A HOARE-STYLE AXIOMATIZATION OF BURSTALL'S INTERMITTENT ASSERTIONS METHOD FOR NON-DETERMINISTIC PROGRAMS

Patrick COUSOT

LRIM-83-04

Septembre 1983

UNIVERSITÉ DE METZ

LABORATOIRE DE RECHERCHE EN INFORMATIQUE
DE METZ

Faculté des Sciences
Ile du Saulcy
57045 METZ CEDEX
FRANCE

A HOARE-STYLE AXIOMATIZATION OF BURSTALL'S INTERMITTENT ASSERTIONS METHOD FOR NON-DETERMINISTIC PROGRAMS

Patrick COUSOT

ABSTRACT

We introduce a Hoare-style logic describing, in a unified manner, Floyd's invariant assertions and Burstall's intermittent assertions total correctness proof methods for sequential programs with random assignments. It is shown, using classical examples, that formal proofs can be informally described as proof outlines, i.e. executable programs with appropriate comments.

RESUME

Nous introduisons une logique de Hoare permettant de décrire d'une manière uniforme la méthode des assertions invariantes due à Floyd et celle des assertions intermittentes due à Burstall pour démontrer la correction totale de programmes séquentiels avec instruction d'affectation aléatoire. Nous montrons par des exemples classiques comment présenter informellement les preuves formelles à l'aide de commentaires appropriés dans des programmes exécutables.

A HOARE-STYLE AXIOMATIZATION OF BURSTALL'S INTERMITTENT ASSERTIONS METHOD FOR NON-DETERMINISTIC PROGRAMS

Patrick COUSOT*

1. INTRODUCTION

Floyd[67]'s invariant assertions method for proving total correctness of sequential programs is based upon computational induction.

It offers the useful advantage that a total correctness proof can be decomposed into separate proofs of partial correctness, absence of run-time errors and termination. Hoare[69] introduced a further decomposition of these separate proofs using induction on the syntax of programs.

Hoare[69] described the partial correctness proof method by a logic in which the notion of proof is rigorously defined by a Hilbert-style formal system. This logic can be generalized to total correctness (see e.g. Manna & Pnueli[74]).

Another important property of Hoare[69]'s logic is that lengthy formal proofs can be made much more understandable by means of a proof outline in which the program is given with comments which are invariant assertions interleaved at entry and exit points of all program statements. The main advantage of proof outlines is that the program and its proof constitute a single piece of text.

Burstall[74]'s intermittent assertions method for proving total correctness of sequential programs is based upon structural induction on the data.

---

* Université de Metz, Faculté des Sciences, Ile du Saulcy, 57045 METZ Cedex, France.

This work was supported by ATP ADI-CNET-CNRS "Parallélisme,Communication, Synchronisation".

One criticism (Gries[79]) of its usual presentation (Manna & Waldinger[78]) is that all parts of the proof (viz. partial correctness, absence of run-time errors and termination) are packaged together. However proofs can be freely decomposed into arbitrarily chosen lemmas and theorems.

Almost syntax-directed axiomatizations of Burstall's method have been proposed using Dynamic Logic (Harel[79]), Temporal Logic (Apt & Delporte[83]), etc. However, because of the explicit reference to program points, lemmas and theorems need not correspond to the syntactic structure of the program.

Consequently, the defect in such formal systems is that proofs involving intermittent assertions must be presented separately from the program text. This is also the case of informal presentations based upon symbolic execution, proof lattices, proof diagrams, etc.

Floyd's and Burstall's methods have contrary advantages and drawbacks. Moreover Hoare's presentation of Floyd's method enjoys many advantages over known presentations of Burstall's method. In order to (partly) reconcile (sometimes antagonistic) qualities of both methods, we introduce a Hoare-style logic describing Floyd's and Burstall's total correctness proof methods in a unified manner.

To attain this end, a number of choices have been made :

- First of all, computational induction (hence Floyd's method) is understood as a special instance of structural induction on the data (hence of Burstall's method).
- The lemmas and theorems in Burstall's method will be restricted so as to correspond to the syntactic structure of programs (a limitation attenuated by the possible use of non-recursive parameterless procedures). Consequently, the explicit use of program points can be avoided because entry and exit points of program commands are implicitly referred to when using Hoare's notation for asserted programs.
- Intermittent assertions are needed only for loops. In this case, informal proofs by symbolic execution and induction can be formally explained by loop transformations. Moreover, induction can be understood as a termination argument so that (when considering programs with partial operations) the traditional decomposition of total correctness proofs into separate proofs of partial correctness, termination (and absence of run-time errors) is applicable to Burstall's method.

The paper is organized as follows :

In paragraph 2 we describe the considered programming language (nondeterministic imperative while-programs incorporating random assignments and non-recursive parameterless procedures) and the reasoning language which consists of the programming language augmented with labels which are used to designate loop bodies.

In paragraph 3 we informally describe the meaning of predicates (also called assertions) and asserted statements. The difference with Hoare's logic is that we consider total correctness. We also generalize Hoare's notation so as to be able to express that a given predicate should inevitably be satisfied after some (unknown) number of executions of a loop body.

In paragraph 4 we set out the proof system. It contains Hoare[69]'s proof system extended from partial to total correctness. Burstall[74]'s "hand simulation" is understood as the use of very simple semantic-preserving program equivalences. A rule of inference, designed in Hoare[69]'s style, provides for Burstall[74]'s proofs by "induction on the data".

Because of the structural induction rule, the proof system is not a classical Hilbert-style formal system. Hence formal proofs must be defined carefully in paragraph 5.

We give several examples of formal proofs. A program with unbounded nondeterminism due to Dijkstra[76] shows that induction upon non-negative integers may be adequate with Burstall's method whereas it is not with Floyd's method.

In paragraph 6 we show that formal proofs can be informally described as proof outlines. We consider the classical proof of Ackermann's function (Manna & Waldinger[78], Gries[79]) so as to concentrate on the presentation of a well-known proof using the logic. Finally we show a very simple program due to Dijkstra[77] for which no simple termination function that decreases monotonically has been found, which is not obtained by recursion elimination and has a trivial termination proof using Burstall's intermittent assertions method.

We conclude with empirical justifications of our choices in the design of the logic.

2. THE PROGRAMMING AND REASONING LANGUAGES

2.1 THE PROGRAMMING LANGUAGE PL

We consider simple non-deterministic imperative while-programs with random assignments. The programming language PL has five syntactic categories :

- PL-X a given set of program variables ranged over by X,
- PL-E a given set of expressions ranged over by E, not containing "?" and such that the (free) variables in E all belong to $FV(E) \subseteq$ PL-X,
- PL-B a given set of boolean expressions ranged over by B such that the (free) variables in B all belong to $FV(B) \subseteq$ PL-X,
- PL-N a given set of command names ranged over by N,
- PL-C a set of commands ranged over by C and with abstract syntax given by :

  $C ::= \underline{skip} \mid X:=? \mid X:=E \mid C_1 ; C_2 \mid \underline{if}\ B\ \underline{then}\ C_1\ \underline{else}\ C_2\ \underline{fi} \mid \underline{while}\ B\ \underline{do}\ C\ \underline{od} \mid N$

For our purpose it is not necessary to elaborate PL-X, PL-E, PL-B, PL-N. Also the abstract syntax does not cope with declarations. However it is assumed that each program variable X has a type written $\underline{Dom}[\![X]\!]$ and that each command name N unambiguously designates a unique command C (not containing N), a fact that we write N:C. For our purpose it is not necessary to specify how names are associated with commands. For instance, in the concrete syntax, one could use non-recursive parameterless procedures. Labels could also be used as shorthands for procedures only called once.

The random assignment X:=? assigns to X any value of the type $\underline{Dom}[\![X]\!]$ of X. We write $\underline{if}\ B\ \underline{then}\ C\ \underline{fi}$ for $\underline{if}\ B\ \underline{then}\ C\ \underline{else}\ \underline{skip}\ \underline{fi}$.

2.2 THE REASONING LANGUAGE RL

In Burstall[74]'s method, intermittent assertions can be restricted to while-loops. Hence one essentially has to express that "If P holds and $\underline{while}\ B\ \underline{do}\ C\ \underline{od}$ is executed then after some number of iterations Q must hold". In Hoare-style notation we will write $\{P\}L\{Q\}$ where L is a label designating the body of loop $\underline{while}\ B\ \underline{do}\ C\ \underline{od}$, a fact that will be denoted

while L:B do C od. Then a total correctness proof {P} while B do C od {Q} (i.e. "if P holds and while B do C od is executed, then execution terminates with Q true") by Burstall[74]'s method essentially consists in (repeatedly and usually by induction) proving the existence of an intermediate intermittent assertion R such that {P}L{R} and {R} while B do C od {Q} hold. Consequently overloading the sequential composition meaning of semi-colon, this fact will be written {P} L;while B do C od {Q}.

The above discussion leads to the consideration of a reasoning language RL which has the syntactic categories of PL and

- RL-L a given set of loop labels ranged over by L,
- RL-S a set of statements ranged over by S and with abstract syntax :
  S ::= C|L|S;S

For simplicity we write while L:B do C od to mean that L designates the body of loop while B do C od.

The distinction between programming and reasoning languages comes from the fact that we can define a (trivially implementable) operational semantics for commands but not for loop labels.

3. THE LOGICAL LANGUAGE LL

The logical language LL contains a set LL-P of predicates (also called assertions) to describe program states and asserted statements of the form {P}S{Q}.

3.1 PREDICATES

The logical language LL includes a first order language such that :

. The set of variables of LL is partitionned into the set PL-X (ranged over by X) of programming variables and a set LL-x (ranged over by x) of logical variables. There is an unlimited provision of logical variables,

. The terms (T...) of LL include PL-E,

. The atomic formulas of LL include equality of terms and the truth values true and false,

. The set LL-P of predicates of LL with typical elements P,Q,..., contains PL-B. It is closed under use of the logical connectives ¬(not), ∧(and), ∨(or), ⇒(implies) and quantifiers over logical (but never programming) variables ∀(for all) and ∃(there exists).

We are faced with the problem that expressions and boolean expressions should always be well-defined in the logical language LL but should contain partial operations in the programming language PL. Among the possible solutions, we have chosen to have only total operations in LL and to model a partial operation of PL by a total operation of LL together with a predicate characterizing its domain. For example integer division / may be defined as a total operation in the logical language LL (if for instance we let 1/0 be the undefined value ⊥). When used in the programming language PL integer division is a partial operation so that we have Dom⟦X/Y⟧= (min≤X≤max ∧ min≤Y≤max ∧ Y≠0 ∧ min≤ X/Y ≤max).

More generally, a predicate Dom⟦X⟧, called the type of X, characterizes the set of values which can be assigned to X in PL. The only free variable of Dom⟦X⟧ which is a programming variable is X.

To each expression E ∈ PL-E corresponds a predicate Dom⟦E⟧ ∈ LL-P characterizing the domain of definition of E in PL. All free variables in

Dom⟦E⟧ which are programming variables should appear in E. Similarly the domain of B ∈ PL-B is characterized by Dom⟦B⟧.

3.2 THE INFORMAL MEANING OF ASSERTED STATEMENTS

An asserted statement is a triple {P}S{Q}. It informally means that execution of S from a state satisfying P inevitably leads to a state satisfying Q. More precisely :

. If C ∉ PL-N then {P}C{Q} is the assertion that "if P is true of the (vector of) values v of the logical variables and initial values $\underline{V}$ of the program variables and command C is executed then execution will terminate without run-time errors and after execution of C is complete Q will be true of v and the final values $\overline{V}$ of the program variables". Otherwise stated {P}C{Q} asserts that C is totally correct with respect to pre-condition P and post-condition Q.

. If N:C then {P}N{Q} stands for {P}C{Q}.

. If while L:B do C od then {P}L{Q} is the assertion that "if P is true of the values of v of the logical variables and initial values $\underline{V}$ of the program variables and command while B do C od is executed, then execution will proceed properly until, after zero or more executions of the loop body, reaching a state $\overline{V}$ of the program variables such that Q holds for v and $\overline{V}$ ". (Paraphrasing Manna & Waldinger[74], {P}L{Q} asserts that if control is at L with P true then sometime later control will be at L with Q true. Notice that L can be understood as designating the control point within the loop body just before the test B).

. $\{P\}S_1;S_2\{Q\}$ is a shorthand for the existence of R such that $\{P\}S_1\{R\}$ and $\{R\}S_2\{Q\}$ hold.

(Observe that we have given two different definitions of $\{P\}C_1;C_2\{Q\}$. Because they are equivalent when $C_1, C_2 \in$ PL-C, we have chosen to overload the meaning of semi-colon).

4. THE PROOF SYSTEM

We use the following notations :

. If ϕ is a term or predicate of LL then FV(ϕ) is the vector of free variables of ϕ, BV(ϕ) is the set of bound variables of ϕ and ϕ^T_w stands for the result of substituting term T for all free occurences of variable $w \in (PL\text{-}X \cup LL\text{-}x)$ (if necessary, after renaming of the bound variables in ϕ so that FV(T) $\cap$ BV(ϕ) is empty).

. We write Wf($\prec$) to mean that the infix relation symbol $\prec$ should be interpreted as a well-founded relation (i.e. there is no sequence $\{a_n\}$ such that $a_{n+1} \prec a_n$ for all n).

. $v, \underline{v}, \ldots$ stand for vectors of logical or programming variables. For vectors $v, \underline{v}$ of length $|v| = |\underline{v}| = \ell$ we write $v = \underline{v}$ as a shorthand for $((v_1 = \underline{v}_1) \wedge \ldots \wedge (v_\ell = \underline{v}_\ell))$.

The proof system contains a version of Hoare[69]'s proof system extented to random assignments and from partial to total correctness in the presence of partial operations.

. Null command :

(N) $\quad \{P\}\ \underline{skip}\ \{P\}$

. Assignment command :

(A) $\quad \{\underline{Dom}[\![E]\!] \wedge (\underline{Dom}[\![X]\!] \wedge P)^E_X\}\ X:=E\ \{P\}$

. Random assignment :

(R) $\quad \{\exists x.\underline{Dom}[\![X]\!]^x_X \wedge \forall x.([\underline{Dom}[\![X]\!] \Rightarrow P]^x_X)\}\ X:=?\ \{P\}$

(The effect of X:=? is to assign to X some value x of type $\underline{Dom}[\![X]\!]$. Execution fails when $\underline{Dom}[\![X]\!]$ is empty. Else, P holds after execution if and only if it is true of all values x in $\underline{Dom}[\![X]\!]$ which can be assigned to X).

. Sequential composition :

(SC) $\quad \dfrac{\{P\}S_1\{Q\}\ ,\ \{Q\}S_2\{R\}}{\{P\}\ S_1;S_2\ \{R\}}$

(Observe that Hoare[69] propounded this rule of inference only for commands. However it can be consistently extended to loop labels hence statements by definition of the connector ";" of statements).

. Alternative composition :

(AC) $$\frac{\{P \wedge \underline{Dom}[\![B]\!] \wedge B\}\ C_1\ \{Q\} \quad , \quad \{P \wedge \underline{Dom}[\![B]\!] \wedge \neg B\}\ C_2\ \{Q\}}{\{P \wedge \underline{Dom}[\![B]\!]\}\ \underline{\text{if}}\ B\ \underline{\text{then}}\ C_1\ \underline{\text{else}}\ C_2\ \underline{\text{fi}}\ \{Q\}}$$

Since $\underline{\text{if}}\ B\ \underline{\text{then}}\ C\ \underline{\text{fi}}$ stands for $\underline{\text{if}}\ B\ \underline{\text{then}}\ C\ \underline{\text{else}}\ \underline{\text{skip}}\ \underline{\text{fi}}$, the following rule of inference can be derived from the null command axiom, the tautology $Q \Rightarrow Q$ and the consequence rule :

(AC') $$\frac{\{P \wedge \underline{Dom}[\![B]\!] \wedge B\}\ C\ \{Q\} \quad , \quad (P \wedge \underline{Dom}[\![B]\!] \wedge \neg B) \Rightarrow Q}{\{P \wedge \underline{Dom}[\![B]\!]\}\ \underline{\text{if}}\ B\ \underline{\text{then}}\ C\ \underline{\text{fi}}\ \{Q\}}$$

. Iterative composition :

(IC) $$\frac{\{P \wedge \underline{Dom}[\![B]\!] \wedge B \wedge v = v'\}\ C\ \{P \wedge \underline{Dom}[\![B]\!] \wedge v \prec v'\}}{\{P \wedge \underline{Dom}[\![B]\!]\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ C\ \underline{\text{od}}\ \{P \wedge \neg B\}}$$

when $|v| = |v'|$, $v' \subseteq LL.x$ and $Wf(\prec)$

(As usual total correctness follows from the fact that execution of the loop body leaves $P \wedge \underline{Dom}[\![B]\!]$ invariant and decreases v with respect to the well-founded relation $\prec$).

. Consequence rule :

(CR) $$\frac{P \Rightarrow P' \quad , \quad \{P'\}S\{Q'\} \quad , \quad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

. Or rule :

(OR) $$\frac{\{P_1\}S\{Q_1\} \quad , \quad \{P_2\}S\{Q_2\}}{\{P_1 \vee P_2\}\ S\ \{Q_1 \vee Q_2\}}$$

. And rule :

(AR) $$\frac{\{P_1\}C\{Q_1\} \quad , \quad \{P_2\}C\{Q_2\}}{\{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

(Observe that contrary to (CR) and (OR), the And rule cannot be extended to loop labels hence statements. For example we have $\{X=0\}L\{X=0\}$ and $\{X=0\}L\{X=1\}$ for while L:true do X:=X+1 od but not $\{X=0\}\,L\,\{(X=0)\wedge(X=1)\}$).

. Command name rule :

$$\text{(CN)}\quad \frac{\{P\}C\{Q\}}{\{P\}N\{Q\}}$$

when N:C.

. Replacement rule :

$$\text{(RR)}\quad \frac{\{P\}S\{Q\}}{\{(\exists x'.P^{x'}_{x})^{T}_{y}\wedge R\}\ S\ \{Q^{T}_{y}\wedge R\}}$$

when $x\in LL\text{-}x$, $x\notin FV(Q)$, $y\in LL\text{-}x$, $FV(T)\subseteq LL\text{-}x$ and $FV(R)\subseteq LL\text{-}x$.

(The predicate R only depends upon logical variables hence it is invariant because their values cannot be modified by program execution. A logical variable y stands for any logical term T (i.e. a logical variable, a constant or an application of a function to logical terms). A logical variable $x\in LL\text{-}x$ designating a value upon which the final result does not depend ($x\notin FV(Q)$) can be eliminated.

This rule is equivalent to the invariance, elimination, substitution I and substitution II rules of Apt[81]).

We now extend the above Hoare-style proof system by considering proof rules relative to Burstall[74]'s intermittent assertions method. Two ingredients of Burstall's method can be handled by the above classical part of the logic. Firstly, proofs can be decomposed into theorems and lemmatas corresponding to the syntax of programs. These theorems and lemmata can be given very concise formulations by giving names to commands and using the command name rule. Secondly, initial or intermediate values of the program variables can be given symbolic names by means of auxiliary logical variables and handled by the replacement rule. The two last ingredients of Burstall's method are "hand-simulation" and "induction upon the data".

"Hand-simulation" or "symbolic execution" will be understood from the equivalent (but time-independent) point of view of program equivalences. In fact since hand-simulation (and induction) is only necessary to handle while-loops, the only program equivalence to be considered is
while B do C od ≡ if B then C fi;while B do C od or using this transformation recursively while B do C od ≡
(if B then C fi)^n;while B do C od for all n≥0 where C^n is skip when n=0, is C when n=1 else is $C^m;C^\ell$ when n=m+ℓ. By hand-simulation and using an intermediate intermittent assertion R, a proof {P} while B do C od {Q} can be decomposed into {P} (if B then C fi)^n {R} and {R} while B do C od {Q}. In order to avoid awkward manipulations of large pieces of program text we use loop labels writing {P}L{R} with while L:B do C od instead of {P} (if B then C fi)^n {R} :

· No iteration axiom :

(NI) $\quad \{P\}L\{P\}$

· One iteration rule :

(OI) $$\frac{\{P\}\ \text{if B then C fi}\ \{Q\}}{\{P\}L\{Q\}}$$

when while L:B do C od

· Multiple iterations rule :

(MI) $$\frac{\{P\}\ L;L\ \{Q\}}{\{P\}L\{Q\}}$$

· Hand-simulation rule :

(HS) $$\frac{\{P\}\ L;\text{while B do C od}\ \{Q\}}{\{P\}\ \text{while B do C od}\ \{Q\}}$$

when while L:B do C od

One must ultimately be able to handle the while-loop in the premise of the hand-simulation rule. Using (IC) would not be in the spirit of Burstall[74]. Hence we introduce the following axiom scheme to serve as a conclusion of hand-simulation :

· Escape while axiom :

(EW) $\{P \wedge \underline{Dom}[\![B]\!] \wedge \neg B\}$ while B do C od $\{P\}$

(This axiom scheme can be derived from (IC) using first order tautologies, $\{\underline{false}\}C\{\underline{false}\}$ which can be proved by induction on the syntax of commands and (CR). Alternatively (IC) can be dispensed with and derived from the other axiom schemata and rules of inference).

Using the previous axioms and rules of inference one can prove programs by "hand-simulation". As stated by Burstall[74] "Since we cannot write out the whole infinite symbolic computation explicitly we will use mathematical induction to prove general statements about what happens at a loop point".

Structural induction can be introduced in several ways within the logic. One can consider an induction axiom scheme (e.g. 4.10 in Harel[79]) providing for the proof of an arbitrary formula of the logic by structural induction such as

$\forall v.[(\forall v'.[(v' \prec v) \Rightarrow \phi(v')]) \Rightarrow \phi(v)] \Rightarrow \forall v.\phi(v)$ where $Wf(\prec)$.

Such an approach would significantly depart from the style of Hoare's logics where quantified asserted programs are not considered. Therefore we will consider a more restricted form of structural induction limited to induction upon the data of an asserted statement :

· Structural induction rule :

(SI) $$\frac{\{v' \prec v \wedge P_v^{v'}\}S\{Q_v^{v'}\} \vdash \{P\}S\{Q\}}{\{P\}S\{Q\}}$$

when $|v| = |v'|$, $v \subseteq LL\text{-}x$, $v' \subseteq LL\text{-}x$ and $Wf(\prec)$.

As usual $\frac{A \vdash B}{C}$ is a rule of inference which permits deduction of C if B is proved; however it also permits A to be assumed as hypothesis in the proof of B.

The intuition behind this rule is that in the proof of the lemma $\{P\}S\{Q\}$ for (initial) values v (of, say, the program variables), we can assume $\{P_v^{v'}\}S\{Q_v^{v'}\}$ when $v' \prec v$ that is to say that the lemma holds for smaller (initial) values v' (of the program variables). Observe that when $v' \prec v$ holds $\{P_v^{v'}\}S\{Q_v^{v'}\}$ is equivalent to $\{v' \prec v \wedge P_v^{v'}\}S\{Q_v^{v'}\}$. When $v' \prec v$ does not hold, no induction hypothesis can be made. Yet, in the case, the structural induction rule allows us to make the hypothesis $\{v' \prec v \wedge P_v^{v'}\}S\{Q_v^{v'}\}$ that is to say $\{\underline{\text{false}}\}S\{Q_v^{v'}\}$. This is correct because $\{\underline{\text{false}}\}S\{Q_v^{v'}\}$ is a tautology of the logic so that it can be assumed without harm (nor help) in any proof.

5. FORMAL PROOFS (WITH EXAMPLES)

5.1 DEFINITION

We write $H,W \vdash \{P\}S\{Q\}$ if and only if there is a proof of $\{P\}S\{Q\}$ from the set H of hypotheses and finite set W of well-founded relations.

Since the proposed proof system is not a usual Hilbert-style system in the sense of first order logic, we must define formal proofs rigorously.

A sequent is a pair $\psi \vdash \phi$ (read ψ yields ϕ) where the members ψ_j of the antecedent $\psi=\psi_1,\ldots,\psi_k$ and the succedent ϕ are formulas (i.e. predicates or asserted statements).

A proof of Φ from H,W is a finite sequence $\psi^1 \vdash \phi^1,\ldots,\psi^\ell \vdash \phi^\ell$ of sequents with ψ^ℓ empty and $\phi^\ell=\Phi$ such that each sequent $\psi^n \vdash \phi^n$, $n=1,\ldots,\ell$ either

- has ψ^n empty and ϕ^n is an axiom of the proof system or belongs to H,
- has $\psi^n=\phi^n$ and ϕ^n is (an induction hypothesis) of the form $\{v' \prec v \wedge P_v^{v'}\}S\{Q_v^{v'}\}$ with $\prec \in W$,
- derives from earlier sequents $\psi^{m_1} \vdash \phi^{m_1},\ldots,\psi^{m_i} \vdash \phi^{m_i}$ by the rule of inference $\dfrac{\phi^{m_1},\ldots,\phi^{m_i}}{\phi^n}$ and any given expression is a member of ψ^n if and only if it is a member of one of the antecedents $\psi^{m_1},\ldots,\psi^{m_i}$,
- derives from an earlier sequent $\psi^{n'} \vdash \phi^{n'}$, $n'<n$ with $\psi^{n'} = \psi^n \cup \{ih\}$, $\phi^{n'}=\phi^n$ and $\dfrac{ih \vdash \phi^n}{\phi^n}$ is a rule of inference of the proof system.

5.2 EXAMPLE 1

We formally rephrase Burstall[74]'s proof of the following program which computes 2^n :

```
Power:  P:=1;
        while Loop: N>0 do
            P:=2xP; N:=N-1
        od
```

Following Burstall[74], we must show by upward induction on i that $\{0\le i\le n \wedge N=n \wedge P=p\}loop\{N=n-i \wedge P=px2^i\}$ holds so as to conclude $H,W \vdash \{N=n\ge 0\}Power\{P=2^n\}$. We choose $W=\{\prec\}$ with $(i',n') \prec (i,n)$ if and only if $0\le i'<i\le n'=n$. For simplicity we choose H as the set of all theorems of arithmetic.

For the sake of conciseness obvious steps of the formal derivation are omitted :

(1) $\vdash \{0=i\leq n \wedge N=n \wedge P=p\}$ Loop $\{N=n-i \wedge P=px2^i\}$ NI,CR

(2) ih $\vdash$ ih, where

$ih=\{(j,n)\prec(i,n) \wedge (0\leq i\leq n \wedge N=n \wedge P=p)^{j,n}_{i,n}\}$ Loop $\{(N=n-i \wedge P=px2^i)^{j,n}_{i,n}\}$

$=\{0\leq j<i\leq n \wedge N=n \wedge P=p\}$ Loop $\{N=n-j \wedge P=px2^j\}$

(3) ih $\vdash \{1\leq i\leq n \wedge N=n \wedge P=p\}$ Loop $\{N>0 \wedge N=n-(i-1) \wedge P=px2^{(i-1)}\}$ 2,RR,CR

(4) $\vdash \{N>0 \wedge N=n-(i-1) \wedge P=px2^{(i-1)}\}$ if N>0 then P:=2xP;N:=N-1 fi $\{N=n-i \wedge P=px2^i\}$ A,SC,AC',CR

(5) $\vdash \{N>0 \wedge N=n-(i-1) \wedge P=px2^{(i-1)}\}$ Loop $\{N=n-i \wedge P=px2^i\}$ 4,OI

(6) ih $\vdash \{1\leq i\leq n \wedge N=n \wedge P=p\}$ Loop;Loop $\{N=n-i \wedge P=px2^i\}$ 3,5,SC

(7) ih $\vdash \{1\leq i\leq n \wedge N=n \wedge P=p\}$`Loop $\{N=n-i \wedge P=px2^i\}$ 6,MI

(8) ih $\vdash \{0\leq i\leq n \wedge N=n \wedge P=p\}$ Loop $\{N=n-i \wedge P=px2^i\}$ 1,7,OR,CR

(9) $\vdash \{0\leq i\leq n \wedge N=n \wedge P=p\}$ Loop $\{N=n-i \wedge P=px2^i\}$ 8,SI

(10) $\vdash \{N=n\geq 0 \wedge P=1\}$ Loop $\{N=0 \wedge P=2^n\}$ 9,RR,CR

(11) $\vdash \{N=0 \wedge P=2^n\}$ while N>0 do P:=2xP;N:=N-1 od $\{P=2^n\}$ EW,CR

(12) $\vdash \{N=n\geq 0 \wedge P=1\}$ Loop;while N>0 do P:=2xP;N:=N-1 od $\{P=2^n\}$ 10,11,SC

(13) $\vdash \{N=n\geq 0 \wedge P=1\}$ while N>0 do P:=2xP;N:=N-1 od $\{P=2^n\}$ 12,HS

(14) $\vdash \{N=n\geq 0\}$ Power $\{P=2^n\}$ A,13,SC,CR,CN

5.3 EXAMPLE 2

Consider Dijkstra[76]'s program with unbounded nondeterminism (on page 77) :

```
U: while L: X≠0 do
     if X<0 then
        Y:=?;X:=Y
     else
        X:=X-1
     fi
   od
```

when Dom⟦Y⟧=(Y≥0). We shall prove H,W $\vdash$ {true}U{X=0} with W={≺} such that m≺n if and only if 0<m<n and H is the set of all theorems of arithmetic. Obvious steps of the proof are omitted.

We first prove that execution of the loop body inevitably leads to a state with X≥0.

(1) $\vdash \{X<0\}$ if $X\neq 0$ then if $X<0$ then $Y:=?;X:=Y$ else $X:=X-1$ fi fi $\{X\geq 0\}$

(2) $\vdash \{X<0\}\ L\ \{X\geq 0\}$ 1,OI

(3) $\vdash \{X\geq 0\}\ L\ \{X\geq 0\}$ NI

(4) $\vdash \{\underline{true}\}\ L\ \{X\geq 0\}$ 2,3,OR,CR

Then we show by induction on the initial value n of X that execution of the loop starting with $X=n\geq 0$ inevitably leads to a state with $X=0$. In the following ih stands for $\{m<n \wedge P_n^m\}\ L\ \{Q_n^m\}$ with $P=(X\geq 0 \wedge X=n)$ and $Q=(X=0)$.

(5) $\vdash \{X\geq 0 \wedge X=n\}$ if $X\neq 0$ then if $X<0$ then $Y:=?;X:=Y$ else $X:=X-1$ fi fi $\{(X=0)\vee(X>0 \wedge X=n-1)\}$

(6) $\vdash \{X\geq 0 \wedge X=n\}\ L\ \{(X=0)\vee(X>0 \wedge X=n-1)\}$ 5,OI

(7) $\vdash \{X=0\}\ L\ \{X=0\}$ NI

(8) $ih \vdash ih$

(9) $ih \vdash \{(m<n \wedge X\geq 0 \wedge X=m)_m^{n-1}\}\ L\ \{X=0\}$ 8,RR,CR

(10) $ih \vdash \{X>0 \wedge X=n-1\}\ L\ \{X=0\}$ 9,CR

(11) $ih \vdash \{(X=0)\vee(X>0 \wedge X=n-1)\}\ L\ \{X=0\}$ 7,10,OR,CR

(12) $ih \vdash \{X\geq 0 \wedge X=n\}\ L;L\ \{X=0\}$ 6,11,SC

(13) $ih \vdash \{X\geq 0 \wedge X=n\}\ L\ \{X=0\}$ 12,MI

(14) $\vdash \{X\geq 0 \wedge X=n\}\ L\ \{X=0\}$ 13,SI

The proposition easily follows from the previous two lemmas :

(15) $\vdash \{\exists n'.\ X\geq 0 \wedge X=n'\}\ L\ \{X=0\}$ 14,RR

(16) $\vdash \{\underline{true}\}\ L;L\ \{X=0\}$ 4,15,CR,SC

(17) $\vdash \{\underline{true}\}\ L\ \{X=0\}$ 16,MI

(18) $\vdash \{X=0\}$ while $X\neq 0$ do if $X<0$ then $Y:=?;X:=Y$ else $X:=X-1$ fi od $\{X=0\}$ EW,CR

(19) $\vdash \{\underline{true}\}\ L;$while $X\neq 0$ do if $X<0$ then $Y:=?;X:=Y$ else $X:=X-1$ fi od $\{X=0\}$ 17,18,SC

(20) $\vdash \{\underline{true}\}\ U\ \{X=0\}$ 19,HS,CN

Observe that the proof of lemma $\{X\geq 0 \wedge X=n\}\ L\ \{X=0\}$ can be done using the natural ordering on non-negative integers. The ability to state this lemma dispense with the use of a well-ordering of higher order than natural numbers which would be necessary when using (IC).

6. PROOF OUTLINES (WITH EXAMPLES)

Formal proofs can be made much more understandable by giving a proof outline in which the proof of each lemma and theorem is presented in a standard form as a program annotated with assertions.

6.1 INFORMAL DEFINITION

Proof outlines for null, assignment, random assignment commands and sequential, alternative composition are classical.

When using the consequence rule (CR) we write :

```
{P}
  {P'}
    S
  {Q'}
{Q}
```

and omit P or P' (and Q' or Q) when the consequence $P \Rightarrow P'$ (and $Q' \Rightarrow Q$) is obvious. The use of the replacement rule is presented in the same way but without omissions :

$$\{(\exists x'.P^{x'}_{x})^{T}_{y} \wedge R\}$$
$$\{P\}$$
$$S$$
$$\{Q\}$$
$$\{Q^{T}_{y} \wedge R\}$$

The use of the And-rule is superfluous whereas the Or-rule is only needed for à la Burstall proofs of while loops.

In our examples commands will be named using labels so that the use of the command name rule will be presented as :

```
N: {P}
     C
   {Q}
```

The proof outline for while-loops using (IC) is classical. When using Burstall[74]'s method the total correctness proof of a loop {P} while L:B do C od {Q} is better handled by a lemma {P'}L{Q'} from which using (RR) and (CR) one derives $\{P\}L\{Q \wedge \underline{Dom}[\![B]\!] \wedge \neg B\}$ and concludes by (EW), (SC) and (HS). This proof can be outlined (within an executable program) as follows :

$\{P\}$
$\{(\exists x'.P'^{x'}_{x})^{T}_{y} \wedge R\}$
$\{L: \{P'\}$
...
$\{Q'\}\}$
$\{Q'^{T}_{y} \wedge R\}$
$\{Q \wedge Dom[\![B]\!] \wedge \neg B\}$
while $\{L:\}$ B **do**
C
od
$\{Q\}$

The proof of the lemma $\{P'\}L\{Q'\}$ can be done by hand-simulation (using (NI),(OI)) and induction (using (SI)) and decomposed into a number of disjoint cases (later merged using (OR)). Such a proof can always be presented as :

$\{L: \{P'\}$
$\{P_0^1 \vee \ldots \vee P_0^{\ell_0}\}$
$S_0;$
$\{P_1^1 \vee \ldots \vee P_1^{\ell_1}\}$
...
$S_{n-1};$
$\{P_n^1 \vee \ldots \vee P_n^{\ell_n}\}$
$\{Q'\}\}$

where for $i=0,\ldots,n-1$, $j=1,\ldots,\ell_i$, S_i is either *if* B *then* C *fi* in which case $\{P_i^j\}L\{P_{i+1}^k\}$ holds for some $k \in 1,\ldots,\ell_{i+1}$ by ((NI) and (CR)) or (OI), or else, S_i is L in which case $\{P_i^j\}L\{P_{i+1}^k\}$ holds for some $k \in 1,\ldots,\ell_{i+1}$ by ((NI) and (CR)) or by induction hypothesis i.e. (SI) (plus (RR),(CR)) applied to lemma $\{P'\}L\{Q'\}$. Then obviously, $\{P_i^1 \vee \ldots \vee P_i^{\ell_i}\}$ L $\{P_{i+1}^1 \vee \ldots \vee P_{i+1}^{\ell_{i+1}}\}$ follows by (OR) so that by successive applications of (MI) we derive $\{P_0^1 \vee \ldots \vee P_0^{\ell_0}\}$ L $\{P_n^1 \vee \ldots \vee P_n^{\ell_n}\}$ and by (CR) we conclude $\{P'\}L\{Q'\}$.

Finally, proofs of partial correctness ([...]), clean behavior ($\not\in\ldots\not\ni$) and termination ($\not\in\ldots\ni$) can be made separately and the relevant assertions enclosed within distinct parentheses.

The relative completeness proof of Cousot[83] shows that a proof exists for all totally correct programs that can be presented as indicated above.

6.2 EXAMPLE 1

We consider the outline of the formal total correctness proof of Burstall[74]'s program which computes 2^n given at paragraph 5.2. We also outline a clean termination proof for implementations of natural numbers between 0 and max, with max≥1 :

```
Power : {[N=n ∧ n≥0] ∧ ∉ 2^n≤max ∌ }
        P:=1;
        {[N=n ∧ n≥0 ∧ P=1] ∧ ∉ 2^n≤max ∌ }
         {Loop : {[0≤i≤n ∧ N=n ∧ P=p] ∧ ∉ 0<px2^n≤max ∌ }
                 {V([0=i≤n ∧ N=n ∧ P=p])
                   ([1≤i≤n ∧ N=n ∧ P=p] ∧ ∉ 0<px2^n≤max ∌ ∧ {0≤i-1<i})}
                 Loop;
                 {V([0=i≤n ∧ N=n ∧ P=p])
                   ([1≤i≤n ∧ N=n-(i-1) ∧ P=px2^(i-1)] ∧ ∉ 0<px2^n≤max ∌ )}
                 if N>0 then
                    { ∉ 0<2xP≤max ∧ 0≤N-1≤max ∌ }
                    P:=2xP; N:=N-1
                 fi;
                 {[N=n-i ∧ P=px2^i]}}
        {[N=0 ∧ P=2^n]}
        while {Loop :} N>0 do
           P:=2xP; N:=N-1
        od
        {[P=2^n]}
```

6.3 EXAMPLE 2

We consider Manna & Waldinger[78]'s iterative algorithm :

```
Ack : S:=<M,N>;
      while size(S)≠1 do
         if S(2)=0 then
            S:=S(...3)|S(1)+1
         elsif S(1)=0 then
            S:=S(...3)|S(2)-1|1
         else
            S:=S(...3)|S(2)-1|S(2)|S(1)-1
         fi
      od
```

to compute Ackermann's function, defined by :

$$A(a,b) = \begin{cases} a=0 & \rightarrow\ b+1 \\ a\neq 0,\ b=0 & \rightarrow\ A(a-1,1) \\ a\neq 0,\ b\neq 0 & \rightarrow\ A(a-1,A(a,b-1)) \end{cases}$$

We follow Gries[79]'s notations so that the above algorithm to compute A(M,N) uses a "sequence" variable S of the form $S=\langle S_n,\ldots,S_2,S_1\rangle$ where $n=\underline{size}(S)\geq 0$. Element S_i of S is referenced within the algorithm by S(i), while S(...i) refers to the possibly empty sequence $\langle S_n,S_{n-1},\ldots,S_i\rangle$. Operation S|X denotes the concatenation of the value of X to the right of sequence S.

The lexicographic ordering on pairs of non-negative integers is $<_2$. Gries[79]'s proof can be outlined as follows :

```
Ack : {M≥0 ∧ N≥0}
      S:=<M,N>;
      {S=<M,N> ∧ M≥0 ∧ N≥0}
       {Loop : {S=s|a|b ∧ a≥0 ∧ b≥0}
               if size(S)≠1 then
                  {S=s|a|b ∧ a≥0 ∧ b≥0}
                  if S(2)=0 then
                     {S=s|a|b ∧ a=0 ∧ b≥0}
                     S:=S(...3)|S(1)+1
                     {S=s|b+1 ∧ a=0 ∧ b≥0}
                     {S=s|A(a,b) ∧ a=0 ∧ b≥0}
                  elsif S(1)=0 then
                     {S=s|a|b ∧ a>0 ∧ b=0}
                     S:=S(...3)|S(2)-1|1
                     {S=s|a-1|1 ∧ a>0 ∧ b=0}
                  else
                     {S=s|a|b ∧ a>0 ∧ b>0}
                     S:=S(...3)|S(2)-1|S(2)|S(1)-1
                     {S=s|a-1|a|b-1 ∧ a>0 ∧ b>0}
                  fi
               fi;
               {∨(S=s|A(a,b) ∧ a=0 ∧ b≥0)
                ∨([S=s|a-1|1 ∧ a>0 ∧ b=0] ∧ {<a-1,1> <2 <a,b>})
                ∨([S=s|a-1|a|b-1 ∧ a>0 ∧ b>0] ∧ {<a,b-1> <2 <a,b>})}
               Loop;
               {∨(S=s|A(a,b) ∧ ((a=0 ∧ b≥0)∨(a>0 ∧ b=0)))
                ∨([S=s|a-1|A(a,b-1) ∧ a>0 ∧ b>0] ∧ {<a-1,A(a,b-1)> <2 <a,b>})}
               Loop;
               {∨(S=s|A(a,b) ∧ ((a=0 ∧ b≥0)∨(a>0 ∧ b=0)))
                ∨(S=s|A(a-1,A(a,b-1)) ∧ a>0 ∧ b>0)}
               {S=s|A(a,b)}}
      {S=<A(M,N)>}
      while {Loop :} size(S)≠1 do
         if S(2)=0 then
            S:=S(...3)|S(1)+1
         elsif S(1)=0 then
            S:=S(...3)|S(2)-1|1
         else
            S:=S(...3)|S(2)-1|S(2)|S(1)-1
         fi
      od
      {S=<A(M,N)>}
```

6.4 EXAMPLE 3

A termination proof of the program :

```
while X>1 do
   if odd(X) then X:=X+1 else X:=X/2 fi
od
```

is given by Dijkstra[77] using the following variant function t of the binary representation x of the value of X :

t(x) = 1 + the number of significant digits of x + the number of "internal" 0's of x (i.e. between the most- and the least-significant 1's), decreased by 2 if and only if x is a power of 2.

Because of the ability offered by hand-simulation to look for a quantity strictly decreased after one iteration when X is even or after two iterations when X is odd, this program has a trivial termination proof by Burstall's method which can be outlined as follows :

```
{true}
  {∃n.X=n}
    {L: {X=n}
     if X>1 then if odd(X) then X:=X+1 else X:=X/2 fi fi;
     {(X≤1)∨(even(n) ∧ n≥2 ∧ X=n/2)∨
          (odd(n) ∧ n≥3 ∧ X=n+1)}
          if X>1 then if odd(X) then X:=X+1 else X:=X/2 fi fi;
     {(X≤1)∨([X=n/2] ∧ {0<n/2<n})∨
          ([X=(n+1)/2]∧{0<(n+1)/2<n})}
     L;
     {X≤1}}
while X>1 do
  if odd(X) then X:=X+1 else X:=X/2 fi
od
{true}
```

7. CONCLUDING DISCUSSION ON THE DESIGN OF THE LOGIC

We have proposed a Hoare-style logic axiomatizing Burstall's intermittent assertions method. We think that the formalization is faithful to the design of the originators of the method. Since they have proposed no accurate definition and a good many of dissimilar descriptions of the intermittent assertions method have been suggested, this claim can only be based on the fact that we have been able to work out all examples considered in Burstall[74] and Manna & Waldinger[78].

However our formalization is not as flexible as some might like. This is because we deliberately restrict Burstall's lemmas of the general form "if sometime P at ℓ then sometime Q at ℓ' " to the particular case when ℓ is the entry point of a command C and ℓ' its exit point. This restriction is acceptable because contrary to Apt & Delporte[83] we do not consider arbitrary liveness properties of arbitrary programs but total correctness of well-structured programs. This choice has several advantages. By using Hoare-style notation {P}C{Q} there is no need to mention program control points. Also the formal system leads to syntax directed proofs such that the decomposition of the proof by syntactic program units exactly correspond to the logical structure of the proof decomposed into lemmas and theorems. Moreover since Hoare's logic is included in the formal system there is no need for two separate proof systems, one for invariance the other for liveness properties. Finally this homogeneity reduces the number of supplementary axioms and rules of inference to be remembered (i.e. essentially six : (NI), (OI), (MI), (EW), (HS), (SI)).

Our extension {P}S{Q} of Hoare's notation P{C}Q cannot be totally conformable with the original definition. This is because invariant and intermittent assertions are of different nature.

We have considered, and later abandoned the alternative which consists in using different notations for invariant and intermittent assertions. The idea is essentially to write {P} while B do C od {Q} for total correctness and {P} while B do C od [Q] to mean that execution of while B do C od with precondition P inevitably leads to Q (a fact that we have written {P}L{Q} with while L:B do C od). The disadvantage of using different notations for invariant and intermittent assertions is that the proof system

is considerably longer (although equivalent, for example (SC) must be split into four cases). Also proof outlines drag on.

Once the decision of merging notations for invariant and intermittent assertions is taken, the problem of expressing {P} while B do C od [Q] must be solved. Our use of a loop-label may seem a bit awkward. Its only merit is conciseness, particularly for proof outlines.

Our decision of using Hoare's notation for both invariant and intermittent assertions is,(at first glance somewhat surprisingly),consistent with Hoare's original definition. To show this, we consider Dijkstra[76]'s definition of wp(C,Q) (i.e. the weakest-precondition for the initial state such that activation of command C will properly terminate, leaving a final state satisfying the post-condition Q). In order to conform to Dijkstra[76]'s original definition we consider bounded nondeterminism (alternatively, we could consider a less restrictive definition of wp which captures weak instead of strong termination). We have {P}C{Q} if and only if $P \Rightarrow$ wp(C,Q) which implies Hoare's P{C}Q (viz. total implies partial correctness). The case of {P}L{Q} where L is a loop body label while L:B do C od was not previously considered by Hoare, hence had no pre-existent meaning. This is the only case when this meaning involves intermittent assertions (i.e. Q will hold after some number of iterations (but not necessarily after any number of iterations)). Clearly, loops are the only case when intermittent assertions are really needed so that the extension is minimal. Also the relationship to weakest-preconditions is preserved. To see this, let us consider an alternative definition of {P}L{Q} (the general case of unbounded nondeterminism would only differ by the use of higher order ordinals instead of natural numbers) :

. $\{P\}L^0\{Q\}$ is {P}skip{Q}

(this case is handled by (NI))

. $\{P\}L^n\{Q\}$ is {P} if B then C fi;L^{n-1} {Q} (or $\{P\}L^0\{Q\}$) for $n>0$

(this case is handled by (OI),(SC),(MI) (and (OR))

. $\{P\}L\{Q\}$ is $\exists n \geq 0.[\{P\}L^n\{Q\}]$

(this case is handled by (RR), because n can be incorporated in P and related to the initial values of the program variables).

Comparing with Dijkstra[76] we observe that $\{P\}\ L^n\ \{Q \wedge Dom[\![B]\!] \wedge \neg B\}$ holds if and only if $P \Rightarrow H_n(Q)$ where $H_n(Q)$ is the weakest precondition such that the while B do C od command will terminate after (at most) n iterations,

leaving a final state satisfying the post-condition Q. Hence the meaning of $\{P\}L\{Q\}$ for loop body labels L is consistent with the meaning $P \Rightarrow \underline{wp}(C,Q)$ of $\{P\}C\{Q\}$ for commands C. Moreover we have $\underline{wp}(\underline{\text{while}}\ B\ \underline{\text{do}}\ C\ \underline{\text{od}},Q) = \exists n \geq 0.H_n(Q)$ so that $\{P\}$ $\underline{\text{while}}$ B $\underline{\text{do}}$ C $\underline{\text{od}}$ $\{Q\}$ holds if and only if we have $\{P\}$ L $\{Q \wedge \underline{Dom}[\![B]\!] \wedge \neg B\}$. This case is handled by axiom (EW) and rule (HS) (which can be merged into a single rule, but this is less convenient for proof outlines). The last rule to be justified is (SI). It is in general very hard to exactly determine the $H_n(Q)$. Hence proof methods should provide tools to handle approximations from below (i.e. P_n's such that $P_n \Rightarrow H_n(Q)$). With Burstall's method the P_n's can be chosen freely and the proof that $P_n \Rightarrow H_n(Q)$ is by induction on n (up to an order isomorphism using (SI)). For example one can choose $P_n = H_n(Q)$ as proposed by Basu & Yeh[75] (on very simple examples). Also one is not tied to any particular form of the recursive definition of the $H_n(Q)$ (and this is very useful for examples such as Ackermann's function). With Floyd's method the P_n's should be of the form $P(x,X) \wedge f(x,X) \leq n$ where the loop invariant P and variant function f depend on (vectors of) logical variables x and program variables X and $(x',X') \prec (x,X)$ in (IC) if and only if $f(x',X') < f(x,X)$, (see Dijkstra[76],p.42). Hence Burstall's method offers a wider range of possible proofs (as shown by example 6.4). However these proofs can always be rephrased using Floyd's method (see Cousot&Cousot[83]).

The last choice to be discussed about the design of the logic is that the rules of inference (IC) and (SI) do not formally state that $\prec$ is a well-founded relation. We think this corresponds to the usual practice in termination proofs of relying upon (order) relations, the well-foundedness of which is taken for granted. Consequently $H,W \vdash \phi$ means that ϕ is true for interpretations I satisfying all hypotheses in H and such that the meaning $\prec_I$ of all $\prec \in W$ is a well-founded relation. An alternative consists in choosing a given range R and a given relation $\prec$ on R to be considered in (IC) and (SI). (For instance $\prec$ is the natural ordering on the set R of non-negative integers in Dijkstra[76], Harel[79]]). However there may be interpretations for which $\prec_I$ is not well-founded on R_I (e.g. non standard models of arithmetic in Harel[79]). Hence again the soundness of $H,W \vdash \phi$ is relative to the hypothesis that $\prec_I$ is well-founded on R_I or (up to an order-isomorphism) that $(R_I,\prec_I)$ is $(\alpha,<)$ for some ordinal α ($\alpha=\omega$ in Harel[79]). Another alternative consists in adding to (IC) and (SI) a premise which expresses the well-foundedness of $\prec$. The difficulty is now

that well-foundedness cannot be defined by a sentence $\underline{Wf}(\prec)$ of ordinary first order logic not even of $L_{\infty\omega}$. We can resort to $L_{\omega_1\omega_1}$ writing

$$\underline{Wf}(\prec) = \neg(\exists x_0\, x_1\, x_2 \ldots .\bigwedge_{n<\omega} (x_{n+1} \prec x_n))$$

but then we have to consider an infinitary logic and countable proofs, an alternative that would not be in the spirit of Hoare-style logics.

8. REFERENCES

APT,K.R.[81], *Ten years of Hoare's logic, a survey, part I*, TOPLAS 3, 4(1981), 431-483.

APT,K.R. & DELPORTE,C.[83], *An axiomatization of the intermittent assertions method*, Research report 82-70, LITP, Paris, France, (Jan. 1983), 21p.

BASU,S.K. & YEH,R.T.[75], *Strong verification of programs*, IEEE Trans. on Sotware Engineering, SE-1, 3 (Sept. 1975), 339-345.

BURSTALL,R.M.[74], *Program proving as hand simulation with a little induction*, Information Processing 74, North-Holland Pub. Co., (1974), 308-312.

COUSOT,P.[83], *On the soundness and completeness of a Hoare-style axiomatization of Burstall's intermittent assertions method*, Research report LRIM-83-05, U. of Metz, France, (Sept. 1983).

COUSOT,P. & COUSOT,R.[83], *SOMETIME = ALWAYS + RECURSION ≡ ALWAYS , on the equivalence of the intermittent and invariant assertions methods for proving inevitability properties of programs*, Research report LRIM-83-03, U. of Metz, (July 1983).

DIJKSTRA,E.W.[76],*A discipline of programming*, Prentice-Hall Inc., Englewood Cliffs, N.J., U.S.A., (1976), 217p.

DIJKSTRA, E.W.[77], *A sequel to EWD592*, EWD600, (Jan. 1977).

FLOYD,R.[67], *Assigning meaning to programs*, in Schwartz (ed.), Proc. Symp. in Applied Math., 19 AMS, Providence, R.I., (1967), 19-32.

GRIES,D.[79], *Is SOMETIME ever better than ALWAYS ?*, TOPLAS, 1, 2(1979).

HAREL,D.[79], *Proving the correctness of regular deterministic programs: a unifying survey using Dynamic Logic*, Research report RC7557, IBM T.J. Watson research center, Yorktown Heights, N.Y., (Mar. 1979), 28p.

HOARE,C.A.R.[69], *An axiomatic basis for computer programming*, CACM 12, 10 (1969), 576-580, 583.

MANNA,Z. & PNUELI,A.[74], *Axiomatic approach to total correctness of programs*, Acta Informatica, 3(1974), 253-263.

MANNA,Z. & WALDINGER,R.J.[78], *Is SOMETIME sometimes better than ALWAYS ?, Intermittent assertions in proving program correctness*, CACM 21, 2(Feb. 1978), 159-172.