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