Theoretical Computer Science 12 (1980) 61–81
© North-Holland Publishing Company

PROVING THE CORRECTNESS OF REGULAR DETERMINISTIC PROGRAMS: A UNIFYING SURVEY USING DYNAMIC LOGIC

David HAREL*
IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.

Communicated by Z. Manna
Received June 1978
Revised February 1979

Abstract. The simple set WL of deterministic *while* programs is defined and a number of known methods for proving the correctness of these programs are surveyed. Emphasis is placed on the tradeoff existing between data-directed and syntax-directed methods, and on providing, especially for the latter, a uniform description enabling comparison and assessment. Among the works considered are the Floyd/Hoare invariant assertion method for partial correctness, Floyd's well-founded sets method for termination, Dijkstra's notion of weakest precondition, the Burstall/Manna and Waldinger intermittent assertion method and more. Also, a brief comparison is carried out between three logics of programs: dynamic logic, algorithmic logic and programming logic.

1. Introduction

In this paper we provide a uniform description of some of the central works in proving the correctness of programs. The task becomes manageable by adopting an ultra-simple programming language, namely, that of deterministic *while* programs. In this way, problems appearing in the presence of recursive, nondeterministic or concurrent programs are avoided (cf. [5, 15, 22, 24, 44]).

A central, but somewhat hidden issue in the literature on program proving is the dichotomy of data-directed versus syntax-directed methods of proof. In the majority of cases, a proof of correctness which employs some sort of reasoning (say induction) directly on the data manipulated by the program is quite natural and in some cases reasonably easy to come by. The generic name used for many of these methods is *structural* induction. There are, however, many advantages in tying up the reasoning to the syntax of the program, even though the proofs are sometimes quite unnatural

* Part of this work was done while the author was with the Laboratory for Computer Science at the Massachusetts Institute of Technology, and was supported by NSF grants MCS76-18461 and MCS77-19754. Author's address from August 1980: Department of Applied Mathematics, The Weizmann Institute, Rehovot, Israel.

and difficult to find. The reasons for preferring the latter are not unlike those given by the proponents of structured programming; a small number of constructs to deal with, rigor and preciseness in description, discipline in style, program and proof developed simultaneously etc. To these we might add that syntax-directed methods lend themselves in a straightforward manner to investigations in meta-theory, such as soundness and completeness, comparative power of expression etc. *Computational* induction is sometimes used as a name for these methods, since the structure of the computation is invariably tied up with the structure of the program.

As it turns out, it is sometimes possible to describe a syntax-directed method as a specialized version of a data-directed one obtained by some restrictions, including the adoption of a simple, structured programming language. Consequently, most of the methods we will be able to talk about in a reasonably precise manner will be either syntax-directed or syntax-directed versions of data-directed methods. However, the distinction between methods of these two kinds is often quite vague.

Considering the diversity in language, notation and rigor which one finds in the relevant literature we are forced into adopting, besides a simple programming language, an equally simple language for describing the types of correctness considered and the proof methods themselves. We have chosen to use standard concepts from mathematical logic such as formula, axiom and inference rule (see e.g. [40]), augmented with concepts from the recently proposed logic of programs, dynamic logic [47, 22]. We hope that the ability to compare otherwise incomparable methods by translating them into a uniform language will partly compensate for the fact that an author's motivation and the naturalness and appeal of his particular presentation are occasionally lost when his method is quite violently reduced to a mere rule of inference. This is particularly true in the data-directed methods, and to the authors to whom we do such violence, we apologize in advance.

Our main thesis is that, for the simple programs we consider, the syntax-directed methods appearing in the literature fall into two main categories: those which use *invariance* to prove the partial correctness of programs and those which use *convergence* to prove their total correctness.

We certainly do not mention all published methods, nor do we exhaust all the ideas in those we do mention; the goal is to illustrate the way in which a formal framework for reasoning about programs can be put to good use in order to compare and evaluate proof methods originally expressed in a variety of other frameworks. The paper is aimed at the reader with some knowledge, both of mathematical logic and of program proving techniques, and is not on a description-by-example basis. Examples of proofs of programs using the methods we mention can be found e.g. in [16, 22, 30, 37, 39].

Section 2 contains preliminary notions and results, and Sections 3 and 4 treat partial and total correctness. Section 5, although short, is somewhat broader in character, comparing three *logics* in which many properties of programs, besides correctness, can be expressed and proved.

2. Preliminaries

In this section, we define the syntax and semantics of the programming language WL and the reasoning language RL, and state some properties of them which will be referred to in the sequel.

We are given a set of *function symbols* and a set of *predicate symbols*, each symbol with a fixed nonnegative integer *arity*. We assume the inclusion of the special binary predicate symbol '$=$' (equality) in the latter set. Predicate symbols are denoted by $p, q, \ldots$ and k-ary function symbols for $k > 0$ by $f, g, \ldots$. Zero-ary function symbols are denoted by $z, x, y, \ldots$ and are called *variables*. A *term* is some k-ary function symbol followed by a k-tuple of terms, where we restrict ourselves to terms resulting from applying this formation rule finitely many times only. For a variable x, we abbreviate $x(\ )$ to x, thus $f(g(x), y)$ is a term provided f and g are binary and unary respectively. An *atomic formula* is a k-ary predicate symbol followed by a k-tuple of terms.

The set L of first-order formulae is defined inductively in the standard way: an atomic formula is in L, and for any variable x and formulae P and Q of L, $\neg P$, $\exists xP$ and $(P \vee Q)$ are in L. The set of programs WL is defined as follows:

(1) for any variable x and term e, $x \leftarrow e$ is in WL,

(2) for any first-order formula S of L and for any α and β in WL, the following are programs in WL:

$$(\alpha\, ; \beta), \qquad \textit{if } S \textit{ then } \alpha \textit{ else } \beta, \qquad \textit{while } S \textit{ do } \alpha.$$

When there is no chance of confusion, we will abbreviate the last two constructs simply to *if* and *while* respectively. We use $\forall x$, $\wedge$, $\supset$ and $\equiv$ as abbreviations in the standard way. The construct $x \leftarrow e$ in (1) is called a (simple) *assignment*. The program constructs in (2) can be expressed as regular expressions over assignments and tests (see, e.g., [5, 47]), hence the adjective 'regular' in the title of this paper.

The semantics of a program in WL is based on the concept of a state. A *state* I consists of a nonempty domain D and a mapping from the sets of function and predicate symbols to the sets of functions and predicates over D, such that to a k-ary function symbol f (resp. predicate symbol p) there corresponds a total k-ary function (resp. predicate) over D denoted by f_I (resp. p_I). In particular, to a variable there corresponds an element of the domain and to a 0-ary predicate symbol (propositional letter) a truth value (*true* or *false*). The standard equality predicate over D should always correspond to the equality symbol $(=)$.

We are interested in special sets of states, namely, the (simple) universes. A *universe* U is a set of states with a common domain, in which all function and predicate symbols have a fixed value, except possibly for a designated set of variables. These are to be *uninterpreted* in U. (See [22] for detailed definitions.) The variables which are fixed are called constants.

The value of a term $e = f(e1, \ldots, ek)$ in a state I is defined inductively by

$$e_I = f_I(e1_I, \ldots, ek_I)$$

and the following standard clauses are adopted for defining the truth of first-order formulae::

$$
\begin{array}{lll}
I \models p(e1, \ldots, ek) & \text{iff} & p_I(e1_I, \ldots, ek_I) \text{ is true,} \\
I \models \neg P & \text{iff} & \text{it is not the case that } I \models P, \\
I \models \exists x P & \text{iff} & J \models P \text{ for some state } J \text{ differing from } I \\
 & & \text{at most in the value of } x, \\
I \models (P \vee Q). & \text{iff} & \text{either } I \models P \text{ or } I \models Q.
\end{array}
$$

$I \models P$ is read 'P is true in I', and we use $\models_U P$ to abbreviate '$I \models P$ for all $I \in U$'. Given a universe U, the meaning of a program $\alpha \in$ WL is the partial function $M(\alpha): U \to U$, defined inductively as follows: (Recall that *if* and *while* abbreviate *if S then α else β* and *while S do α* respectively.)

$M(x \leftarrow e)(I) = J$, where J is the state which differs from I at most in the value of x, and $x_J = e_I$

$$M(\alpha; \beta)(I) = M(\beta)(M(\alpha)(I)),$$

$$M(if)(I) = \begin{cases} M(\alpha)(I), & \text{if } I \models S, \\ M(\beta)(I), & \text{if } I \models \neg S. \end{cases}$$

Let $M^0(\alpha)(I)$ stand for I and $M^{i+1}(\alpha)(I)$ for $M(\alpha)(M^i(\alpha)(I))$. Then,

$$M(while)(I) = \begin{cases} M^k(\alpha)(I), & \text{if } k \geq 0 \text{ is such that } M^k(\alpha)(I) \text{ is defined, } S \text{ is false in} \\ & M^k(\alpha)(I)\text{, but true in } M^j(\alpha)(I) \text{ for any } j < k, \\ \text{Undefined}, & \text{if no such } k \text{ exists.} \end{cases}$$

We shall say that a program α, *when started in state I, terminates in state J* if and only if $M(\alpha)(I)$ is defined and is equal to J.

Our reasoning language RL is now defined as follows:

(1) $L \subseteq$ RL,

(2) for any formula $P \in L$, program $\alpha \in$ WL and formulae E and F in RL, the following are formulae of RL:

$\neg E, \quad E \vee F, \quad \langle \alpha \rangle P.$

The last construct is read 'diamond-α P'. The truth of a formulae of RL in a state I is defined standardly, with the addition of the clause

$$I \models \langle \alpha \rangle P \quad \text{iff} \quad M(\alpha)(I) \text{ is defined and } (M(\alpha)(I)) \models P.$$

We abbreviate $\neg\langle \alpha \rangle \neg P$ to $[\alpha]P$ ('box-α P'). Thus, one can see that given a universe U, $\models_U (R \supset [\alpha]Q)$ asserts that under the assumption R, if α terminates, then it terminates in a state satisfying Q, and $\models_U (R \supset \langle \alpha \rangle Q)$ asserts that under the assumption R, α indeed terminates and does so in a state satisfying Q. In this paper, we will be mainly interested in formulae of RL of the forms $(R \supset [\alpha]Q)$ and

$(R \supset \langle\alpha\rangle Q)$ for first-order R and Q, but have defined RL in order to provide a formal framework for stating axioms and rules of inference.

Definition 1. Given a universe U, a program $\alpha \in$ WL is said to be *partially* (respectively *totally*) *correct* with respect to first-order formulae R and Q if we have $\models_U (R \supset [\alpha]Q)$ (respectively $\models_U (R \supset \langle\alpha\rangle Q)$). In both cases, R and Q are called the *precondition* and *postcondition* of α respectively. We say that α *terminates* under condition R if α is totally correct with respect to R and *true*.

Manna [36] makes essentially first mention of the fact that total correctness is 'dual' to partial correctness (i.e. that by $[\alpha]P$ we mean $\neg\langle\alpha\rangle\neg P$) by expressing, in [36, Theorem 2], the formula $R \supset \langle\alpha\rangle Q$ as $\neg(R \wedge [\alpha]\neg Q)$.

RL is a sublanguage of the deterministic dynamic logic (DDL) of [22]. The following lemma summarizes some properties of RL, and proofs of the various parts of it can be found in [47, 22]. In the following, for first-order Q, let Q^e_x stand for Q with all free occurrences of the variable x replaced by the term e where no free variable in e becomes bounded by the replacement (otherwise rename bounded variables appropriately).

Lemma 1. *For any* $\alpha, \beta \in$ WL *and first-order formulae P and Q, the following are true in all states*:

(a) $\langle\alpha\rangle P \supset [\alpha]P$,

(b) $\langle\alpha\rangle P \equiv ([\alpha]P \wedge \langle\alpha\rangle \mathit{true})$,

(c) $\langle\alpha\rangle(P \wedge Q) \equiv (\langle\alpha\rangle P \wedge \langle\alpha\rangle Q)$,

(d) $[\alpha](P \wedge Q) \equiv ([\alpha]P \wedge [\alpha]Q)$,

(e) $\langle x \leftarrow e\rangle Q \equiv Q^e_x$,

(f) $\langle\alpha ; \beta\rangle Q \equiv \langle\alpha\rangle P$, *where* $P \equiv \langle\beta\rangle Q$,

(g) $\langle \mathit{if}\rangle Q \equiv ((S \wedge \langle\alpha\rangle Q) \vee (\neg S \wedge \langle\beta\rangle Q))$.

Lemma 1(a) confirms that total correctness is stronger than partial correctness, and by Lemma 1(b) total correctness can be seen to be equivalent to partial correctness plus, so to speak, termination. Lemma 1(c) and 1(d) allow in effect splitting a proof of correctness into two parts by splitting the required postcondition.

Central to this paper are the syntax-directed proof methods. These involve the idea of proving the partial or total correctness of a complex program by proving similar statements about the immediate components of that program (e.g. α and β in *if S then* α *else* β), and then combining these subresults to obtain the required one using some kind of rule. The rules are to be, in our case, formally expressible in the language RL and when the program is simply an assignment, i.e., has no program components, there should be a rule for composing an 'initial' correctness assertion about it simply from formulae of the first-order language L. All the methods we consider in this paper use, for an assignment $x \leftarrow e$, a composition $\alpha ; \beta$ and a conditional *if*, Lemma 1 (e, f, g) for total correctness and, for partial correctness, the easily derived duals:

$$[x \leftarrow e]Q \equiv Q^e_x,$$

$$[\alpha ; \beta]Q \equiv [\alpha]P, \quad \text{where } P \equiv [\beta]Q,$$

$$[if]Q \equiv ((S \supset [\alpha]Q) \wedge (\neg S \supset [\beta]Q)).$$

The case for composition requires some clarification. Lemma 1(f) gives rise to a rule which states that, for any P, from $\models_U \langle\alpha\rangle P$ and $\models_U (P \supset \langle\beta\rangle Q)$ one can conclude $\models_U \langle\alpha ; \beta\rangle Q$. This can be written, having U in mind,

$$\frac{\langle\alpha\rangle P, \quad P \supset \langle\beta\rangle Q}{\langle\alpha ; \beta\rangle Q}.$$

Thus, the proof of a claim of the form $\langle\alpha ; \beta\rangle Q$ involves finding an intermediate formula P satisfying the premises of the rule.

The main issue which we shall address when describing various proof methods is the approaches adopted for tackling the problem of loops; in our case the *while* construct. Here, straightforward equivalences such as Lemma 1(e, f, g) which would reduce the problem to one involving only components of the *while* construct are not available. However, we have the following results, the significance of which will become apparent in the sequel.

Lemma 2: *Given a universe U, if* $\models_U ((P \wedge S) \supset [\alpha]P)$, *then* $\models_U (P \supset [while](P \wedge \neg S))$.

In other words, if, whenever P holds and the body of the loop is guaranteed to be executed (i.e. S holds too), we know that P will hold upon termination, then if the *while* loop as a whole is started in a state where P holds, then it (as well as $\neg S$) will be true upon termination.

Now let A be an *arithmetical universe* (cf. [22]). Intuitively, the domain of A includes the natural numbers, and variables n, m etc. range over these. Also, standard symbols, such as $+$, $\times$, 0, 1 etc., receive their standard interpretations.

Lemma 3. *Assume n does not appear in α. If* $\models_A (P(n+1) \supset (S \wedge \langle\alpha\rangle P(n)))$ *and* $\models_A (P(0) \supset \neg S)$, *then* $\models_A (P(n) \supset \langle while\rangle P(0))$.

Here the intuition is as follows: if, whenever some property is true of $n+1$, it is the case that the body of the loop will be executed (S holds) and will indeed terminate properly in a state in which that property is true of n (i.e. we are guaranteed a 'decrease' of sorts), and if furthermore, we are assured that when the property 'reaches' being true of 0 the body of the loop will no longer be executed ($\neg S$ will hold), we can conclude that whenever the *while* loop as a whole is started in a state in which the property holds of some arbitrary natural number, then it will indeed terminate in a state in which the property will be true of 0.

More information on the subject matter of this section can be found in the survey paper [3].

Lemmas 2 and 3 illustrate the fundamental ideas of *invariance* and *convergence*, and will be seen to be the essence of almost all the methods we consider in the following sections. These two concepts are shown in [22] to be based, rather straightforwardly, on the principle of mathematical induction.

3. Partial correctness

Naur [43] and Floyd [18] can be regarded as the first substantial contributions to the art of proving the correctness of programs, and in them, independently, the *invariant assertion* method for proving the partial correctness of a deterministic program is described. The invention of the method is attributed by Floyd to Gorn [21] and Perlis and appears implicitly in the early work of Goldstine and Von Neuman [20] and Turing [51]. According to this method, a proof of the partial correctness of a program given in flowgraph form, is carried out by attaching assertions to some points in the program (the set of points including at least one on every cycle of the flowgraph), and verifying local implication between pairs of them, assuming that the path connecting them is indeed taken. This establishes the truth of the postcondition, whenever the precondition is true and the program terminates. This method, usually referred to as 'Floyd's method', can be viewed as being data-directed since the particular points to which assertions are attached are left unspecified and depending upon how they are chosen, the proof can take the form of an induction over a natural part of the data manipulated by the program (cf. [31]). However, specialized versions of Floyd's method, obtained by restricting the programming language to be rigidly structured become syntax-directed, as the points chosen in each cycle (loop) are fixed. The first and main such special version was introduced by Hoare [27] who adopted the programming language WL and described Floyd's method as an axiom system, writing $R\{\alpha\}Q$ for $\models_U (R \supset [\alpha]Q)$. ("... the treatment given below is essentially due to Floyd but is applied to texts rather than to flowcharts..." [27].) In fact, the idea of syntax-directed proofs in itself, substantiated by designing rules of inference which allow for proving the correctness of a program by 'breaking it up' and thus reducing its complexity, is due to Hoare.

The *while* loop is dealt with in [27] by observing that Lemma 2 gives rise to the following rule of inference

$$\frac{(P \wedge S) \supset [\alpha]P}{P \supset [\textit{while}](P \wedge \neg S)} \tag{3.1}$$

from which, in turn, one can derive

$$\frac{R \supset P, \quad (P \wedge S) \supset [\alpha]P, \quad (P \wedge \neg S) \supset Q}{R \supset [\textit{while}]Q}. \tag{3.2}$$

According to rule (3.2), in order to prove the partial correctness of a *while* loop with respect to R and Q one must find a first-order formula P which is implied by R, which, together with $\neg S$, implies Q, and which remains *invariant* under execution of the body α of the *while* loop. Such a formula must be found for every loop in the program to be proved and these are the *invariant assertions*. (Naur [43] calls them *general snapshots*). Thus, the points in every loop are fixed to be those immediately preceding the body α.

Notice that in the process of proving the partial correctness of a program using the Floyd/Hoare method, new first-order formulae are constructed (e.g. $R \supset P$ of (3.2)). These are the *verification conditions* of [18]. A proof, then, consists of appropriately choosing a set of invariant assertions and with them, using Hoare's axiom system or the algorithm described by Floyd, *translating* the $R \supset [\alpha]Q$ formula into a set of first-order (i.e. program-free) verification conditions. These are to be checked, manually or otherwise (e.g. using a theorem prover), to be true in all states of a given universe U. Another specialized version of Floyd's method is obtained by considering the programming language of and/or subgoal trees in which the points in loops are similarly fixed, cf. [23].

Katz and Manna [31] and others describe heuristics for constructing invariant assertions, but it is known that there is no general algorithm for producing ones which are sufficient for proving partial correctness. Some issues related to Hoare's system are discussed in [29] and a detailed implementation-oriented version of it is presented. The precise sense in which Hoare's axiom system is a specialized version of Floyd's method is described in [10, 25, 7].

Cook [14] introduces the important concept of the *relative completeness* of Hoare-like axiom systems for partial correctness, and proves that for some specific universes invariant assertions always exist, i.e. a proof as described above can always be carried out.

The work of Manna [36] serves to formalize the notion of $P \supset [\alpha]Q$ (and also $P \supset \langle\alpha\rangle Q$, see Section 4.2.3) in terms of satisfiability of logical formulae. Manna constructs, for a program α and first-order R and Q, a formula $W_\alpha[R, Q]$ which is basically the conjunction of Floyd's verification conditions where uninterpreted predicate symbols replace the invariant assertions. The result in [36] is that $R \supset [\alpha]Q$ is true (in all states of a universe U) iff $W_\alpha[R, Q]$ is satisfiable (in U); put another way, a proof using Floyd's method exists iff one can find suitable invariant assertions.

A method for proving partial correctness which is similar, and in a way dual to Floyd's is the *subgoal induction* method of [36, 42]. We refer the reader to [42, 39, 22] for more information on the analogy.

All the aforementioned methods are based on the concept of invariance, which itself is an offspring of the method of computational induction for a more general class of programs (cf. [37, 45]). In fact, natural extensions of WL in which programs can be nondeterministic, recursive or concurrent, cf. [5, 22, 23, 26, 44], all give rise to similar invariance-based methods of proof.

4. Total correctness

Here too, the basic method for proving assertions of the form $\models_U(P \supset \langle\alpha\rangle Q)$ was introduced by Floyd [18], and was described for flowgraphs. In [18], however, only termination, i.e. taking Q to be *true*, was considered. The method, termed the *well-founded sets* method, calls for attaching an expression over the program variables to points in the flowgraph (similarly also a point in every cycle) and showing that, whenever, these points are reached, the attached expressions will take values in some well-founded set W. Furthermore, one must show that whenever control moves from one such point to another, the value of the second expression is smaller in W than the value of the first. Thus, since the set is well-founded, this process cannot go on forever and the program must eventually terminate.

As in Floyd's method for partial correctness, this method can be viewed as being data directed. However, here describing a specialized, syntax-directed version seems to require that the well-founded set used, and the way in which decreasing values are obtained be made explicit.

In Section 4.1 some suggestions for solving this problem are described, while Section 4.2 describes some data-directed variants of the well-founded sets method.

4.1. *Syntax-directed methods*

One way of providing a syntax-directed version of Floyd's well-founded sets method [18] tailored to the language WL, is given by Lemma 3, from which the following rule, interpreted in arithmetical universes, can be derived:

$$\frac{P(n+1) \supset (S \wedge \langle\alpha\rangle P(n)), \quad P(0) \supset \neg S}{P(n) \supset \langle while\rangle P(0)}, \tag{4.1}$$

which in turn gives rise to

$$\frac{R \supset \exists n P(n), \quad P(n+1) \supset (S \wedge \langle\alpha\rangle P(n)), \quad P(0) \supset (Q \wedge \neg S)}{R \supset \langle while\rangle Q}. \tag{4.2}$$

(Note: in these and other rules we mention, the integer variables n, m, k etc. are assumed not to appear in α.)

In order to see why this rule represents Floyd's method, note that $P(n)$ can be taken to be $n = E$ for some expression E, and the well-founded set to be the set of natural numbers. (Here we have immediately generalized the method by having arbitrary Q as a post condition instead of simply *true*, thus providing for the ability to prove total correctness.) As mentioned in Section 2, assignments, composition and conditional statements are taken care of via Lemma 1 (e, f, g).

The notion of *arithmetical completeness* [22] has been used, analogously to the relative completeness of Cook [14], to show that in arithmetical universes rule (4.2) is sufficient for proving total correctness. In other words, an adequate formula $P(n)$ (termed *convergent* in [22], analogously to invariant for partial correctness) always exists.

Manna and Pnueli [38] were the first to suggest a syntax-directed analogue of the well-founded sets method. Also, they generalized it, as above, to prove total correctness. $\models_U (R \supset \langle\alpha\rangle Q)$ is written in [38] as $\langle R|\alpha|Q\rangle$. The notation of Manna and Pnueli is somewhat complicated by allowing Q to refer to the values of the variables as they were in the state before α was executed. This seems to be needed in order to compare the values of the "convergence function $u(\mathbf{x})$ mapping the program variable's domain X into [a well-founded set] W" [38], before and after α. However, here too we can, for simplicity, take W to be the set of natural numbers and 'freeze' the value of the decreasing function in a variable n, thus capturing the basic idea in their *while* rules as follows:

$$\frac{(P \wedge S \wedge u(\mathbf{x}) = n) \supset \langle\alpha\rangle(P \wedge (\neg S \vee u(\mathbf{x}) < n))}{P \supset \langle \mathit{while}\rangle(P \wedge \neg S)}, \tag{4.3}$$

where $\mathbf{x}$ is the vector of variables assigned to in α. This rule can be modified to

$$\frac{(P(n) \wedge n > k) \supset (S \wedge \langle\alpha\rangle(P(m) \wedge n > m \geqslant k)), \quad P(k) \supset \neg S}{(P(n) \wedge n \geqslant k) \supset \langle \mathit{while}\rangle P(k)}. \tag{4.4}$$

The differences between (4.1) and (4.4) are:

(a) some fixed integer k (not necessarily 0) is the 'lower bound' on P, and

(b) the decrease when α is executed need not necessarily be by 1.

This renders (4.4) more helpful in practice.

The work of Wang [52] is very similar to that of [38]. Wang's notation for $\models_U (R \supset \langle\alpha\rangle Q)$ is $[\mathit{in}\colon R\{\alpha\}\mathit{out}\colon Q]$ and he does not treat loops directly, but derives a treatment of them from an axiomatization of *goto* programs. Wang's derived rule for *while* (rule TG of [52]) is identical to (4.3) and his rule TF is identical to (4.1). The presentation of (4.1) is justified in [52] by: "It often happens that the induction used ... is based directly on the number of executions of the controlled statement α." This remark serves to illustrate the delicate borderline between data-directed and syntax-directed methods.

Sokolowski [50] describes an axiom system for proving total correctness, which is similar to [38], but in which the *while* rule is based on 'bounding loop counters', a method used by Knuth [32] and described in [30]. In [50], $\models_U (R \supset \langle\alpha\rangle Q)$ is written $\{R\}\alpha\{Q\}$ and the rule (rule K4' in [50]) can be written:

$$\frac{R \supset T(0), \quad (T(n) \wedge S) \supset \langle\alpha\rangle T(n+1), \quad (T(n) \wedge \neg S) \supset Q, \quad \forall \mathbf{x}\, \exists k\, (T(k) \supset \neg S)}{R \supset \langle \mathit{while}\rangle Q}, \tag{4.5}$$

where $\mathbf{x}$ is as in (4.3). The first three premises indicate that, starting from R being true, the body α of the *while* can be repeatedly executed keeping T true for increasing values; the fourth makes sure that for every value of the input variables there is a bound on the number of times α can be executed. This rule, however, is derived from rule (4.2) by defining $P(n)$ in state I to be $T(k\text{-}n)$ where k is the least integer in I such that $\forall\mathbf{x}(T(k) \supset \neg S)$, and to be *false* if such a k does not exist. Thus the loop counter method of [30] and [50] is derived from Floyd's [18] well-founded sets method applied to the integers.

Sokolowski also proves arithmetical completeness of his system ("it is assumed that . . . the language incorporates the calculus of the non-negative integers"), using essentially the *convergents* approach appearing later in [22].

We turn to Dijkstra [15, 16]. The notion of *weakest precondition* of a program α with respect to a predicate Q, written $\mathrm{wp}(\alpha, Q)$, was introduced by Dijkstra for dealing with nondeterministic programs, where $\mathrm{wp}(\alpha, Q)$ is to be true precisely in those states which have the property that when started in them α is guaranteed to terminate in a state satisfying Q. In [28, 22, 24] it has been shown that wp is, for nondeterministic programs, a nontrivial notion depending on methods of execution. However, for the special case of deterministic programs, in particular for WL, $\mathrm{wp}(\alpha, Q)$ turns out to be simply $\langle\alpha\rangle Q$. That is, $\langle\alpha\rangle Q$ is the weakest condition one can impose on the starting state such that α, when started in that state, is guaranteed to terminate in a state satisfying Q. Now, since it was Basu and Yeh [9] who applied wp to *while* programs, we summarize the parts of [9] which are relevant to this paper. $\models_U (P \supset \langle\alpha\rangle Q)$ is denoted in [9] by $P[\alpha]Q$, and $\langle\alpha\rangle Q$, as noted, by $\mathrm{wp}(\alpha, Q)$. Although not presenting explicit inference rules for *while*, Basu and Yeh describe $\langle \mathit{while}\rangle Q$ [9, formulae (6), (7) and (8)] as

$$\langle \mathit{while}\ S\ \mathit{do}\ \alpha\ \mathit{od}\rangle Q \equiv \exists n \langle (\mathit{if}\ S\ \mathit{then}\ \alpha\ \mathit{else}\ \mathit{loop})^n\rangle(\neg S \wedge Q),$$

where $(\beta)^n$, for some program β, abbreviates $\beta; \beta; \ldots; \beta$ with n appearances of β, and *loop* stands for some nonterminating program. Interestingly, they show that $P(n)$, in a rule such as (4.2), can be taken to be $\langle (\mathit{if}\ S\ \mathit{then}\ \alpha\ \mathit{else}\ \mathit{loop})^n\rangle(\neg S \wedge Q)$, and thus are essentially using convergents as in [22]. The examples in [9] involve computing this $P(n)$ as a function of n, and then (in order to prove $R \supset \langle \mathit{while}\rangle Q$) proving $R \supset \exists n\, P(n)$ "from properties of the integers". This again sketches the basics of the well-founded sets method of Floyd [18]; i.e. the program-free formulae resulting from carrying out the transformation from RL to the first-order language L are to be separately verified.

Here too, we see that these syntax-directed methods are all variations of the generic notion of convergence captured, for the well-founded set of the natural numbers, by Lemma 3.

4.2. *Data-directed methods*

This section is aimed at describing three different directions taken in the literature, each of which can be considered as providing a data-directed, but very general recipe for proving the total correctness of deterministic programs. Here it is more difficult to unify the presentation; in fact, the approach mentioned in Section 4.2.2 would require the development of so much additional technical machinery that a complete description would be well out of the scope of this paper. However, although somewhat artificially, we do attempt to describe one of the approaches, in Section 4.2.1, by presenting a syntax-directed analogue.

4.2.1. Intermittent assertions

A method for proving the total correctness of deterministic *goto* programs was suggested by Burstall [11], and was described in detail, using a variety of examples, in [39]. The method, termed the *intermittent assertion* method in [39], consists of proving $\models_U (R \supset \langle\alpha\rangle Q)$, written

"if sometime R at *start*, then sometime Q at *finish*",

by attaching assertions to points in α (as in [18]) and by proving that execution will *eventually* reach the points, satisfying the assertions. This is to be contrasted with Floyd's *whenever* a point is reached the assertion will be satisfied. For the non-*while* parts of α, the method coincides with Lemma 1(e, f, g).

In order to focus on the treatment of a loop, and so to see how *while*'s are dealt with, we introduce, for a program α, the notation α^* to stand for the nondeterministic program "execute α any number of times" (including possibly 0). The notation of RL is temporarily extended with $\langle\alpha^*\rangle Q$ thus being equivalent to $\exists n\langle\alpha^n\rangle Q$, or in other words "there is a way of executing α some number of times such that the execution will terminate in a state satisfying Q". Rule (4.1) for $\langle$*while*$\rangle$ can be derived from the following rule appearing in [47, 22]:

$$\frac{P(n+1)\supset\langle\alpha\rangle P(n)}{P(n)\supset\langle\alpha^*\rangle P(0)}. \tag{4.6}$$

Both the fact that this rule imposes the use of the natural numbers as the well-founded set, and the fact that one has to show decrease by 1, can be eliminated by rewriting it as

$$\frac{P(x)\supset\langle\alpha\rangle(P(y)\wedge y<x)}{x\geqslant z\supset(P(x)\supset\langle\alpha^*\rangle(P(y)\wedge z\geqslant y))}, \tag{4.7}$$

where x, y, z do not appear in α and are always to be elements of some set W with the well-founded order $<$. Rule (4.7) then, provides a general syntax-directed description of Floyd's method:

Attach a function to a cutpoint of each loop in α and show that its value is in some well-founded set W, and also show that the value decreases each time around the loop. (4.8)

Note now, that if $\langle\alpha\rangle$ in the premise of (4.7) is replaced by $\langle\alpha^*\rangle$ the rule remains sound. (If you can *eventually* get a decrease by doing α's, then you can eventually get as far down in W as you wish.) However, as it stands this modified version of (4.7) is no longer helpful since we have not reduced the complexity of the program involved; proving a $\langle\alpha^*\rangle$ claim requires proving a different $\langle\alpha^*\rangle$ claim. The point of using this rule though, is that one might be able to prove its premise by applying induction on some other quantity that the program manipulates. Hence, we can replace (4.7) by both the rule

$$\frac{P(x)\supset\langle\alpha^*\rangle(P(y)\wedge y<x)}{x\geqslant z\supset(P(x)\supset\langle\alpha^*\rangle(P(y)\wedge z\geqslant y))} \tag{4.9}$$

(differing from (4.7) only in the additional * in the premise), and an induction axiom scheme of the general form

$$(Q(y) \wedge \forall z((\forall x < z)Q(x) \supset Q(z))) \supset (\forall x \geqslant y)Q(x) \qquad (4.10)$$

with the appropriate restrictions on $x, y, z \in W$, and these for any relevant well-founded set W. Of course lacking here is a rigorous definition of $Q(y)$, for in (4.10) we are providing for the proof of an arbitrary formula Q of RL, one which might involve a program. However, for the sake of this discussion the above should suffice.

What we have done is basically to describe a method for proving $\langle\alpha^*\rangle$ formulae based on a generalized well-founded set method. While (4.7) serves to prove $\langle\alpha^*\rangle$ using (4.8), the combination of (4.9) and (4.10) requires that one

> Attach a function to a cutpoint of each loop in α, and show that its value is in some well-founded set W, and also show that the value eventually decreases after some times around the loop. (4.11)

Whereas proving the last part of (4.8) is easy since the loop has been 'cut open' ($\langle\alpha^*\rangle$ has been reduced to $\langle\alpha\rangle$ in (4.7)), proving the last part of (4.11) is of the same degree of difficulty as the whole of (4.11) ($\langle\alpha^*\rangle$ stays $\langle\alpha^*\rangle$ in (4.9)). To our help comes (4.10).

This is the essence of the intermittent assertion method of Burstall [11] and Manna and Waldinger [39]. The property they express is $\models_U (R \supset \langle\alpha\rangle Q)$, and the method they use to prove it is that of (4.9) and (4.10). The main idea is the transition from (4.8) to (4.11). We mention [10, Section 4], where a similar explanation of this method is apparent from their example.

Of course, what makes the intermittent assertion method data-directed is the fact that in [11, 39] it is *not* described as (4.9) and (4.10) but rather as (4.11), and for a general programming language in which *goto*'s replace *while*'s. The present section can, therefore, be viewed as an attempt to present almost a syntax-directed analogue. It is obvious to anyone who has seen the examples in [11, 39] that the power of the method lies in the fact that proofs are obtained in a natural manner. It is perhaps the intermittent assertion method which best exposes the naturalness of data-directed methods versus the rigidity of syntax-directed ones when it comes to manual proofs of programs. [12] also contains an exposition of this method, showing that induction on some quantity in the program is its basic feature (see also Section 4.2.3).

For the reader familiar with [11, 39], it is worth noting that the only examples appearing in these papers, the proofs of which make essential use of the subtle difference between (4.8) and (4.11), are two programs which are iterative versions of naturally described recursive ones, and which can be proved totally correct quite easily in their natural versions. These are the tips-of-the-tree program ([11, Section 5] and [39, Section 2.1]) and Ackerman's function ([39, Section 2.2]). It would be interesting to find a naturally constructed example for which (4.11) gives a natural proof whereas (4.8) does not.

4.2.2. *Temporal-like logics*

The papers of Schwarz [49], Ashcroft [1, 2], Kroeger [34, 35] and Pnueli [46] all describe temporal logics for reasoning about one given program. (A logic with the property that the one program it can discuss is implicit and does not appear in the formulae, is termed *endogenous* in [46]. In its original form, the intermittent assertion method [11, 39] is essentially endogenous too.) An implicit time scale, measuring the time passing as more of the program is executed, is assumed in all cases. With the aid of such a scale, one might state that P will eventually become true at some future time. In [49] these explicit time phenomena are somewhat less transparent (and for this reason we chose to include [49] in the works discussed in Section 4.2.3 too). The aforementioned notion of eventuality, for example, is expressed, in [2, 34, 46] as "*eventually* P", "*som* P" and "FP", respectively.

All four approaches regard their systems as kinds of modal logics, and they all claim to be formalizing the intermittent assertion method of [11, 39]. However, the precise relationships between these and other non-endogenous logics of programs (cf. Section 7) is still to be worked out. Restricting our attention to proof methods for deterministic programs, these works do not seem to go beyond the general notions of invariance and convergence, occasionally (as in [46]) in a form which captures the additional power of the intermittent assertion method, i.e. (4.11).

4.2.3. *Transliterating total correctness*

We now turn to three papers, Manna [36], Harel, Pnueli and Stavi [26] and Schwarz [49], which at first sight seem unconnected but which all describe the same process. The process is that of translating a formula of the form $R \supset \langle\alpha\rangle Q$ into a first-order formula V with free predicate symbols, having the property that $\models_U V$ iff $\models_U (R \supset \langle\alpha\rangle Q)$; i.e. α is totally correct w.r.t R and Q iff every assignment of predicates to the free predicate symbols of V satisfies V. How $\models_U V$ is to be proved (establishing the total correctness) is left unspecified, although all three papers give examples in which V is proved by induction on the integers. Of interest is the fact that V is essentially the same in all three cases, and is of the form $\neg(R \wedge Y)$, where Y is the conjunction of the verification conditions generated when Floyd's method is applied to $true \supset [\alpha]\neg Q$, with uninterpreted predicate symbols replacing the concrete invariant assertions. In this way, (by [36, Theorem 1]) $[\alpha]\neg Q$ holds iff there exists an assignment of predicates satisfying Y. Consequently, $\neg(R \wedge (\exists$ predicates$) Y)$ or really $(\forall$ predicates$)\neg(R \wedge Y)$, is the same as $\models_U \neg(R \wedge [\alpha]\neg Q)$ which in turn is equivalent to $\models_U (R \supset \langle\alpha\rangle Q)$.

Manna's $\tilde{W}_\alpha[R, Q]$ (without the $\exists$) is our $(R \wedge Y)$, and his Theorem 2 is essentially "α is [totally] correct w.r.t R and Q iff $(R \wedge Y)$ is unsatisfiable [in states of U]" (or equivalently $\models_U \neg(R \wedge Y)$ holds). Y is constructed by supplying a new predicate symbol for every label (essentially before every statement, if we translate Manna's programming language into ours). In [36] an example is presented, in which a program is proved totally correct by showing $R \wedge Y$ to be unsatisfiable "using the induction principle [for the integers]".

Although the relatively complete axiom system of Harel, Pnueli and Stavi [26] was designed for nondeterministic recursive programs, we shall ignore these features here and consider the parts relevant to this study. The system is an extension of Hoare's system and provides for proofs of *sequents* of the form $s\colon A_1, \ldots, A_n \models A$, where A and the A_i are either first-order or of the form $R \supset [\alpha]Q$. Here, s means that if $\models_U A_i$ holds for all $1 \leq i \leq n$, then so does $\models_U A$. Using this notation, $\models_U (R \supset \langle\alpha\rangle Q)$ can be written as the sequent

$$R(\mathbf{c}), \quad \mathbf{c} = \mathbf{x} \supset [\alpha]\neg Q \models \mathit{false}, \tag{4.12}$$

where $\mathbf{c}$ is a constant tuple, and $\mathbf{x}$ is the tuple of all variables in α. This is the technique of [26] for relating the world in which R is assumed to hold to the world in which $[\alpha]\neg Q$ does. Focusing on the *while* statement, a [*while*] formula appearing in this fashion (i.e. on the left of the $\models$ symbol) is derived essentially by the rule ($D15$ in [26], termed the rule of Inverse Iteration)

$$\frac{F, \quad R \supset P, \quad (P \wedge S) \supset [\alpha]P, \quad (P \wedge \neg S) \supset Q \models A}{F, \quad R \supset [\mathit{while}]Q \models A}, \tag{4.13}$$

where P is a new predicate symbol and F is any set consisting of first-order formulae or formulae of the form $R \supset [\alpha]Q$. Another way of stating this rule is

$$\frac{R \supset [\mathit{while}]Q}{\exists P(R \supset P, \quad (P \wedge S) \supset [\alpha]P, \quad (P \wedge \neg S) \neg Q)}. \tag{4.14}$$

A proof of (4.12) is carried out in the system of [26] by adding new predicate symbols for each ';' and *while*. The final program-free formulae will involve these new symbols, and careful analysis of this system shows that the conjunction of these formulae is precisely $\neg(R \wedge Y)$.

Schwarz [49] has described a proof system which he states "is based directly on Burstall" [11], but which seems to fit into the present framework too. The real connection with [11] seems to be that the same property is proved, namely, $\models_U (R \supset \langle\alpha\rangle Q)$. The method in [49] is based on an analysis of the computation sequence of a program. An 'event' is the fact that a label is reached with specific values for the variables. $\models_U (R \supset \langle\alpha\rangle Q)$ then, is written as in [11], $l\colon R \supset l'\colon Q$, where l and l' are the labels attached to both sides of α respectively. [49] too supplies a predicate symbol for each label of the program precisely as in [36], and a method for deriving theorems to be proved about them. The truth of these theorems implies the truth of the original claim. For $\langle$*while*$\rangle$, the method in [49] reduces to rule (4.14), and the $R \supset \langle\alpha\rangle Q$ assertion is to be proved "from the axioms generated by [the program] together with what we know about the data structures involved". In the examples in [49]"... normal mathematical induction [is used] but any other form of induction can...". This remark conveys the essence of this approach which is perhaps the 'most' data-directed of all. The assertion of total correctness is translated into a large formula to be proved by induction and one is left with the problem of finding an appropriate part of the manipulated data on which to carry out that induction.

Cartwright and McCarthy [12] have attempted to put the intermittent assertions method [11, 39] into the framework of the present section, by pointing out that for programs such as ours, taking α to be a function symbol results in $R \supset \langle\alpha\rangle Q$ being a first-order formula (see Section 4.3). Then, they claim, [11, 39] tells one to prove the validity of that very formula by induction. We feel that, as expressed in Section 4.2.1, the intermittent assertion method is somewhat more substantial.

4.3. Notations for total correctness

An unfortunate phenomenon observed when reading the papers described above is the variation of notation introduced in them. For partial correctness, Hoare's $R\{\alpha\}Q$ and its variant $\{R\}\alpha\{Q\}$ have been quite widely accepted, but for total correctness we can almost state

$$|notations| = |authors|.$$

We summarize this remark by tabulating authors with their notations of $\models_U (R \supset \langle\alpha\rangle Q)$ for deterministic programs. In each case, a fixed universe U is implicit, and whenever labels are required we take *start* and *finish* to be the entrance and exit labels respectively:

Reference	*Notations for* $\models_U (R \supset \langle\alpha\rangle Q)$
De Bakker [4, 5]	$R \subseteq \alpha \circ Q$,
Basu and Yeh [9]	$R[\alpha]Q$,
Burstall [11]	*Sometime*(At(*start*) and R), implies *Sometime*(At(*finish*) and Q)
Cartwright and McCarthy [12]	$(\forall \mathbf{x} \in D)(R(\mathbf{x}) \supset (\alpha(\mathbf{x}) \in D \wedge Q(\alpha(\mathbf{x}))))$,
Constable [13]	$R \supset \alpha; Q$,
Dijkstra [15, 16]	$R \supset \mathrm{wp}(\alpha, Q)$,
Harel, Pnueli and Stavi [26]	$R(\mathbf{c}), \mathbf{c} = \mathbf{x}\{\alpha\}\neg Q \models false$,
Kroeger [35]	$som(start \wedge R) \supset som(finish \wedge Q)$,
Manna [36]	$\neg \tilde{W}_\alpha[R, Q]$,
Manna and Pnueli [38]	$\langle R \vert \alpha \vert Q\rangle$,
Manna and Waldinger [39]	if sometime R at *start*, then sometime Q at *finish*,
Pnueli [46]	$[\pi = start \wedge R] \rightsquigarrow [\pi = finish \wedge Q]$,
Salwicki [48]	$R \supset \alpha Q$,
Schwarz [49]	$start{:}\, R \supset finish{:}\, Q$,
Sokolowski [50] and Manna [37]	$\{R\}\alpha\{Q\}$,
Wang [52]	$[start{:}\, R\{\alpha\}\, finish{:}\, Q]$.

In this paper, we do not attempt to justify our use of yet another notation.

5. Logics of programs

In this section, we briefly describe three logics which are all extensions of first-order predicate calculus oriented towards reasoning about programs The main concern will be to compare notation and power of expression, enabling free translation between them. We shall also indicate the approaches each takes towards supplying a proof theory. A common denominator is the fact that a formula can contain many programs and thus the power of expression of the logic is not limited to one or more kinds of correctness.

5.1. Dynamic Logic (Pratt et al.)

The ideas incorporated into *first-order dynamic logic* (DL) were suggested by Pratt [47] and the logic was further investigated in a variety of papers. Since our reasoning language in the present paper, RL, is derived from DL, we will only briefly indicate the general spirit here referring the reader to the literature for more details.

DL consists of predicate calculus augmented with an additional formation rule stating, inductively, that for a formula P and a program $\alpha \in PROG$, $\langle\alpha\rangle P$ is a formula, where $PROG$ is any predetermined class of programs. In most of the work on DL the set RG of regular (nondeterministic) programs over assignments and tests was adopted. This class is defined as the least set of programs including assignments and tests ($P?$, for first-order or quantifier-free P) and closed under the binary operations ';' and '$\cup$', and the unary operation '*'. The semantics of a program is given as a binary relation over a universe of states, with the meaning of $x \leftarrow e$ being given as $m(x \leftarrow e) = \{(I, M(x \leftarrow e)(I))\}$, that of $P?$ as $m(P?) = \{(I, I) \mid I \models P\}$, and the meanings of $\alpha ; \beta$, $\alpha \cup \beta$ and α^* being the composition, union and reflexive transitive closure of those of their components, respectively. The semantics of formulae are similar to those of RL in Section 2, with

$$I \models \langle\alpha\rangle P \quad \text{iff} \quad \exists J((I, J) \in m(\alpha) \wedge J \models P).$$

One can see that in DL, formulae such as $[\alpha](R \wedge \langle\beta\rangle[\gamma]Q) \supset \langle\alpha\rangle R$ are legal. Deterministic DL (DDL) is defined in [22], writing *if S then α else β* for the program $(S?; \alpha) \cup (\neg S?; \beta)$ and *while S do α* for $(S?; \alpha)^*; \neg S?$.

A proof theory is supplied in [22] by providing arithmetically complete axiomatizations in the spirit of rules (3.1), (4.1) and (4.6). A bibliography of relevant papers, including work on the *propositional* version of DL, PDL, can be found in [22].

5.2. Algorithmic logic (Salwicki et al.)

Algorithmic logic (AL) was introduced by Salwicki [48], whose work touched off many subsequent papers. A bibliography can be found in [8]. The general ideas incorporated into AL are based on early work of Engeler [17] and the structure of AL and DL (which was developed later) are remarkably similar. Most of the work on AL is concerned with a deterministic programming language and the proof methods considered are embodied in *infinitary* axiom systems.

Salwicki [48] uses essentially $[x/e]$ for $x \leftarrow e$ (terming assignments *substitutions*), $\circ[\alpha\beta]$ for $\alpha;\beta$, $\underline{\vee}\,[S\alpha\beta]$ for *if S then α else β*, and $*[S\alpha\beta]$ for (*while $\neg S$ do α*); β. Banachowski [6] later modifies the latter and writes $*[S\alpha]$ for *while S do α*. Thus, the programming language of AL is precisely WL. Formulae are defined in [48] similarly to DL, but with the following three formation rules, for a formula P and program α, replacing the $\langle\alpha\rangle P$ formation rule of DL:

$$\alpha P, \quad \bigcup \alpha P, \quad \text{and} \quad \bigcap \alpha P,$$

which mean, respectively,

$$\langle\alpha\rangle P, \quad \exists n\langle\alpha^n\rangle P \quad \text{and} \quad \forall n\langle\alpha^n\rangle P.$$

And we immediately adopt here Kreczmar's [33] extension, in which $\forall xP$ is added to the list of formation rules.

At first glance the $\bigcap\alpha P \equiv \forall n\langle\alpha^n\rangle P$ construct does not seem to be of use (notice, on the other hand, that $\exists n\langle\alpha^n\rangle P$ is the same as $\langle\alpha^*\rangle P$ of dynamic logic). The interesting thing is that although in AL $\bigcap\alpha P$ is definable in terms of other constructs its importance lies in its application to *nondeterministic* programs where $\bigcap$ can be used to succinctly express the absence or presence of infinite loops [22].

Mirkowska [41] is concerned with providing a complete axiomatization of AL using *infinitary* axiom systems. Loops are treated by transforming *while* into $\bigcup$ via the equivalence $*[S\alpha]P \equiv \bigcup\underline{\vee}[S\alpha[\ ]](\neg S \wedge P)$ and then using the axiom

$$\langle\alpha^*\rangle \equiv P \vee \langle\alpha^*\rangle\langle\alpha\rangle P \tag{5.1}$$

and the infinitary rule

$$\frac{[\alpha^n]P \quad \text{for all } n}{[\alpha^*]P}. \tag{5.2}$$

If one attempts to prove the infinite set of premises of (5.2) by induction on n, then in formulating the right inductive hypothesis he will in fact be coming very close to finding the invariant assertion of Floyd [18]. (See also the example in [34].) For applications of AL to proving correctness of programs see [6, 7].

5.3. *Programming logic* (*Constable*)

Programming Logic (PL) introduced by Constable [13] is very similar to the AL of Salwicki [48] in that $\langle\alpha\rangle P$ is taken as a primitive and the programs are deterministic. No provision, however, is made for top level iteration such as $\bigcup\alpha$ of [48]. We will refer here only to the first order logic of [13] ('polyadic') and not to either the propositional case ('monadic quantifier-free') or the version which allows quantification over states ('monadic').

The programming language of [13] is, again, precisely WL; assignments, $\alpha;\beta$, *if* (written $(S \to \alpha, \beta)$) and *while* (*written* $(S^*\alpha)$). Formulae are built up inductively from first-order formulae, and the constructs α and $\alpha; P$ for a program α and formula P, standing respectively for $\langle\alpha\rangle true$ and $\langle\alpha\rangle P$. $R \supset \langle\alpha\rangle Q$ then, is written as $R \supset \alpha\,;\,Q$, and $R \supset [\alpha]Q$as $R \supset (\neg\alpha\,;\,\neg Q)$.

Constable notes that "$\langle\alpha\rangle P$ behaves like wp(α, P)" and points out that in PL a substitution rule does not exist in general; that is, although it might be that $\alpha \equiv \beta$ (i.e. $\langle\alpha\rangle true \equiv \langle\beta\rangle true$), it is not in general true that $\alpha\,;\,P \equiv \beta\,;\,P$ (i.e. $\langle\alpha\rangle P \equiv \langle\beta\rangle P$). [13] does not provide an explicit proof method for formulae of AL.

6. Conclusion

We have attempted to bring together many known approaches to proving the correctness of simple *while* programs. In so doing, we were motivated by the desire to obtain a uniform description of them, to the extent that that end is possible. In the process two important issues emerged: the dual principles of invariance and convergence in treating, respectively, partial and total correctness, and the dichotomy of syntax- versus data-directed methods of proof.

Studies of this kind, especially in the area of programming logics and program verification, seem to be of some importance, as the notation, terminology and methods used are becoming more and more diverse and harder and harder to follow. Specifically, the topics of Sections 4.2.2 and 5 seem to deserve detailed attention: that is, it would be of considerable value to produce, for the topics of these sections, comparative studies, unifying the nature, from which the fundamental issues will emerge.

Acknowledgment

We would like to thank N. Dershowitz, Z. Manna, A. R. Meyer, V. R. Pratt and A. Shamir for many helpful conversations and comments.

The constructive remarks of A. Pnueli, especially his observations about syntax-directed versus data-directed methods, have contributed significantly to the presentation.

References

[1] E.A. Ashcroft, Intermittent assertions in lucid, Res. Report CS-76-47, Dept. of Comp. Science, University of Waterloo, Canada (1976).

[2] E.A. Ashcroft and W.W. Wadge, Lucid – a formal system for writing and proving programs, *SIAM J. Comput.* 5(3) (1976).

[3] K.R. Apt, Ten years of Hoare's logic, a survey. Manuscript, Erasmus University, The Netherlands (1979).

[4] J.W. de Bakker, Flow of control in the proof theory of structured programming, *Proc. 16th IEEE Symposium on Foundations of Computer Science* (1975).

[5] J.W. de Bakker and L.G.L.T. Meertens, On the completeness of the inductive assertion method, *J. Comput. System Sci.* **11** (1975) 323–357.

[6] L. Banachowski, Extended algorithmic logic and properties of programs, *Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys.* **23**(3) (1975).

[7] L. Banachowski, Modular properties of programs, *Bull. Acad. Polen. Sci. Sér. Sci. Math. Astronom. Phys.* **23**(3) (1975).

[8] L. Banachowski, A. Kreczmar, G. Mirkowska, H. Rasiowa and A. Salwicki, An introduction to algorithmic logic; metamathematical investigations in the theory of programs, in: Mazurkiewitcz and Pawlak, Ed., *Mathematical Foundations of Computer Science* (Banach Center Publications, Warsaw, 1977).

[9] S.K. Basu and R.T. Yeh, Strong verification of programs, *IEEE Trans. Software Eng.* **1** (3) (1975) 339–345.

[10] D.E. Britton, R.B. McLaughlin and R.J. Orgas, A note concerning intermittent assertions, *SIGACT News* (Summer 1977).

[11] R.M. Burstall, Program proving as hand simulation with a little induction, in: J.L. Rosenfeld, Ed., *Information Processing 74* (North-Holland, Amsterdam, 1974).

[12] R. Cartwright and J. McCarthy. First order programming logic, *Proc. 6th ACM Symposium on Principles of Programming Languages*, San Antonio, TX (1979).

[13] R.L. Constable, On the theory of programming logics, *Proc. 9th ACM (SIGACT) Symposium on Theory of Computing*, Boulder, CO (1977).

[14] S. Cook, Soundness and completeness of an axiom system for program verification, *SIAM J. Comput.* **7**(1) (1978).

[15] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, *Comm. ACM* **18**(8) (1975).

[16] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[17] E. Engeler, Algorithmic properties of structures, *Math. Systems Theory* **1**(1975) 183–195.

[18] R. Floyd, Assigning meaning to programs, in Schwartz, Ed., *Proc. Symposium in Applied Mathematics 19* (AMS, Providence, RI, 1967).

[19] S.L.Gerhart, Proof theory of partial correctness verification systems, *SIAM J. Comput.* **5**(3) (1976).

[20] H.H. Goldstine and J. Von Neumann, Planning and coding problems for an electronic computer department, in: A.H. Traub Ed., *Collected Works of John Von Neumann*, Vol. 5 (Pergamon Press, New York, 1963) 80–235.

[21] S. Gorn, Common programming language task, Part 1, Section 5, Final Report AD59UR1, U.S. Army Signal Corp., Moore School of Elec. Eng. (1959).

[22] D. Harel, *First-Order Dynamic Logic*, Lecture Notes in Computer Science **68** (Springer, Berlin, 1979).

[23] D. Harel, And/or programs: a new approach to structured programming, *ACM Trans. Programming Languages and Systems* **2** (1) (1980) 1–17.

[24] D. Harel, On the total correctness of nondeterministic programs, *Theoret. Comput. Sci.*, to appear.

[25] D. Harel, A. Pnueli and J. Stavi, Completeness issues for inductive assertions and Hoare's method, TR, Dept. of Math. Sciences, Tel-Aviv University, Israel (1976).

[26] D. Harel, A. Pnueli and J. Stavi, A complete axiomatic system for proving deductions about recursive programs, *Proc. 9th Annual ACM Symposium on Theory of Computing*, Boulder, CO. (1977) 249–260.

[27] C.A.R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* **12** (1969) 576–580, 583.

[28] C.A.R. Hoare, Some properties of predicate transformers, *J. ACM* **25**(3) (1978).

[29] S. Igarishi, R.L. London and D.C. Luckham, Automatic program verification 1: a logic basis and its implementation, *Acta Informat.* **4** (1975) 145–182.

[30] S.M. Katz and Z. Manna, A closer look at termination, *Acta Informat.* **5** (1975) 333–352.

[31] S.M. Katz and Z. Manna, Logical analysis of programs, *Comm. ACM* **19**(4) (1976) 188–206.

[32] D. Knuth, *The Art of Computer Programming*, Vol. 1 (Addison-Wesley, Reading, MA, 1968).

[33] A. Kreczmar, Effectivity problems in algorithmic logic, *Proc. 2nd Colloquium on Automata, Languages and Programming* (1974).

[34] F. Kroeger, Logical rules of natural reasoning about programs, in: *Automata, Languages and Programming 3*, (Edinburgh University Press, Edinburgh, 1976) 87–98.

[35] F. Kroeger, A uniform logical basis for the description, specification and verification of programs, *Proc. IFIP Working Conference on Formal Description of Programming Concepts*, St. Andrews, New Brunswick (1977).
[36] Z. Manna, The correctness of programs, *J. Comput. System Sci.* **3** (1969) 119–127.
[37] Z. Manna, *Mathematical Theory of Computation* (McGraw-Hill, New York, 1974).
[38] Z. Manna and A. Pnueli, Axiomatic approach to total correctness of programs, *Acta Informat.* **3** (1974) 253–263.
[39] Z. Manna and R. Waldinger, Is 'sometime' sometimes better than 'always'? Intermittent assertions in proving program correctness, *Comm ACM* **21**(2) (1978).
[40] E. Mendelson, *Introduction to Mathematical Logic* (Van Nostrand, New York, 1974).
[41] G. Mirkowska, On formalized systems of algorithmic logic, *Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys.* **19**(16) (1971).
[42] J.H. Morris Jr. and B. Wegbreit, Subgoal induction, *Comm ACM* **20**(4) (1977).
[43] P. Naur, Proof of algorithms by general snapshots, *BIT* **6**(1966) 310–316.
[44] S. Owicki and D. Gries, Verifying properties of parallel programs: an axiomatic approach, *Comm. ACM* **19**(5) (1976).
[45] D. Park, Fixpoint induction and proofs of program properties, in: *Machine Intelligence 5* (Edinburgh University Press, Edinburgh, 1969).
[46] A. Pnueli, The temporal logic of programs, *Proc. 18th IEEE Symposium on Foundations of Computer Science*, Providence, RI (1977).
[47] V.R. Pratt, Semantical considerations on Floyd–Hoare logic, *Proc. 17th IEEE Symposium on Foundations of Computer Science* (1976) 109–121.
[48] A. Salwicki, Formalized algorithmic languages, *Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys.* **18**(5) (1970).
[49] J. Schwartz, Event based reasoning – a system for proving correct termination of programs, *Proc. of Automata, Languages and Programming* (1976).
[50] S. Sokolowski, Axioms for total correctness, *Acta Informat.* **9**(1) (1977).
[51] A. Turing, Checking a large routine, *Report Conference on High Speed Automatic Calculating Machines*, Institute of Computer Science, McLennan Lab., University of Toronto, Toronto, Ontario, Canada (1950).
[52] A. Wang, an axiomatic basis for proving total correctness of goto programs, *BIT* **16** (1976) 88–102.