (.3)

$[<s, L[k \leftarrow l]_{\sim m}> \in I(l'') \wedge s' \in \mathbb{N}\mathrm{extS}[\![C_k]\!]<s, l> \wedge l' \in \mathbb{N}\mathrm{extL}[\![C_k]\!]<s, l>]$

$\Rightarrow <s', L[k \leftarrow l']_{\sim m}> \in I(L'')$

. if $L \in \mathbb{A}\mathrm{fter}[\![Pp]\!]$ then $[\forall k \in \{1, \dots, n\}.\ <s, L_{\sim k}> \in I(L_k)] \Rightarrow s \in q$ (.4)

Observe that interference freedom (232.3) disappears when considering monoprocess programs (n = 1, in which case (232) exactly amounts to Floyd's stepwise partial correctness proof method (45)) or multiprocess programs with assertions about parts of the store such that only operations acting upon separate parts may be performed concurrently (as in HOARE [1975] or MAZURKIEWICZ [1977] for example).

Although partial correctness proof methods (230) and (232) are both semantically complete, it may be the case that some assertions $I \in \mathbb{L}\mathrm{inv}_L[\![Pp]\!]$ satisfy (230) but cannot be proved to be invariant using (232) without being strengthened.

Example *Weak invariants for program (204)* (233)

Parallel program (204) with a = 1 and b = 2 is partially correct with respect to specification <p, q> such that $p = \{s \in S : s(X) = 0\}$ and $q = \{s \in S : s(X) = 3\}$. This can be proved using the verification conditions of (230) given at example (229) and the following invariants:

$I[L_{11}] = \{<s, L_2> : \mathit{even}(s(X))\}$

$I[L_{12}] = \{<s, L_2> : s(X) = 1 \vee s(X) = 3\}$

$I[L_{21}] = \{<s, L_1> : s(X) = 0 \vee s(X) = 1\}$

$I[L_{22}] = \{<s, L_1> : s(X) = 2 \vee s(X) = 3\}$

These invariants are too weak to satisfy the verification conditions of (232) given at example (231). ■

8.9.2.4.3 The lattice of proof methods including Lamport's method

Any proof method $[\exists\, I \in \mathbb{Linv}_L[\![Pp]\!].\ lvc[\![Pp]\!][p, q](I)]$ with local verification conditions $lvc[\![Pp]\!][p, q]$ such that $lvc_N[\![Pp]\!][p, q](I) \Rightarrow lvc[\![Pp]\!][p, q](I) \Rightarrow lvc_{SLO}[\![Pp]\!][p, q](I)$ for all $I \in \mathbb{Linv}_L[\![Pp]\!]$ is sound (since $lvc[\![Pp]\!][p, q](I) \Rightarrow lvc_{SLO}[\![Pp]\!][p, q](I) \Rightarrow gvc[\![Pp]\!][p, q](\gamma(I)) \Rightarrow \underline{\{p\}Pp\{q\}}$) and semantically complete (since $\underline{\{p\}Pp\{q\}} \Rightarrow [\exists\, i \in \mathcal{P}(\Gamma).\ gvc[\![Pp]\!][p, q](i)] \Rightarrow [\exists\, i \in \mathcal{P}(\Gamma).\ lvc_N[\![Pp]\!][p, q](\alpha(i))] \Rightarrow [\exists\, I \in \mathbb{Linv}_L[\![Pp]\!].\ lvc_N[\![Pp]\!][p, q](I)] \Rightarrow [\exists\, I \in \mathbb{Linv}_L[\![Pp]\!].\ lvc_N[\![Pp]\!][p, q](I)]$). This is the case of LAMPORT [1977]'s method which is OWICKI & GRIES [1976a] method with control predicates instead of auxiliary variables:

THEOREM LAMPORT [1977] *Lamport's partial correctness proof method* (234)

$\underline{\{p\}Pp\{q\}}$ where Pp is $[C_1 \parallel C_2 \parallel \ldots \parallel C_n]$ is true if and only if their exists $I \in \mathbb{Linv}_L[\![Pp]\!]$ such that for all $k \in \{1, \ldots, n\}$, $m \in \{1, \ldots, n\} - \{k\}$ and $s, s' \in S$:

. if $l \in \mathbb{At}[\![C_k]\!]$ and $L \in \mathbb{At}[\![Pp]\!]$ then $s \in p \Rightarrow \langle s, L_{\sim k}\rangle \in I(l)$ (.1)

. if $l \in \mathbb{At}[\![C_k]\!] \cup \mathbb{In}[\![C_k]\!]$ and $L \in \mathbb{Lab}[\![Pp]\!]$ then (.2)

$[\langle s, L_{\sim k}\rangle \in I(l) \wedge s' \in \mathbb{NextS}[\![C_k]\!]\langle s, l\rangle \wedge l' \in \mathbb{NextL}[\![C_k]\!]\langle s, l\rangle]$
$\Rightarrow \langle s', L_{\sim k}\rangle \in I(l')$

. if $l \in \mathbb{At}[\![C_k]\!] \cup \mathbb{In}[\![C_k]\!]$, $l'' \in \mathbb{Lab}[\![C_m]\!]$ and $L \in \mathbb{Lab}[\![Pp]\!]$ then (.3)

$[\langle s, L[k \leftarrow l]_{\sim m}\rangle \in I(l'') \wedge \langle s, L[m \leftarrow l'']_{\sim k}\rangle \in I(l) \wedge s' \in \mathbb{NextS}[\![C_k]\!]\langle s, l\rangle \wedge$
$l' \in \mathbb{NextL}[\![C_k]\!]\langle s, l\rangle] \Rightarrow \langle s', L[k \leftarrow l']_{\sim m}\rangle \in I(L'')$

. if $L \in \mathbb{After}[\![Pp]\!]$ then $[\forall\, k \in \{1, \ldots, n\}.\ \langle s, L_{\sim k}\rangle \in I(L_k)] \Rightarrow s \in q$ (.4)

Example *Partial correctness proof of program (204) by Lamport's method* (235)

To prove partial correctness of parallel program (204), we can use local invariants (226) and check the following local verification conditions (for all $s, s' \in S$, $c_1 \in \{L_{11}, L_{12}\}$ and $c_2 \in \{L_{21}, L_{22}\}$):

- Initialization (234.1):
  - . $s \in p \Rightarrow \langle s, L_{21}\rangle \in I(L_{11})$
  - . $s \in p \Rightarrow \langle s, L_{11}\rangle \in I(L_{21})$
- Sequential correctness (234.2):
  - . $\langle s, c_2\rangle \in I(L_{11}) \Rightarrow \langle s[X \leftarrow s(X) + a], c_2\rangle \in I(L_{12})$
  - . $\langle s, c_1\rangle \in I(L_{21}) \Rightarrow \langle s[X \leftarrow s(X) + b], c_1\rangle \in I(L_{22})$
- Interference freedom (234.3):
  - . $[\langle s, c_2\rangle \in I(L_{11}) \wedge \langle s, L_{11}\rangle \in I(L_{21})] \Rightarrow \langle s[X \leftarrow s(X) + b], c_2\rangle \in I(L_{11})$
  - . $[\langle s, c_2\rangle \in I(L_{12}) \wedge \langle s, L_{12}\rangle \in I(L_{21})] \Rightarrow \langle s[X \leftarrow s(X) + b], c_2\rangle \in I(L_{12})$