A THEORETICAL BASIS FOR STEPWISE REFINEMENT AND THE PROGRAMMING CALCULUS

Joseph M. MORRIS
Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland

Communicated by M. Rem
Received January 1986
Revised December 1986

Abstract. A uniform treatment of specifications, programs, and programming is presented. The treatment is based on adding a specification statement to a given procedural language and defining its semantics. The extended language is thus a specification language and programs are viewed as a subclass of specifications. A partial ordering on specifications/programs corresponding to 'more defined' is defined. In this partial ordering the program/specification hybrids that arise in the construction of a program by stepwise refinement form a monotonic sequence. We show how Dijkstra's calculus for the derivation of programs corresponds to constructing this monotonic sequence. Formalizing the calculus thus gives some insight into the intellectual activity it demands and allows us to hint at further developments.

1. Introduction

The construction of a program begins with a specification, proceeds through a series of hybrids that are part specification and part program, and ends with a program. The programmer tries to ensure that after each step in the process he has preserved the correctness of the program with respect to the given specification while moving ever closer to his goal of a fully realized program. Both the given specification and the final program can be regarded as formal objects: the specification can be expressed in predicate logic, and the program can be viewed as a function on predicates [5] all in such a way that, at least in principle, the programmer can verify that the program meets its specification. It is well established, however, that the verification of an arbitrary program is an arduous task and that the programmer is best served by a methodology that requires program and proof to develop in tandem [5, 7, 8]. Such a methodology, however, has been subject to this weakness: although specifications and programs are formal objects, the hybrids that arise in the construction process are partly informal. Formal verification remains apart from the construction process.

In this paper, we present a view of programming that better integrates the activities of programming and program verification. The essential idea is to add a specification statement to a given procedural language, defining its semantics in the same way as other statements. The extended language is thus a specification language and programs now appear as a subclass of specifications. We view programming as
constructing a sequence of specifications, each one better defined than, but preserving the meaning of, its predecessors; the final specification is a program in the given language. More formally, we will define a partial ordering on specifications so that the specifications arising in the construction of a program form a monotonic sequence. We formally define programming as the construction of these monotonic sequences.

The expected advantage of formalizing programming in this way is simply that we will program better, firstly because correctness concerns are more part of the methodology, and secondly because the formal framework in which we are working may allow us to deploy more readily formal techniques when it seems best to do so; we will give some hints of this. These ideals underlie Dijkstra's calculus of programming [5, 7, 8]; essentially we are presenting a more formal treatment of the calculus.

We presume a customer comes to us with a problem specified by a construct we shall call a 'prescription'; it consists of a pair of predicates $P, Q$ written "$P\parallel Q$". Prescription $P\parallel Q$ specifies a mechanism that when executed in a state satisfying $P$ terminates in a state satisfying $Q$. We now take a given programming language and add to it prescriptions; prescriptions are statements having the same formal status as other statements. We no longer have a programming language but a specification language, and a program is now viewed as a special kind of specification—one without prescriptions. Prescriptions will play an important role in program construction but have no place in the final solution.

The specification language has quite different properties from the underlying programming language for it is possible to specify unrealizable mechanisms. Nonetheless, we will see that the language retains enough nice properties to be pleasant to work with. In particular, we can define a partial ordering $\sqsubseteq$ on specifications such that $\sqsubseteq$ corresponds to 'correctness preserving'. This allows us to formalize the following methodology. We proceed from the initial prescription $P\parallel Q$ through a sequence of specifications $s_i$ such that

$$P\parallel Q \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \cdots \sqsubseteq s.$$  

Each specification after $P\parallel Q$ is obtained either by replacement of a prescription in the specification preceding it (and quite likely introducing other prescriptions), or—more surprisingly—by taking the limit of the specifications preceding it. All going well this process terminates and we can go back to the customer in full confidence that we are giving him what he asked for, a program $s$ satisfying $P\parallel Q \sqsubseteq s$.

The reader will doubtless recognize this as nothing but stepwise refinement [13] in formal dress. Here is a trivial example:

$$true \parallel x = 0 \text{ and } y = 1$$

$$\sqsubseteq (true \parallel x = 0); (x = 0 \parallel x = 0 \text{ and } y = 1)$$

$$\sqsubseteq x := 0; (x = 0 \parallel x = 0 \text{ and } y = 1)$$

$$\sqsubseteq x := 0; y := 1.$$  

Of course, we shall have to give formal rules for justifying each refinement.
2. Mathematical preliminaries

A relation on a set $C$ is a 'partial ordering' of $C$ if it is reflexive, antisymmetric, and transitive; it is a 'total ordering' if in addition each pair of elements in $C$ are comparable. We denote the structure consisting of set $C$ and a partial ordering $\sqsubseteq$ on $C$ by $(C, \sqsubseteq)$. However, we shall often be lazy in our notation and neglect the distinction between $C$ and $(C, \sqsubseteq)$ when the partial ordering is understood or when the context makes clear whether we are regarding $C$ as a set or as a set with a partial ordering.

A set $B \subseteq C$, with $C$ and hence $B$ partially ordered by $\sqsubseteq$, has an 'upper bound' $u \in C$ if $x \sqsubseteq u$ for all $x \in B$; $u$ is a 'least upper bound'—'lub' for short—if in addition $u \sqsubseteq v$ for every upper bound $v$ of $B$. The lub of set $B$ when it exists is denoted by $\biguplus B$. An element $\bot \in C$ is a 'bottom' or 'least element' of $C$ if $\bot \sqsubseteq x$ for all $x \in C$, and $\top \in C$ is a 'top' or 'greatest element' of $C$ if $x \sqsubseteq \top$ for all $x \in C$. An element $m \in C$ is a 'minimal element' of $C$ if $x \sqsubseteq m \Rightarrow x = m$ for all $x \in C$.

If sets $C$ and $D$ are partially ordered by $\sqsubseteq_C$ and $\sqsubseteq_D$, respectively, then the cartesian product $C \times D$ is partially ordered by $\sqsubseteq$, defined by

$$ (w, x) \sqsubseteq (y, z) \iff w \sqsubseteq_C y \text{ and } x \sqsubseteq_D z $$

for all $w, y \in C$, and $x, z \in D$. Definition (1) can be generalized to $n$-fold cartesian products in the obvious way. The product sets in this paper will always be partially ordered by (1) and we will not state this explicitly in each case.

Given sets $C$ and $D$ with $D$ partially ordered by $\sqsubseteq_D$, the set $C \to D$ of functions from $C$ to $D$ is partially ordered by $\sqsubseteq$, defined by

$$ f \sqsubseteq g = f(x) \sqsubseteq_D g(x) \text{ for all } x \in C. $$

In this paper, functions will be partially ordered by (2) only and we will not state this explicitly in each case. If, in addition $C$ is partially ordered by $\sqsubseteq_C$, then $f : C \to D$ is said to be 'monotonic' if

$$ x \sqsubseteq_C y \Rightarrow f(x) \sqsubseteq_D f(y) \text{ for all } x, y \in C. $$

We denote by $(C \to D)$ the set of monotonic functions from $C$ to $D$.

A 'complete lattice' is a partially ordered set in which each subset has a lub. It can be shown that every complete lattice has a bottom and a top. We now give some well-known properties of complete lattices, omitting proofs; the reader looking for more details should refer to the literature, e.g. [3, 12].

**Lemma 2.1.** Any finite totally ordered set is a complete lattice.

**Lemma 2.2.** If $C, D$ are complete lattices, then so is $C \times D$. Moreover, for $B \subseteq C \times D$,

$$ \biguplus B = \biguplus \{ x : (\exists y : (x, y) \in B) \}, \biguplus \{ y : (\exists x : (x, y) \in B) \}.$$
Lemma 2.2 can be generalized to n-fold cartesian products in the obvious way.

**Lemma 2.3.** If $C$ is a partially ordered set and $D$ is a complete lattice, then $(C \to D)$ is a complete lattice. Moreover, for $B \subseteq (C \to D)$,

$$(\bigsqcup B)(x) = \bigsqcup \{f(x) : f \in B\} \text{ for all } x \in C.$$ 

The reader may prefer just to scan the rest of this section on first reading; it is hardly needed until Section 4.2.

As is conventional we will use small Greek letters to denote ordinals; $\omega$ will denote the first infinite ordinal, i.e. the set of natural numbers. For any complete lattice $C$ and $f : (C \to C)$, we define

- $f^0 = \text{the identity function on } C$,
- $f^{\lambda + 1} = f \circ f^\lambda$ (functional composition) for successor ordinals $\lambda + 1$,
- $f^\lambda = \bigsqcup \{f^\gamma : \gamma < \lambda\}$ for limit ordinals $\lambda$.

**Lemma 2.4.** Given $f : (C \to C)$ for $C$ a complete lattice, and $c \in C$ satisfying $c \sqsubseteq f(c)$,

(i) $\beta \leq \gamma \Rightarrow f^\beta(c) \sqsubseteq f^\gamma(c)$ for all ordinals $\beta, \gamma$;

(ii) There exists a least ordinal $\alpha$ such that

$$\forall\gamma(\gamma \geq \alpha) : f^\gamma(c) = f^\alpha(c).$$

In Lemma 2.4(ii) $\alpha$ is called the 'closure ordinal' of $f$ in $C$. As we can safely replace the closure ordinal $\alpha$ in $f^\alpha(c)$ with any ordinal $\geq \alpha$, we can conveniently work with one 'super-closure' ordinal $\infty_f$ for each $f$; for $\infty_f$ take any ordinal containing all closure ordinals of $f$. For brevity, we will write simply $\infty$, letting context supply the implicit subscript.

Any $c$ satisfying $f(c) = c$ is called a 'fixed point' of $f$; if in addition $c \sqsubseteq d$ for every fixed point $d$ of $f$, then $c$ is a 'least fixed point' of $f$. The least fixed point of $f$ when it exists, is denoted by $\mu x. f(x)$.

**Lemma 2.5.** Given $f : (C \to C)$ for $C$ a complete lattice with bottom $\bot$, $f$ has a least fixed point satisfying $\mu x. f(x) = f^\infty(\bot)$.

Monotonicity also applies to sequences in the obvious way: a (possibly transfinite) sequence $(x_0, x_1, \ldots, x_n, \ldots)$ with elements drawn from a set partially ordered by $\sqsubseteq$ is said to be 'monotonic' if $\alpha \leq \beta \Rightarrow x_\alpha \sqsubseteq x_\beta$ for all indices $\alpha, \beta$.

In the rest of the paper we will employ only $\sqsubseteq$ to denote a partial ordering, letting context resolve any ambiguity that might otherwise arise.
3. Programs and specifications

3.1. Assertions

We partially order the set of booleans \{true, false\} by \(x \sqsubseteq y\) iff \(x \Rightarrow y\); by Lemma 2.1 we have a complete lattice which we denote by \(\text{Bool}\). We presume the programmer is asked to make a program that operates in a given ‘state-space’ determined in the usual way by a set of variables. For simplicity, we use integer variables only. We will use letters \(b, c, \ldots\) (possibly primed) as integer variables and \(e, e_1, e_2\) as integer expressions. We let \(\text{Sta}\) denote the set of states partially ordered by the equality relation. We describe restrictions on the allowable states in the usual way by boolean-valued functions on the state-space called ‘assertions’. A state is said to ‘satisfy’ an assertion if the assertion applied to the state yields ‘true’. Assertions are expressed as logical formulae. The assertion \(\text{true}\) is satisfied by all states, and the assertion \(\text{false}\) by none. We will use the letters \(P, Q, R, \ldots\) to stand for assertions.

As usual, we will specify the behaviour of a program with two assertions, the first assumed to be satisfied by the starting state of the program, and the second to be satisfied by the final state.

Assertions have a dual existence. On the one hand, they are syntactic objects and we shall want to perform syntactic-operations on them. In particular, we denote by \(P[b:e_1, c:e_2, \ldots]\) the assertion got by simultaneously replacing all occurrences of variables \(b, c, \ldots\) in \(P\) by expressions \(e_1, e_2, \ldots\), respectively.

Assertions also have a semantic existence as functions on the state-space: each assertion is true or false when its variables are replaced by their values. Strictly, we should distinguish between the two natures of assertions in our notation. However, it is convenient not to do so: it may not matter what view we are taking; we may be taking both views simultaneously; and in any case when we are viewing an assertion just syntactically or just semantically it will be clear from the context what view we are taking. Consequently, we can work with a simpler notation.

It is easy to prove that \(\text{Sta} \to \text{Bool}\) and \((\text{Sta} \to \text{Bool})\) are identical, and hence that assertions are embedded in a complete lattice; we denote this lattice by \(\text{Asn}\). For any assertion \(P\) let us denote by \([P]\) the universal quantification of \(P\) over the state-space, i.e. the proposition “\(P\) holds in all states of the state space”. (Bear in mind that \([\ldots]\)’s are universally quantified expressions and so we may want to employ the properties of universal quantifiers in manipulating them.) Although setting assertions in complete lattices may not be familiar, the following lemmas show that we have not strayed far from the usual treatment.

**Lemma 3.1.** For \(P, Q \in \text{Asn}\), \(P \equiv Q = [P \Rightarrow Q]\).

**Proof.** Exercise. \(\square\)
Lemma 3.2. For \( B = \{P_j: j \in J\} \) any set of assertions, \( \bigvee B \equiv \exists j (j \in J): P_j \).

Proof. (i) (upper bound). For any \( j \in J \) we have
\[
P_j \subset \exists j (j \in J): P_j \quad \text{by Lemma 3.1, predicate calculus}
\]
\[
\equiv \bigvee \{P_j: j \in J\} \subset \exists j (j \in J): P_j \quad \text{by definition of } \bigvee
\]
\[
\equiv \bigvee B \subset \exists j (j \in J): P_j
\]

(ii) (least upper bound). Let \( L \) be any upper bound of \( B \).

\[
L \text{ an upper bound of } B
\]
\[
\equiv \forall j: j \in J \Rightarrow (P_j \subset L) \quad \text{by Lemma 3.1}
\]
which, using predicate calculus
\[
\equiv [(\exists j: j \in J \text{ and } P_j) \Rightarrow L]
\]
\[
\equiv (\exists j (j \in J): P_j) \subset L \quad \text{by Lemma 3.1.} \quad \Box
\]

3.2. Specifications

We will adopt as our programming language the notation of ‘guarded commands’ [5]. Statements will be denoted by \( p, q, r, s, \ldots \) possibly with trailing digits or subscripts. Boolean expressions (‘guards’) will be denoted by \( g, g_1, g_2, \ldots \). The statements are \textbf{abort}, \textbf{skip}, \( b := e \) (assignment), \( p; q \) (sequential composition), \( \text{if } g_1 \rightarrow s_1 \bigcup g_2 \rightarrow s_2 \text{ fi} \) (if-statement, denoted by IF) and \( [\text{new } b; p] \) (block). In IF the syntactic form \( g \rightarrow s \) is called a ‘guarded command’. We also admit recursive procedure definitions such as \( f : F \) — or \( f : F[f] \) when we wish to make the recursion explicit—where \( F \) denotes a program to be invoked by the statement \( f \).

We give programs a semantic existence by regarding them as functions on \( \text{Asn} \). The result of applying \( p \) to assertion \( R \) is denoted by \( p(R) \) (which is equivalent to Dijkstra’s \( \text{wp}(p, R) \) [5]). The functions are defined so that \( [Q \Rightarrow p(R)] \) has the interpretation that \( p \) executed in any state satisfying \( Q \) will terminate in a state satisfying \( R \). As with assertions, programs have a significant syntactic and semantic existence and our notation does not explicitly distinguish between them. \( p(R) \) is called the ‘weakest precondition’ for \( p \) to establish \( R \). We define \( p(R) \) for each \( p \) as follows; definition (e) can be generalized in an obvious way:

(a) \( \textbf{abort}(R) \equiv \text{false} \),
(b) \text{skip}[R] = R,

(c) b:= e[R] = R[b: e],

(d) p;q[R] = p(q[R]),

(e) \text{IF}[R] = (g1 \text{ or } g2) \text{ and } (g1 \Rightarrow s1[R]) \text{ and } (g2 \Rightarrow s2[R]),

(f) [\text{new } b; p][R] = (\forall b: p[R[b:b']])[b':b] \\
\text{where } b' \text{ does not occur in } R \text{ or } p.

We postpone giving the semantics of recursion. Programs composed only of statements (a) to (f) will be called 'straight-line' programs.

For each pair of assertions \( P, Q \) we admit into our language the statement \( P \parallel Q \) which we shall call a 'prescription'; informally, it specifies a mechanism that whenever executed in a state satisfying \( P \), terminates in a state satisfying \( Q \), and when executed in a state not satisfying \( P \) behaves uninterestingly. More formally, we define its weakest precondition:

\[ P \parallel Q[R] = P \text{ and } [Q \Rightarrow R]. \quad (3) \]

Prescriptions obviously make 'programming' very easy—just write the prescription! However, it remains the task of the programmer to produce solutions containing no prescriptions—they are used only as a means to an end. Because we have now admitted prescriptions to our language we shall call its elements 'specifications', reserving the term 'program' to describe a specification containing no prescriptions. In the example in the Introduction, for example, each line is a specification but only the final line is a program.

In a moment we shall embed specifications in a complete lattice. Anticipating that, we include in the class of specifications the 'limit' specification \( \square W \) for every totally ordered set \( W \) of specifications.

We will denote by \( F[x, y, \ldots], G[x, y, \ldots], \ldots \) specifications possibly dependent on arbitrary specifications \( x, y, \ldots \). We will denote by \( F[x:p, y:q, \ldots] \)—or simply \( F[p, q, \ldots] \)—when the \( x, y, \ldots \) are understood—the specification resulting from simultaneously replacing \( x, y, \ldots \) in \( F \) by the specifications \( p, q, \ldots \), respectively.

We summarize our naming conventions:

\begin{itemize}
  \item \( b, c, d, \ldots \) integer variables,
  \item \( e, e1, e2, \ldots \) integer expressions,
  \item \( g, g1, g2, \ldots \) boolean expressions,
  \item \( F, G, \ldots \) specifications possibly containing specification variables,
  \item \( p, q, r, s, \ldots \) specifications (including programs, statements),
  \item \( P, Q, R, \ldots \) assertions.
\end{itemize}

We want to partially order specifications so that we can give a formal meaning to: "specification \( q \) is a refinement of specification \( p \)", i.e. \( q \) is in some sense better defined than \( p \) while maintaining its correctness. It turns out that the partial ordering we want is that of monotonic functions on \( \text{Asn} \).
Theorem 3.3. Every specification \( s \in (\text{Asn} \rightarrow \text{Asn}) \).

Proof. We must show that given \( [R \Rightarrow S], [s\{R\} \Rightarrow s\{S\}] \). The proof is by structural induction on the syntax of specifications. For the case of straight-line programs see [8]. For prescriptions we have

\[
P \parallel Q\{R\}
\]

\[= P \land [Q \Rightarrow R] \quad \text{by (3)}
\]

\[
\Rightarrow P \land [Q \Rightarrow S] \quad \text{by predicate calculus, [R \Rightarrow S]}
\]

\[= P \parallel Q\{S\} \quad \text{by (3)}.
\]

The case of limit specifications follows from Lemma 2.3. □

It follows that specifications are embedded in the complete lattice \( (\text{Asn} \rightarrow \text{Asn}) \) which we denote by \( \text{Spec} \). \( \text{Spec} \) has \text{abort} as bottom. We now show that this partial ordering of \( \text{Spec} \) is just what we want because it corresponds exactly to 'correctness preserving'.

Theorem 3.4. \( P \parallel Q \sqsubseteq s = P \sqsubseteq s\{Q\} = [P \Rightarrow s\{Q\}] \).

Proof. The second equivalence follows immediately from Lemma 3.1. We prove the first equivalence.

\((\Rightarrow)\):

\[
P \parallel Q \sqsubseteq s
\]

\[
\Rightarrow P \parallel Q\{Q\} \sqsubseteq s\{Q\} \quad \text{by definition of} \ \sqsubseteq \ \text{on Spec, Asn}
\]

\[= P \land [Q \Rightarrow Q] \sqsubseteq s\{Q\} \quad \text{by (3)}
\]

\[= P \sqsubseteq s\{Q\} \quad \text{by predicate calculus}.
\]

\((\Leftarrow)\):

\[
P \sqsubseteq s\{Q\}
\]

\[= [P \Rightarrow s\{Q\}] \quad \text{by Lemma 3.1}
\]

\[
\Rightarrow [(P \land [Q \Rightarrow S]) \Rightarrow (s\{Q\} \land [Q \Rightarrow S])] \quad \text{for any} \ S
\]

\[\Rightarrow [P \parallel Q\{S\} \Rightarrow s\{S\}] \quad \text{for any} \ S \quad \text{by (3), Theorem 3.3, predicate calculus}
\]

\[= P \parallel Q \sqsubseteq s\{S\} \quad \text{for any} \ S \quad \text{by Lemma 3.1}
\]

\[= P \parallel Q \sqsubseteq s \quad \text{by definition of} \ \sqsubseteq \ \text{on Spec}.
\]

Theorem 3.4 is important. It allows us to conclude that if we proceed from a prescription \( P \parallel Q \) through a series of refinements satisfying

\[
P \parallel Q \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \cdots \sqsubseteq s
\]
ending with program s, then s meets its specification, i.e. \([P \Rightarrow s\{Q\}]\). When specifications \(p, q\) satisfy \(p \sqsubseteq q\) we say that \(q\) is a 'refinement' of \(p\); if in addition \(q\) is a program we say that \(q\) is an 'implementation' of \(p\). We also write \(p \sqsubseteq q\) as \(q \sqsupseteq p\).

### 3.3. Properties of specifications

It will be instructive to consider the four simplest prescriptions:

- **chance**: \(true\)
- **miracle**: \(true\)
- **abort**: \(false\)
- **abort**: \(false\)

The last two are both named **abort** because, as is easily verified, they satisfy \(\text{abort}(R) = false\) for all \(R\). Prescription **miracle** satisfies \(\text{miracle}(R) = true\) for all \(R\)—no matter what the circumstances it gives us whatever we desire! **miracle** is the top of \(\text{Spec}\). Prescription **chance** satisfies \(\text{chance}(R) = [R]\) for all \(R\). **chance** behaves like a roulette wheel: it is guaranteed to terminate in one of the (usually infinite number of) states of the state-space, but no particular state or subset of states can be guaranteed. Observe

\[
\text{abort} \subseteq \text{chance} \subseteq \text{skip} \subseteq \text{miracle}.
\]

We leave it as an exercise to show that there is no prescription equivalent to **skip**.

We know from [5] that programs enjoy certain properties, the following ones being fundamental (in Properties 1 to 3 \(p\) denotes a program).

**Properties.** For all programs \(p\),

1. \(p\{false\} = false\);
2. \(p\{P \land Q\} = p\{P\} \land p\{Q\}\);
3. \(p\{\exists i(i \in \omega): P_i\} = \exists i(i \in \omega): p\{P_i\}\) for all monotonic sequences \(\langle P_0, P_1, \ldots \rangle\).

Specifications as we have defined them, however, do not enjoy Properties 1 ('law of the excluded miracle') or 3 ('continuity in postconditions'). **miracle** clearly violates Property 1. **chance** in an infinite state-space exhibits what is called 'unbounded nondeterminism' which can be shown to be equivalent to discontinuity in postconditions [6]. Specifications continue to enjoy Property 2:

**Theorem 3.5.** \(s\{P \land Q\} = s\{P\} \land s\{Q\}\).

**Proof.** We will not need the theorem and omit its proof. \(\square\)

Specifications also enjoy the following property whose importance will become clear shortly.
Theorem 3.6. Let $F[x]$ denote a specification dependent on specification $x$. Then $F \in (Spec \to Spec)$.

Proof. The proof is by induction on the structure of $F$. For the case of straight-line programs see [8]. The case of prescriptions is trivial: we forbid specification variables in prescriptions! The case of limit specifications follows from Lemma 2.3. □

Theorem 3.6 has an obvious generalization to the case of $n, n \geq 0$, specification variables.

Let us summarize at this point. We begin a particular programming task with a specification $P \parallel Q$. We take a first step, say, by finding specifications $s_1, s_2$ such that $P \parallel Q \subseteq s_1; s_2$.

If neither $s_1$ nor $s_2$ contain prescriptions, then we have solved the problem, for Theorem 3.4 tells us that $[P \Rightarrow (s_1; s_2)\{Q\}]$. Otherwise, we refine $s_1$ and $s_2$, in turn, yielding, let us say:

$$s_1 \subseteq \text{if } g_1 \to p_1 \Box g_2 \to p_2 \Downarrow,$$

$$s_2 \subseteq p_3; p_4.$$

Appealing to Theorem 3.6 twice, we can infer firstly

$$P \parallel Q \subseteq \text{if } g_1 \to p_1 \Box g_2 \to p_2 \Downarrow; s_2$$

and then

$$P \parallel Q \subseteq \text{if } g_1 \to p_1 \Box g_2 \to p_2 \Downarrow; p_3; p_4.$$

If now $p_1$ to $p_4$ contain no prescriptions we are done. Otherwise we press on until, all going well, we arrive at a program. We will have simultaneously constructed and verified the program if in each replacement of a prescription $P \parallel Q$ by $s$ we are sure that $P \parallel Q \subseteq s$.

4. Program construction

We have taken the view that programming consists in constructing a monotonic sequence of specifications beginning with a prescription and ending with a program. At each step we extend the monotonic sequence by replacement or by taking a limit. We consider each of these in turn.

4.1. Replacement

Given $P \parallel Q$ there are six ways of choosing $s \supseteq P \parallel Q$: $s$ will be a skip, an assignment, a prescription, an if-statement, a composition, or a block. There is a rule governing each of these choices. We present the rules as theorems notwithstanding the fact that their proofs are trivial deductions from Theorem 3.4: they gain their status by their usefulness.
Theorem 4.1. $P \parallel Q \equiv \text{skip}$ iff $[P \Rightarrow Q]$.

Proof. Theorem 3.4, (b), Lemma 3.1. □

Theorem 4.2. $P \parallel Q \equiv b := e$ iff $[P \Rightarrow Q[b; e]]$.

Proof. Theorem 3.4, (c), Lemma 3.1. □

Theorem 4.3. $P \parallel Q \equiv R \parallel S$ iff ([P \Rightarrow R] and [S \Rightarrow Q]) or [not P].

Proof.

\[
P \parallel Q \equiv R \parallel S
= [P \Rightarrow R] \parallel [S\{Q\}]
= [P \Rightarrow R \text{ and } [S \Rightarrow Q]]
= [P \Rightarrow R \text{ and } [P \Rightarrow [S \Rightarrow Q]]]
\]

by predicate calculus

\[
= ([P \Rightarrow R] \text{ and } [S \Rightarrow Q] \text{ or } [\text{not } P])
\]

by predicate calculus

\[
= ([P \Rightarrow R] \text{ and } [S \Rightarrow Q] \text{ or } [\text{not } P])
\]

by predicate calculus. □

Theorem 4.4. $P \parallel Q \equiv \text{IF}$ iff

$[P \Rightarrow g_1$ or $g_2]$ and $(P \text{ and } g_1\parallel Q \equiv s_1)$ and $(P \text{ and } g_2\parallel Q \equiv s_2)$.

Proof. Apply Theorem 3.4 and (e); the proof then proceeds as in [5]. □

Theorem 4.4 has an obvious generalization to an if-statement with many guarded commands. We also observe as an aside that if our language admitted (boolean) functions we would allow $g_1$ and $g_2$ in $\text{IF}$ to be assertions rather than boolean expressions.

Theorem 4.5. $P \parallel Q \equiv R \parallel S; T \parallel U$ if $[P \Rightarrow R], [S \Rightarrow T], \text{ and } [U \Rightarrow Q]$.

Proof.

\[
P \parallel Q \equiv P \parallel S; S \parallel Q
= P \equiv P \parallel S; S \parallel Q\{Q\}
= P \equiv P \parallel S\{S\}
= P \equiv P
= \text{true}
\]

Now apply Theorems 3.6 and 4.3 twice. □

Theorem 4.6. $P \parallel Q \equiv [\text{new } b; s]$ if $b$ does not occur in $P$ or $Q$, and $P \parallel Q \equiv s$. 
The only purpose of a block is to enlarge the state-space thereby giving us extra freedom in the subsequent refining of $s$.

Although Theorems 4.1-4.6 cover all the possible choices in constructing straight-line programs, nonetheless any practical methodology will have auxiliary rules for shortening the way. The following lemmas state some such rules.

**Lemma 4.7.** If $P \parallel Q \subseteq s$, and $R \parallel S \subseteq s$, then

$$(P \text{ and } R) \parallel (Q \text{ and } S) \subseteq s,$$

$$(P \text{ and } R) \parallel (Q \text{ or } S) \subseteq s$$

and

$$(P \text{ or } R) \parallel (Q \text{ or } S) \subseteq s.$$

**Proof.** Exercise. □

**Lemma 4.8.** $P \parallel Q \subseteq (P \{s(Q); s\}) \text{ and } s\{P\} \parallel Q = (s; P \parallel Q)$.

**Proof.** Exercise. □

Lemma 4.7 is useful, for example, when we have a program $s \supseteq P \parallel Q$ but we wish to use $s$ in an enlarged state-space, the additional variables satisfying predicate $R$. If $s$ assigns to no variables in $R$ it follows that $R \parallel R \subseteq s$ and hence, by Lemma 4.7, $(P \text{ and } R) \parallel (Q \text{ and } R) \subseteq s$. Lemma 4.8 is useful when we know that a refinement must include a statement $s$. This arises, for example, in the construction of recursive procedures where $s$ might be a statement to decrease a 'variant function'; this will be explained in the next section.

We can also prove now the following lemma on the behaviour of lubs of specifications; we will use it in the next section.

**Lemma 4.9.** $\bigcup \{P_j \parallel Q: j \in J\} = \bigcup \{P_j: j \in J\} \parallel Q$ for any set of prescriptions $\{P_j\parallel Q: j \in J\}$ with common postcondition $Q$.

**Proof.** (i) (upper bound). For any $j \in J$ we have

$$P_j \supseteq \bigcup \{P_j: j \in J\} \quad \text{by definition of } \bigcup$$

$$= [P_j \rightarrow \bigcup \{P_j: j \in J\}] \quad \text{by Lemma 3.1}$$

$$\Rightarrow P_j \parallel Q \supseteq \bigcup \{P_j: j \in J\} \parallel Q \quad \text{by Theorem 4.3.}$$
(ii) (least upper bound). Let \( L \) be any upper bound of \( \{P_j \mid Q : j \in J\} \). Hence

\[
P_j \parallel Q \subseteq L \quad \text{for all } j \in J
\]

\[
= P_j \subseteq L\{Q\} \quad \text{for all } j \in J \quad \text{by Theorem 3.4}
\]

\[
= \bigsqcup \{P_j : j \in J\} \subseteq L\{Q\} \quad \text{by definition of } \bigsqcup
\]

\[
= \bigsqcup \{P_j : j \in J\} \parallel Q \subseteq L \quad \text{by Theorem 3.4}. \quad \square
\]

4.2. Limits

It may well turn out that in a refinement of \( P \parallel Q \), \( P \parallel Q \) again appears. For example, with

\[
P_0: \quad 0 \leq i \leq n \quad \text{and } f = i!
\]

\[
Q_0: \quad f = n!
\]

we might arrive (after several steps) at

\[
P_0 \parallel Q_0 \subseteq \text{if } i = n \rightarrow (P_0 \text{ and } i = n) \parallel Q_0
\]

\[
\square \ i < n \rightarrow i := i + 1; f := f \times i;
\]

\[
P_0 \parallel Q_0
\]

\[
f
\]

We capture such a relationship by writing

\[
P \parallel Q \equiv F[P \parallel Q]. \quad (4)
\]

We can advance from (4) along the monotonic sequence by applying the same refinement(s) to \( P \parallel Q \) in \( F \). Indeed, we can repeat it as often as we like and so we infer from Lemma 2.4

\[
P \parallel Q \equiv F^\infty[P \parallel Q]. \quad (5)
\]

Now we attribute to procedure \( f \) defined by \( f : F[f] \) the meaning

\[
f = \mu x. F[x]
\]

—we do so because we find it convenient, and because it admits of a good implementation of \( f \) when \( F \) contains no prescriptions [8]. Note that by Lemma 2.5 \( \mu x. F[x] \) can be constructed using limit specifications—it equals \( F^\infty[\text{abort}] \)—and so is not an extension of our specification language. We are interested in using \( f \) to implement \( P \parallel Q \).

**Theorem 4.10.** Given procedure definition \( f : F[f] \) and prescription \( P \parallel Q \subseteq F[P \parallel Q] \),

\[
P \parallel Q \subseteq f \quad \text{iff } f = F^\infty[P \parallel Q].
\]
Proof. The implication to the left follows immediately from (5). For the implication to the right

\[ P \parallel Q \subseteq f \]

\[ \Rightarrow F^\infty[P \parallel Q] \subseteq F^\infty[f] \quad \text{by Theorem 3.6} \]

\[ \Rightarrow F^\infty[P \parallel Q] \subseteq f \quad \text{as } f \text{ a fixed point of } F. \]

Also, using Theorem 3.6 and Lemma 2.5 we have

\[ f = F^\infty[\text{abort}] \subseteq F^\infty[P \parallel Q] \]

The result follows by antisymmetry of \( \subseteq \). \( \Box \)

When we construct a relationship such as (4) there is no guarantee, in general, that our enterprise in passing to (5) is useful—if \( F \) happened to be the identity function on specifications to take a trivial example, then \( F^\infty[P \parallel Q] \) is nothing but \( P \parallel Q \)! What Theorem 4.10 assures us is that passing from (4) to (5) is useful—i.e. allows us to dispense with any further refinement of \( P \parallel Q \)—precisely when \( P \parallel Q \subseteq f \); it justifies the statement that the second method of extending the monotonic sequence of specifications is by taking the limit of all preceding specifications.

There is a more workable criterion for deciding if \( f \) implements \( \mathcal{P} \parallel \mathcal{Q} \). It uses the well-known device of a variant function, but applied to a recursive procedure rather than a loop, and using an ordinal-valued rather than an integer-valued function. The use of ordinals rather than natural numbers is forced on us by the presence of unbounded nondeterminacy, and in defining weakest preconditions was apparently first employed by Boom [4].

**Theorem 4.11.** Given procedure definition \( f:F[f] \), prescription \( P \parallel Q \), \( t \) a function on \( \text{Sta} \), and ordinal \( \rho \) such that

\[
[P \Rightarrow t \in \rho] \quad (\$)
\]

and

\[
P \land t = \lambda \parallel Q \subseteq F[P \land t < \lambda \parallel Q] \quad \text{for all } \lambda \in \rho, \quad (*)
\]

we have

\[ P \parallel Q \subseteq f. \]

**Proof.** We first show by transfinite induction

\[ P \land t = \lambda \parallel Q \subseteq f \quad \text{for all } \lambda \in \rho. \]

Assume for given \( \lambda \in \rho \),

\[ P \land t = \alpha \parallel Q \subseteq f \quad \text{for all } \alpha < \lambda. \quad (\star) \]
Then

\[(*)\]

\[= \bigsqcup \{P \text{ and } t = \alpha \parallel Q: \alpha < \lambda\} \subseteq f\]

by definition of \(\bigsqcup\)

\[= \bigsqcup \{P \text{ and } t = \alpha: \alpha < \lambda\} \parallel Q \subseteq f\]

by Lemma 4.9

\[= (P \text{ and } \exists \alpha (\alpha < \lambda: t = \alpha)) \parallel Q \subseteq f\]

by Lemma 3.2, predicate calculus

\[\implies P \text{ and } t < \lambda \parallel Q \subseteq f\]

by predicate calculus, Theorem 3.2

\[\implies F[P \text{ and } t < \lambda \parallel Q] \subseteq F[f] = f\]

by Theorem 3.6, \(f\) a fixed point

\[\implies P \text{ and } t = \lambda \parallel Q \subseteq f\]

by (\(\#\)), transitivity of \(\subseteq\).

This completes the induction. Hence

\[\bigsqcup \{P \text{ and } t = \lambda \parallel Q: \lambda \in \rho\} \subseteq f\]

\[= P \text{ and } t \in \rho \parallel Q \subseteq f\]

by arguments used above

\[= P \parallel Q \subseteq f\]

by (\(\$\)), predicate calculus.

\[\square\]

Theorem 4.11 generalizes the well-known operational argument for termination of a recursive procedure \(f\): if there exists an integer function \(t\) of the current state bounded from below by zero, and if \(t\) is decreased by at least one before each recursive call of \(f\), then \(f\) terminates. In the example on factorials that begins the section, we see \(P0 \Rightarrow n - i \geq 0\), \(n - i\) is decreased by 1 before \(P0 \parallel Q0\) in the body, and hence \(P0 \parallel Q0 \subseteq \text{fac}\), where

\[
\text{fac: if } i = n \rightarrow P0 \text{ and } i = n \parallel Q0
\]

\[\square i < n \rightarrow i := i + 1; f := f \ast i;\]

\[
\text{fac}
\]

\[\fi.\]

See [4] for an example requiring the generality of ordinals.

It may well turn out that, although we establish \(P \parallel Q \subseteq f\) where \(f:F[f]\), \(P \parallel Q\) again turns up in the subsequent refinement of \(F\). It is easy to see that we can once again replace \(P \parallel Q\) in \(F\) by \(f\), provided we check for usefulness with the same variant \(t\), for relationship (4) does not assume just one occurrence of \(P \parallel Q\) in \(F\). We leave it to the reader to confirm that a different variant function for each occurrence would not suffice.
One may also prove termination using well-founded sets—partially ordered sets in which each non-empty subset possesses a minimal element:

**Theorem 4.12.** Given procedure definition $f : F[f]$, prescription $P \parallel Q$, $t$ a function on $Sta$, and well-founded set $C$ such that

\[
[P \Rightarrow t \in C]
\]

and

\[
P \text{ and } t = x \parallel Q \subseteq F[\text{P and } t = x \mid Q] \quad \text{for all } x \in C,
\]

we have

\[
P \parallel Q \subseteq f
\]

($t = x$ denotes $t \equiv x$ and $t \neq x$).

**Proof.** The proof proceeds just as the proof of Theorem 4.11, but using induction on well-founded sets; it is left as an exercise. \qed

4.3. More limits

This section may be omitted without loss of continuity.

Relationship (4) can be generalized to a number of mutually dependent relationships; we consider the pair

\[
P \parallel Q \subseteq F[P \parallel Q, R \parallel S],
\]

\[
R \parallel S \subseteq G[P \parallel Q, R \parallel S]
\]

and show in outline how the theory of the preceding section generalizes to a solution of the pair (6).

Specifications $P \parallel Q$ and $R \parallel S$ must be resolved simultaneously. Following [2] we define for given specifications $p$, $q$:

\[
\langle F, G \rangle[p, q] = \langle F[p, q], G[p, q] \rangle.
\]

It is a routine exercise to show

\[
\langle F, G \rangle \in (Spec \times Spec \rightarrow Spec \times Spec).
\]

Therefore, we can rewrite (6) as

\[
\langle P \parallel Q, R \parallel S \rangle \subseteq \langle F, G \rangle[P \parallel Q, R \parallel S].
\]

We attribute to procedures $f_1$ and $f_2$ defined by

\[
f_1 : F[f_1, f_2] \quad \text{and} \quad f_2 : G[f_1, f_2]
\]

the meaning

\[
\langle f_1, f_2 \rangle = \mu x. \langle F, G \rangle(x).
\]

Analogous to Theorem 4.11, given function $t$ on $Sta$, and ordinal $\rho$ such that

\[
[P \Rightarrow t \in \rho], \quad [R \Rightarrow t \in \rho]
\]
and

\[(P \land t = \lambda \parallel Q, R \land t = \lambda \parallel S)\]

\[\subseteq (F, G)[P \land t < \lambda \parallel Q, R \land t < \lambda \parallel S] \quad \text{for all } \lambda \in \rho,\]

we have \(P \parallel Q \subseteq f1\), and \(R \parallel S \subseteq f2\). The proof, which is similar to the proof of Theorem 4.11, but with appeals to Lemma 2.2, is left to the reader.

Considering again relationship (4), it is evident that we can implement \(P \parallel Q\) with a procedure that exhibits so-called 'tail-recursion' only, i.e. each recursive call is a dynamically last call [8]. This must be so because we are trying to use procedure body \(F\) to establish \(Q\), and as \(P \parallel Q\) does so, any statements in \(F\) dynamically following \(P \parallel Q\) would be superfluous.

A more general form of recursion is created by encountering

\[P \parallel Q \subseteq F[P[b:h(b)] \parallel Q[b:h(b)]]\]

where \(h(b)\) denotes some function of \(b\). For example, we might arrive at

\[n \geq 0 \parallel f = n! \subseteq \text{if } n = 0 \rightarrow f := 1\]

\[\square n > 0 \rightarrow n - 1 \geq 0 \parallel f = (n - 1)!; f := f \times n\]

\[\text{fi.}\]

We could admit parameterized procedures to reason about such relationships, but as it would not contribute essentially to our thesis we decline to do so in the present paper.

Finally, we make brief mention of loops. The loop

\[\text{do } g \rightarrow s \text{ od}\]

is equivalent to \(\text{DO}\) defined by

\[\text{DO: if } g \rightarrow s; \text{ DO } \not{g} \rightarrow \text{skip fi.}\]

One usually reasons with \(\text{DO}\), when \(s\) denotes a program, using an 'invariant relation' \(P\) and a variant function \(t\): if for all natural numbers \(n\)

\[P \land g \parallel P \subseteq s \quad \text{and} \quad [P \Rightarrow t \geq 0]\]

and

\[P \land g \land t \leq n \parallel t < n \subseteq s\]

then

\[P \parallel P \land \not{g} \subseteq \text{DO}\]

We leave it to the reader to verify the loop rule.
5. Discussion

We have converted a given programming language by adding to it prescriptions which are used to specify desired mechanisms. Despite the fact that prescriptions may not be realizable we have shown that they have a good weakest precondition semantics, and so specifications and programs inhabit the same semantic framework. As a result, program construction can be nicely formalized because the intermediate 'programs' of the construction process always have a formal status and in a very rigorous sense may be shown to be correct with respect to the initially given requirements.

Specifications admit of a natural partial ordering, by regarding them as monotonic functions on assertions. With respect to this partial ordering, the specification/program's that arise in the construction of a program form a monotonic sequence. Moreover, each specification is derived either from its immediate predecessor by replacement or by taking the limit of its predecessors.

The cost of admitting prescriptions to the language is two properties: the law of the excluded miracle, and continuity in postconditions. These losses are not serious in a specification language. The admission of miracle has as its consequence that we may specify unrealizable mechanisms and attempt futilely to implement them. But we have always been free to attempt the impossible, and prescriptions simply allow us to do so formally.

The loss of continuity in postconditions has the consequence that instead of using integer-valued variant functions to reason about termination, we must use functions yielding values in the ordinals or in some well-founded set. But ordinals and well-founded sets are not unpleasant things and, in any case, integer-valued variant functions will still suffice in many, even most, cases. That a specification is discontinuous in postconditions has no bearing on the existence of an implementation, for we can equate discontinuity with unbounded nondeterminacy [6] and nondeterminacy decreases along chains: skip, for example, is an implementation of chance.

We have presented a formal view of the process of constructing programs according to the calculus described in [5, 7, 8]. The formal view clarifies the calculus somewhat by showing that each constructive step is one of replacement or limit taking and by presenting as theorems what in [5, 7, 8] are axioms or informal statements. It further suggests that we might usefully put effort into designing not mere programming languages but unified languages in which we can both specify and program, and in which we can smoothly transform specifications into programs. Although programming languages are good for expressing the final product, they tend to lack good notational support for discovering and refining programs, and they do not encourage us to present a final text that records both the program and its justification. The use of prescriptions illustrates that such an extended language can be expressive to the extent of admitting unrealizable mechanisms, and that such expressiveness is advantageous.
We know well that the design of a language should be driven by the methodology it is intended to support, but in practice this principle has been used philosophically rather than technically. I believe that formalizing the methodology will enable us to employ the principle with more effect by shifting language design from sociology to science where it properly belongs.

The calculus we have been treating has been used in [5, 7, 8] to develop many elegant, albeit small, programs. The calculus nevertheless falls short of being a universal medium of program construction. With large programs the formulae could become quite unwieldy and we would quite likely find ourselves entwined in many cumbersome proofs. It remains to discover good factoring methods, notations, and perhaps a collection of useful theorems before such a calculus would be generally useful.

6. Related work and acknowledgements

Weakest precondition semantics and the calculus of programming are due to Dijkstra [5]. The incorporation of recursion and predicate pairs into the calculus is due to Hehner. Hehner's predicate pairs have the status of procedure names; the basic insight in the present paper is to grant them a formal status of their own. I have taken much notation from [2, 5, 8], and much inspiration from [8]. The idea of embedding programs and specifications in the one framework has appeared in the writings of many authors, for example [1, 9-11]. I received critical comments on a draft of the present paper from H. Gibbons, E.C.R. Hehner, and A. Mullins. To two most meticulous referees, Carroll Morgan and W.H.J. Feijen, I am extremely grateful. Carroll Morgan has independently discovered prescriptions. I thank Susannah Dean for typing.

References


