ED 380 500 TM 022 863

AUTHOR Wadkins, J. R. Jefferson
TITLE Program Control as a Set-Theoretic Concept. Research Report RR-94-56.
INSTITUTION Educational Testing Service, Princeton, N.J.
PUB DATE Dec 94
NOTE 28p.
PUB TYPE Reports - Evaluative/Feasibility (142)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS Computer Software Development; *Definitions; Programming; *Programming Languages; *Semantics; *Set Theory

ABSTRACT

This paper provides operational semantics for imperative programming languages that legitimize the phraseology used in the statement and proof of a fundamental theorem of program correctness. Some of the phrases used in the theorem are normally undefined, but intuitively appealing. This paper attempts to give precise meaning to the questionable phrases in the theorem. The real value of this theorem lies in a corollary that provides a programming template for the construction of a loop with a built-in proof of its correctness just by writing two segments of straight-line code to fit given specifications (without having to visualize a repetitive process). In defining this language, a distinction is made between strings and sequences, with strings defined as equivalent classes of sequences. Also essential is the definition of an imperative programming language and of program control, which turns out to be a function mapping each pair consisting of a position (in the program) and a state into a (possibly different) pair also consisting of a position and a state. Appendixes present a minimal example and a proof of the theorem. (SLD)

RR-94-56

RESEARCH REPORT

PROGRAM CONTROL AS A SET-THEORETIC CONCEPT

J. R. Jefferson Wadkins

Educational Testing Service
Princeton, New Jersey
December 1994

PROGRAM CONTROL AS A SET-THEORETIC CONCEPT

by J. R. Jefferson Wadkins

PROGRAM CONTROL AS A SET-THEORETIC CONCEPT

by J. R. Jefferson Wadkins

A point does not move, a circle does not shrink, a number does not change its value, and a function does not decrease. Each of these mathematical entities is a set; it is static; it takes no action; it does not change; it just sits there being a set. Nevertheless, the notion of variable mathematical entities is present in most mathematical activities and permeates the teaching of mathematics.

Thus teachers feel no shame in saying things like "as the point (r, A) moves from right to left on this curve, the circle with radius r shrinks...", because they are justifiably confident that the language they use, which employs the notion of variable mathematical entities, can be converted to the static language of set theory in the context of axioms for a complete ordered field. While there are purists who frown upon use of "variables" in serious mathematical discourse, most mathematicians are comfortable with such phraseology because of the existence of precise semantics that legitimize these notions in terms of set-theoretic concepts from which the purist might prefer never to emerge.

1.0 PURPOSE AND MOTIVATION FOR THIS PAPER

Precise proofs of program correctness and precise proofs in calculus are arguments about static entities. Both are typically introduced with teacher as actor and students as spectators, but there is an implicit understanding that students are only being "exposed" to these techniques. The answer to the ever-present, student-as-spectator question, "Are we responsible for this on the next test?" is almost always negative in both cases. However, there is a vast difference in how introductory courses handle intuitive notions consistent with corresponding precise ideas.

In typical beginning calculus courses, the notion of "variable" is employed to give plausibility arguments for fundamental theorems that are seldom stated in such courses very precisely, and seldom justified with epsilon-delta arguments; but students do gain an appreciation of the fundamental nature of such theorems by constantly taking part in both home-work exercises and classroom activities that employ "variables" to reason about specific applications of those fundamental theorems. Although there is a programming notion used in introductory courses for other purposes that could be used to give convincing arguments for both fundamental theorems of program correctness and instances of their application to code written by beginning programmers, use of this intuitive notion in teacher presentations which give plausibility arguments that code segments satisfy their informal specifications would seem to be relatively rare.

That programming notion is "program control", an entity used by both neophytes and professional programmers for "desk checking" of code before testing it. The readiness of mathematics teachers to use hand-waving arguments in calculus presentations, as opposed to the hesitancy of computer science teachers to use the notion of "program control" in their classroom presentations, is probably best explained by the fact that "real variables" have a widely understood logical basis, while "program control" is often thought to be a mere heuristic, unrelated (indeed, contrary to) precise proofs of correctness about static entities employing the weakest-precondition predicate transformer.

Use of "program control" provides a short and simple argument for a fundamental theorem of program correctness*, which Edsger Dijkstra stated without proof in his seminal 1975 paper [1] and whose proof, using the symbolism and tools of the formal predicate calculus, is outlined (as the answer to three exercises) taking up more than two pages of densely packed symbolism in The Science of Programming by David Gries [2].

The purpose of this paper is to provide operational semantics for imperative programming languages that legitimize the phraseology used in the statement and proof of our version of that fundamental theorem. We would venture to claim that both the statement and proof of our version of this theorem (as given in the next section) can be made understandable by, and convincing to, typical beginning students who have no preparation other than what is typically offered in the first half of CS1 -- and that this can be accomplished with very little effort on the part of the teacher. This is clearly not the case with even the statement of the theorem in the development of either Dijkstra [1] or Gries [2].

* That argument and that theorem is given at the beginning of Section 1.1.

Motivation for the research that culminated in this paper was two-fold. The first was a question posed to this writer by a high school teacher in 1988.[4] The second was a number of rewrites by David Gries of arguments offered by this writer during collaboration on a paper [3] in 1989-90. The question from the teacher, Alexander Z. Warren, arose when he and two other high school teachers of computer science were asked to review an earlier paper on loop invariants by this writer. (The collaboration on [3] was kindled by Gries' review of the earlier paper.) The earlier paper was intended to convince those high school teachers of computer science who were also teachers of mathematics that the mysterious concept of a loop invariant could be easily comprehended if they considered it merely the induction hypothesis for an induction proof that the loop accomplishes what the programmer intended. Warren's otherwise positive reaction to the paper was tempered by worries about an argument in that paper referring to program control. Those worries are best summarized by the question, "What are the axioms?" Additionally, the rewrites by Gries during collaboration on [3] always resulted in exclusion of any reference to program control. Those repeated exclusions coming after Warren's tantalizing question led to this writer's determination to validly give the answer, "The same as those for set theory." Our paper here does provide the validity for that answer.

1.1 A FUNDAMENTAL THEOREM OF PROGRAM CORRECTNESS

Several phrases used in the following theorem and its proof are normally undefined, but intuitively appealing. We here contend that programming students would not question the validity of their use any more than mathematics students question the normally undefined, but intuitively appealing, phraseology of "variables". The purpose of this paper is to give precise meaning to the questionable phrases used in this section.

1.1.1 <u>Theorem</u>: In some program containing no **goto**'s, let *Expr* denote an expression that is a function of the program variables, let c be a number, let *Post* and *Inv* be statements about current values of program variables, and let W denote the following loop.

```
1:
    while <guard> do
    begin
2:
       <body>
3:
    end
4:
```

(1) A sufficient condition that W terminate is that, during any execution of W,

(i) both <guard> and <body> terminate,

(ii) no program variable changes value as a result of any evaluation of <guard>,

(iii) the statement "$Expr \leq c$" is true each time program control reaches <guard>, and

(iv) whenever program control reaches line 2, if v_0 denotes the value of *Expr* at that time, then the value of *Expr* is at least v_0+1 the next time program control reaches line 3.

(2) A sufficient condition that *Post* be true whenever program control reaches line 4 is that

(v) (*Inv* **and not** <guard>) implies *Post*

(vi) *Inv* is true each time program control reaches <guard>. and

(vii) during any execution of W that terminates, no program variable changes value as a result of any evaluation of <guard>.

Proof of (1): Let E be any execution of W, and assume (i), (ii), (iii), and (iv). By (i), program control must reach line 3 subsequent to any time during E that it reaches line 2; and, by (iv), the value of *Expr* must increase by at least 1 each time <body> is executed. *Expr* has some value v_0 the first time program control reaches <guard> during E. By (iii), "$v_0 \le c$" is true at that time. There are only finitely many adjacent intervals of unit length beginning at v_0 and ending at a number that is at most c, so <body> can only be executed finitely many times during E (perhaps zero times) -- for, otherwise, because of (ii) and (iv), the value of *Expr* would have to increase beyond c during some execution of <body>, rendering assumption (iii) false. Thus, it must also be true that <guard> can only be executed finitely many times; so E must terminate, which completes the proof that W terminates.

Proof of (2): Let E be any execution of W that terminates, and assume (v), (vi) and (vii). We must show that *Post* is true when program control reaches line 4 at the end of E. Because of termination, there is a last time t that program control reaches <guard>, and since *Inv* is true each time program control reaches <guard>, *Inv* is true at time t. Also "**not** <guard>" is true at time t -- because, otherwise, t would not be the last time program control would reach <guard>. Thus by (v), *Post* is true at time t. By (vii), no program variable can change value during the execution of <guard>, so since *Post* is a statement about current values of the program variables, *Post* must still be true when program control leaves <guard> the last time during E and then reaches line 4 as E terminates. q.e.d.

Note on a theorem parallel to Theorem 1.1.1: It should be clear that the argument indicated above is easily adjusted to form the proof of the parallel theorem obtained by substituting "$\ge$" for "$\le$" and substituting "at most v_0-1" for "at least v_0+1" in (iii) and (iv), respectively, of part (1) of Theorem 1.1.1. This substitution comes close to Dijkstra's use of "variant function", which Gries calls a "bound function". However, there is a difference: for our theorem, "c - *Expr* ≥ 0" is an invariant, whereas in the Dijkstra-Gries approach, the bound function (which corresponds to our "c - *Expr*") is only required to be nonnegative up until the final evaluation of the guard; upon termination, the variant function (bound function) is allowed to have a negative value. Finally, a variant function (bound function) is required to be integer-valued, whereas c - *Expr* has no such restriction.

Note on the proof of (1): It should be clear that the proof of (1) goes through if any arbitrary positive number d were substituted for the increment 1 in the "v_0+1" of (iv) in the statement of Theorem 1.1.1, and in the "v_0-1" of the "parallel theorem" just discussed. (The real numbers form an Archimedean field.) However, the only practical effect of such a substitution known to this writer is in using the invariant inequality "*Expr* ≥ 0", where *Expr* $= \log_2(k+1)$, in the proof of termination for any loop that uses "$k := k$ **div** 2" to make progress toward termination -- in which case the decrement to k from 2 to 1 results in only a decrement to *Expr* of $\log_2 3 - \log_2 2$, which equals $\log_2(3/2) < 1$. Termination can be proved in such a case using either k (if proof of termination is the only goal) or $\lceil \log_2(k+1) \rceil$ (if an expression that counts iterations of the loop is desired). Thus, there seems no good reason to complicate matters in the statement of a theorem claimed to be understandable to the most novice of programmers.

For writing code, the real value of Theorem 1.1.1 lies in the following corollary, which provides a programming template for the construction of a loop with a built-in proof of its correctness just by writing two segments of straight-line code to fit given specifications, i.e., without having to visualize a repetitive process.

1.1.2 Corollary: Using the notation of Theorem 1.1.1: In order to write correct code for a loop intended to establish assertion *Post*, it is sufficient that

(1) a Boolean program expression, <guard>, and a statement *Inv* about the program variables be chosen such that <guard> terminates and has no side effects, and such that the implication "(*Inv* **and not** <guard>) implies *Post*" is true,

(2) code be written as an initialization that terminates and makes *Inv* true in line 1,

(3) a number c and an expression *Expr* be chosen such that

  (i) "(<guard> **and** $Expr \leq c$) implies $Expr+1 \leq c$" and
  (ii) the already-written initialization also makes "$Expr \leq c$" true in line 1, and

(4) code for <body> be written that

  (iii) terminates and contains no **goto**,
  (iv) increases the value of *Expr* by at least 1, and
  (v) has the property that if "<guard> **and** *Inv* **and** $Expr \leq c$" is true when program control reaches line 2 during any execution of W, then "*Inv* **and** $Expr \leq c$" must be true the next time program control reaches line 3.

This Corollary follows from Theorem 1.1.1 and its argument is in the same spirit as the argument for the Theorem, and it is omitted.

While the purist might argue that the wording of the Corollary encourages operational thinking in the process of creating a loop, as opposed to the static reasoning used in the predicate-transformation approach, such operational thinking is restricted to two segments of straight-line code (the initialization and the body), a restriction that would certainly move the student in the direction of static reasoning and away from the total dependence on the vision of repetitive processes that usually permeate the teaching of loop creation in introductory courses.

2.0 THE OPERATIONAL SEMANTICS

Section 1 presented the fundamental theorem and its proof using terminology that might be considered plausible but not rigorous, intuitive but not precise. This Section 2 provides precise operational semantics for imperative programming language systems in terms of set theory. Section 3 provides the translation of the words and phrases of the fundamental theorem and its proof in Section 1 into the precise set-theoretic terms of Section 2, thereby refuting the natural conjecture that the theorem and its proof is "plausible but not rigorous, intuitive but not precise".

The only prerequisite for understanding the technical development of the operational semantics here is a high level of mathematical maturity. Although the mathematics is elementary, the extensive mathematical shorthand that is employed presents an intimidating facade that might be expected to prevent any but the most interested reader to gloss over the details of the proofs and definitions.

In mathematics we often have two choices for development of a system: 1) choosing axioms and definitions that state many of the properties objects satisfying those premises should have, but sometimes the price paid is an inordinate complexity in the proof of some fundamental theorems based on those premises; 2) choosing premises requiring more complexity in early development, but the reward is sometimes to make the proof of some fundamental theorems simpler.

To begin: The approach taken by Dijkstra in [1] using the weakest-precondition predicate transformer makes definition of his guarded-command language very straight-forward, without need for any concept of execution. A comparison of the proof of his fundamental theorem (as given in Gries [2]) with our proof of the corresponding Theorem (our Theorem 1.1.1), or with the reader's own plausibility argument for our version, shows the price paid for the simplicity of his early development. The price we pay here for a simple proof of this fundamental theorem is a heavy layer of complexity in the development of operational semantics from the axioms of set theory.

2.1 PREVIEW OF THE SEMANTICS

Many phrases appearing in Theorem 1.1.1 need precise definition, but the offending phrase that is most fundamental is "program control". In our development, "control" is a function taking ordered triples (S,s,p) into other such triples, where S is a string of "tokens", s is a "state", and p is a "position". A state is a function from the nonnegative integers (intuitively, the addresses in the memory of an ideal machine) into a set of "values" stored in ROM, with each "program" having some final segment of addresses available beyond ROM; a program is a finite sequence of tokens; and a position in the program is one of its subscripts.

For the stated purpose of this paper, the only programming construct needing definition is a loop. However, in our development, the syntactical form of a loop is the same as that of a simple conditional; and until we specify how such a construct is executed, there is no way to tell which is which. An execution is a sequence of ordered pairs in the domain of "program control", which is a function, derived from the system-defined function called "control", that maps pairs (s,p) of states and positions into other such pairs. The position of the first term in the execution of a program is the position of the first token of the program. It is a theorem that, for any state s and any program P, there exists a unique execution of P with $(s, 1)$ as its first term; and, in general, if there exists any execution of program segment S with $(s, min.Domain.S)$ as its first term, then that execution is unique.

Our use here of "*Domain.S*" indicates that we consider S to be a function, and we do indeed make strong use of the viewpoint that a sequence is a function whose domain is a segment of the integers and that a function is a set of ordered pairs no two of which have the same first element. While there are other legitimate points of view, the basis for the semantics to be given here is set theory, so every object used in the development is defined to be a set.*

For this development, we must distinguish between strings and sequences. Execution is only defined for program segments, which are sequences whose domains can differ, depending on their positions in parent programs. Yet a programming construct should be defined without reference to where it appears in a program, and should be distinguishable from its particular appearance in a program. The mechanism we use for this purpose is to define a string as an equivalence class of finite sequences with each member of a given class having the same length and the same terms in the same order. In this way, given a programming construct as a string, each appearance of that construct as a program segment (a sequence) is a member of the equivalence class that is the string. Conversely, given a subsegment of a program, there is a unique equivalence class, i.e., string, containing that subsegment as a member.

2.2 BACKGROUND NOTATION, CONVENTIONS, AND POINT OF VIEW

Because the view of a function as a set is so fundamental to our development, we state again for emphasis the characterization of functions already indicated.

2.2.1 Note: A relation is a set of ordered pairs. A function is a set of ordered pairs no two of which have the same first element.

We assume as familiar the usual notation for sets. As logical shorthand: "$\exists$ - $\ni$" denotes "there exist(s) - such that"; "$\forall$" denotes "for each"; and "$\wedge$" denotes "and". We use two kinds of definitions. One we call "Definition", wherein we define the meaning of a statement in terms of statements already defined. The other we call "Notation", which provides a new name to a set that is already defined. In "Definition", we use "$\equiv$" as shorthand for "denotes" whenever later reference is needed for a long expression. In "Notation", we also use "$\equiv$" whenever new symbolism is defined in terms of already-defined symbolism; and there we sometimes use the concept of a "class" for aggregates that are in a sense too large to be sets. This is for notational convenience; we perform no operations on such classes; so we need not worry about bringing down on our development the wrath of such things as Russell's Paradox. Finally, we employ the label "Observation" for theorems whose proof either follows directly from immediately preceding definitions or is a completely standard and well known exercise in mathematics. Each observation that we make facilitates understanding of a later definition, theorem, or proof.

2.2.2 Notation: "*Set*" denotes the class of all sets.

"*Relat*" denotes the class of all relations.

$\forall\ S \in Set$, "$\#S$" denotes the cardinality of S.

$\forall\ r \in Relat$, "$Dom.r$" and "$Rng.r$" denote the domain and range, respectively, of r.

$Z \equiv \{\, n \mid n \text{ is an integer}\} \wedge Z^+ \equiv \{n \in Z \mid n > 0\}$.

$\forall\ m \in Z, \forall\ n \in Z\cup\{\infty\}$, if $n \in Z$ then $Z_m{}^n \equiv \{k \in Z \mid m \le k \le n\} \wedge m..n \equiv Z_m{}^n$

$\wedge$ if $n = \infty$ then $Z_m{}^n \equiv \{k \in Z \mid m \le k\} \wedge Z_m{}^+ \equiv Z_m{}^\infty$.

* Actually, for convenient notational purposes, we shall recognize "classes" too large to be sets, but these could be eliminated from the development without loss of accuracy.

2.2.3 Observation: $Z^+ = Z_1^\infty \wedge \forall\, m,n \in Z,\ Z_m^n = m..n.$

2.2.4 Convention: $\forall\, k \in Z,\ \ k+\infty = \infty+k = \infty-k = \infty > k.$

Although the following notation is fairly standard, it seems appropriate to state it specifically.

2.2.5 Notation: $\forall\, S, T \in Set$, "$f: S \to T$" is a statement and

$$f: S \to T \iff (f \text{ is a function} \wedge Dom.f = S \wedge Rng.f \subseteq T)$$
$$\wedge\ T^S \equiv \{ f \mid f: S \to T\}$$
$$\wedge\ 2^S \equiv \{T \in Set \mid T \subseteq S\}.$$

$\forall\, n \in Z,\ \forall\, S \in Set,$

if $S \subseteq Z_n^+ \wedge S \neq \varnothing$ then "$min.S$" denotes the minimum of S

$\wedge$ if $\#S \in Z^+$ then "$max.S$" denotes the maximum of S.

2.3 SEQUENCES AND STRINGS

As already indicated, the view of a sequence as a function is not just a matter of taste; it is fundamental to our development. In the following definition, we restrict the meaning of "an infinite sequence" to mean "an infinite sequence with domain the nonnegative integers". These are the only kind of infinite sequences that we need in this paper. By restricting the meaning to the subject matter at hand, we can offer succinct comments to the reader, which are also technically true, but which would be technically false in more general contexts. On the other hand, the following definition also expands the meaning of "a finite sequence" to include functions whose domains are arbitrary segments of the integers -- not just the nonnegative integers. This expansion of the usual meaning of "a finite sequence" allows us to handle in the general case what would otherwise be special cases.

2.3.1 Definition: (sequences)

s is an infinite sequence $\iff (\exists\, A \in Set \ni s: Z_0^+ \to A)$

s is a finite sequence $\iff (\exists\, m,n \in Z_0^+ \wedge \exists\, A \in Set \ni s: m..n \to A)$

2.3.2 Notation: $\forall\, A \in Set,$

$$Seq.A \equiv \{ s \mid s: Z_0^+ \to A\}$$
$$\wedge\ Fsq.A \equiv \{ s \mid s \text{ is a finite sequence} \wedge Rng.s \subseteq A\}$$
$$\wedge\ \forall\, s \in Fsq.A,$$
$$SEG.s \equiv \{ t \mid \varnothing \neq t \subseteq s \wedge \forall\, k \in Z,\ \forall\, i,j \in Dom.t, \text{ if } i < k < j \text{ then } k \in Dom.t\}$$

For general functions, we often use "$f.c$" to denote the standard "$f(c)$" in order to minimize nested parentheses, employing parentheses only for functions with more than one argument or to avoid ambiguity. We generally use "f_c" whenever f is a sequence. (If f is a tuple, we also use f_c to denote the c^{th} component of f.)

2.3.3. Definition: (equivalent sequences)

$$s \text{ is } A\text{-equivalent to } t \iff \begin{cases} A \in Set \wedge s,t \in Fsq.A \\ \wedge\ \#s = \#t \\ \wedge\ \forall\, i \in 0..\#s-1,\ \ s_{min.Dom.s+i} = t_{min.Dom.t+i} \end{cases}$$

2.3.4 Observation: $\forall\, A \in Set$, "is A-equivalent to" is an equivalence relation on $Fsq.A$.

2.3.5 Definition: (strings)

$$Str \text{ is an } A\text{-string} \iff \begin{cases} A \in Set \wedge Str \subseteq Fsq.A \\ \wedge\ \forall\, s,t \in Fsq.A, \\ (s,t \in Str \iff s \text{ is } A\text{-equivalent to } t) \end{cases}$$

As previously indicated, program constructs will be defined as strings of tokens. These constructs are most easily remembered and recognized as the concatenation of singleton strings whose defining token is a mnemonic identifier. The notation "$S \frown T$" is used for the concatenation of strings S and T, and is precisely defined as follows.

2.3.6 Notation: $\forall\, A \in Set,\ \ STR.A \equiv \{\, x \mid x \text{ is a nonempty } A\text{-String}\}$

$\wedge\ \forall\, S, T \in STR.A,\ \ S \frown T$ denotes the set defined by:

$$u \in S \frown T \iff \begin{cases} \exists\, s \in S \wedge \exists\, t \in T \\ \ni u = s \cup t \wedge \#u = \#s + \#t \\ \wedge\ min.Dom.u = min.Dom.s \wedge max.Dom.s + 1 = min.Dom.t \end{cases}$$

2.3.7 Observation: $\forall\, A \in Set,\ \forall\, S, T, U \in STR.A,\ \ S \frown T \in STR.A \wedge S \frown (T \frown U) = (S \frown T) \frown U.$

In addition to the difference between sequences and strings, one should recognize that every finite sequence into a given set A has a unique A-string containing it. This idea is embodied more precisely in the following observation, which justifies the notation that follows it.

2.3.8 Observation: $\forall\, T \in Set,\ \forall\, S \in Fsq.T,\ \ \exists\, Str \in STR.T \ni S \in Str \wedge \forall\, St \in STR.T$, if $S \in St$ then $St = Str$.

2.3.9 Notation: $\forall\, T \in Set,\ \forall\, S \in Fsq.T,\ \ [S]$ denotes the unique member of $STR.T$ such that $S \in [S]$.

2.4 IMPERATIVE PROGRAMMING LANGUAGE SYSTEMS

In the definition of "imperative programming language system" given below: Sets *Tk*, *Val*, *Rel*, and *VAL* will represent tokens, values, relations, and $Val \cup \{\infty\}$, respectively. A "program" is a member of *Fsq.Tk* with certain properties. A "state" is a member of *Seq.VAL*. *FVal.P* and *ADR.P* represent, respectively, memory locations in ROM where finitely many values are stored, and locations (out beyond ROM) allotted to program P for its use. A position in the program is an element of the program's domain, i.e., a subscript of the program. The domain of the "control function", *Ctrl*, is the Cartesian product $Fsq.Tk \times Seq.VAL \times Z^+$. We use the projection functions *St* and *Pos* to pick out the state and position, respectively, from a given member of that domain. This motivates the following notation.

2.4.1 Notation: $\forall\, Tk, VAL \in Set,\ \forall\, (S', s, k) \in Fsq.Tk \times Seq.VAL \times Z^+,$

$$St(S', s, k) \equiv s \wedge Pos(S', s, k) \equiv k.$$

An "imperative programming language system" is a 14-tuple whose first three components, *Tk* (tokens), *Val* (values), *Rel* (relations), and last component, *Ctrl*, were discussed in the preceding paragraph. The next four components are *start*, *stop* (two tokens), *tru*, and *fals* (two values).* The other six are functions: *OprCorr*, *Com*, *Compat*, *Rcvr*, *v*, and *Loc*. These building blocks of programming systems, outlined in the table below, involve (a) receivers (strings that can legitimately appear on the receiving

* We refrain from appending "e" to these names to remind ourselves that these are just abstract values not known *a priori*.

end of an assignment statement [*]), hence *Rcvr*; (b) values, hence *v*; (c) locations where values are stored, hence *Loc*; (d) rules of type compatibility, hence *Compat*; (e) commands, hence, *Com*; and (f) a mechanism for determining operational correctness, hence *OprCorr*, where "*S* is operationally correct in *s*" means (intuitively) that *S* causes no error during execution either of the compiler or of the program containing *S* -- provided *s* is the "state of memory" when *S* is encountered in either execution (this does not prevent the execution of *S* from being nonterminating).

FUNCTION	TAKING	TO	INTUITION
OprCorr	(S, s)	*tru*	if *S* is operationally correct in state *s*
		fals	otherwise
Com	(S, s)	*tru*	if *S* is a command in state *s*
		fals	otherwise
Rcvr	(S, s)	*tru*	if *S* is a receiver in state *s*
		fals	otherwise
v	(S, s)	some value	if *S* is an "expression" (a *Val*-valued function of the receivers) in state *s*
		∞	otherwise
Loc	(S, s)	some address	if *S* is a receiver in state *s*
		∞	otherwise
Compat	$(S1, S2, ss)$	*tru*	if *S1*, *S2* are type-compatible "expressions" in state *s*
		fals	otherwise

The intuition behind the requirements labelled (1) - (8) in Definition 2.4.2 below is indicated as follows:

1) ∞ is not a value, every value is a string of tokens, and *Rel* is a set of relations between values.
2) Every value is operationally correct in every state and has itself as value in any state.
3) Every command is operationally correct.
4) A receiver in a given state is operationally correct and has its value stored in a location determined by that state.
5) Every value is compatible with itself; any two type-compatible strings are operationally correct; and two strings compatible with some value are compatible with each other.
6) Every value has a compatible receiver in some state and is stored in the location for that receiver determined by that state.
7) A program is a finite sequence of tokens, always starting and ending with *start* and *stop*, respectively; *Ctrl* stops at *stop*; its states have finitely many values stored in ROM; and *Ctrl* will not map any state or position in a segment out of its program.
8) Everything in *Rel* has a Boolean representing it.

A minimal example demonstrating the consistency of the following definition is in Appendix 1.

[*] Choosing "receiver of values" as primitive avoids the issue of whether things like "*A*[2]" are "variables".

2.4.2 Definition: (imperative programming language systems)

L is an imperative programming language system $\iff$

$\exists\ Tk, Val, VAL, Rel \in Set \wedge \exists\ start, stop \in Tk \wedge \exists\ tru, fals \in Val \wedge VAL = Val \cup \{\infty\}$

$\wedge \exists\ OprCorr : STR.Tk \times Seq.VAL \to \{tru, fals\}$

$\wedge \exists\ Com : STR.Tk \times Seq.VAL \to \{tru, fals\}$

$\wedge \exists\ Compat : STR.Tk \times STR.Tk \times Seq.VAL \to \{tru, fals\}$

$\wedge \exists\ Rcvr : STR.Tk \times Seq.VAL \to \{tru, fals\}$

$\wedge \exists\ v : STR.Tk \times Seq.VAL \to VAL$

$\wedge \exists\ Loc : STR.Tk \times Seq.VAL \to Z_0^+ \cup \{\infty\}$

$\wedge \exists\ Ctrl : Fsq.Tk \times Seq.VAL \times Z^+ \to Fsq.Tk \times Seq.VAL \times Z^+$

$\ni L = (Tk, Val, Rel, start, stop, tru, fals, OprCorr, Com, Compat, Rcvr, v, Loc, Ctrl)$

1) $\wedge\ start \neq stop \wedge tru \neq fals \wedge \infty \notin Val \subseteq STR.\ Tk \wedge Rel \subseteq 2^{Val \times Val}$

$\wedge \forall\ (S, s) \in STR.Tk \times Seq.VAL,$

2) if $S \in Val$ then $OprCorr(S, s) = tru \wedge v(S, s) = S$

3) $\wedge$ if $Com(S, s) = tru$ then $OprCorr(S, s) = tru$

4) $\wedge$ if $Rcvr(S, s) = tru$ then $OprCorr(S, s) = tru \wedge s_{Loc(S,s)} = v(S, s) \in Val$

5) $\wedge \forall\ c \in Val, \forall\ s \in Seq.Val,\ Compat(c, c, s) = tru$

$\wedge \forall\ S1, S2 \in STR.Tk$

if $Compat(S1, S2, s) = tru$ then $Compat(S2, S1, s) = tru = OprCorr(S1, s) = OprCorr(S2, s)$

$\wedge$ if $Compat(S1, c, s) = tru = Compat(S2, c, s)$ then $Compat(S1, S2, s) = tru$

6) $\wedge \forall\ c \in Val,\ \exists\ (S, s) \in STR.Tk \times Seq.VAL$

$\ni Rcvr(S, s) = tru \wedge Compat(S, c, s) = tru \wedge v(S, s) = c = s_{Loc(S,s)}$

7) $\wedge$ if $\mathcal{P}_L$ is the set defined by

$$P \in \mathcal{P}_L \iff \begin{cases} P \in Fsq.Tk \wedge min.Dom.P = 1 \wedge P_1 = start \wedge P_{\#P} = stop \\ \wedge \exists\ s \in Seq.VAL \ni OprCorr([P], s) = tru \end{cases}$$

then $\exists\ ADR : \mathcal{P}_L \to \{\ Z_m^+ \mid m \in Z^+\} \wedge \exists\ ST : \mathcal{P}_L \to 2^{Seq.VAL} \wedge \exists\ FVal : \mathcal{P}_L \to 2^{Val}$

$\ni \forall\ P \in \mathcal{P}_L, \forall\ s \in Seq.VAL,\ Ctrl(P, s, \#P) = (P, s, \#P)$

$\wedge \forall\ c \in FVal.P, \exists\ a \in 0..min.Dom.ADR.P\text{-}1 \ni \forall\ s \in ST.P,\ s_a = c$

$\wedge \forall\ (S', s, k) \in Fsq.Tk \times Seq.VAL \times Z^+,$

$St.Ctrl(S', s, k) \in ST.P \wedge Pos.Ctrl(S', s, k) \in Dom.P$

8) $\wedge \forall\ R \in Rel, \forall\ s \in Seq.VAL,$

if $RCV.s \equiv \{\ S \mid Rcvr(S, s) = tru\} \wedge$

$EXPR.s \equiv \{\ S \mid v(S, s) \in Val \wedge \exists\ n \in Z^+ \ni \exists\ f : (RCV.s)^n \to STR.Tk \wedge S \in Rng.f\}$

then $\exists\ b : (EXPR.s)^2 \to EXPR.s$

$\ni \forall\ E1, E2 \in EXPR.s,$ if $(v(E1, s), v(E2, s)) \in R$) then $v(b(E1, E2), s) = tru$

$\wedge$ if $(v(E1, s), v(E2, s)) \notin R$) then $v(b(E1, E2), s) = fals$

The following notation focuses attention on several sets that depend on a given state s : *UNIT. s*, *RCV. s*, *COM. s*, *EXPR. s*, *BE. s*, and *TypeEquiv. s*. Their names are intended to suggest "units", "receivers", "commands", "expressions", "Boolean expressions", and "type equivalence", respectively. Intuitively: a unit is operationally correct in the state; receivers and expressions we have already characterized; a Boolean expression is one whose only values are *tru* and *fals*; and type equivalence is a relation on values. It seems difficult to characterize what "command" should mean. A first thought might be a string S of tokens for which there is a state s such that the execution of S begun in state s would result in a change of state. However, that would exclude the empty command, as well as loops and conditionals whose bodies consist of the empty command. Thus, we only specify that there must be a way to determine whether a given string of tokens in a given state is a command or not.

Also defined below is a very important function, PC_P, which is called "program control in P". "*St*" and "*Pos*", defined earlier as projections from a triple Cartesian product, are here overloaded to designate projections from a double Cartesian product as well.

2.4.3 Notation: *IPLS* denotes the class of all imperative programming language systems.

$\forall\ L \in IPLS,\ \forall\ s \in Seq.\,VAL,$

$\wedge$ if $Tk \equiv L_2 \wedge Val \equiv L_3 \wedge tru \equiv L_6 \wedge OprCorr \equiv L_8 \wedge v \equiv L_{12} \wedge Loc \equiv L_{13} \wedge Ctrl \equiv L_{14}$

then

$\mathcal{P}_L$, *ADR*, *ST*, *FVal*, *RCV. s* and *EXPR. s* are defined in Definition 2.4.2

$\wedge\ UNIT.\,s \equiv \{u \in STR.\,Tk \mid OprCorr(u,s) = tru\}$

$\wedge\ COM.\,s \equiv \{C \in STR.\,Tk \mid Com(C,s) = tru\}$

$\wedge\ BE.\,s \equiv \{b \in EXPR.\,s \mid v(b,s) \in \{tru, fals\}\}$

$\wedge\ TypeEquiv.\,s \equiv \{(u,w) \in Val \times Val \mid Compat(u,w,s) = tru\}$

$\wedge\ \forall\ P \in \mathcal{P}_L,\ PC_P$ denotes the function such that $PC_P : ST.\,P \times 1..\#P \rightarrow ST.\,P \times 1..\#P$

$\wedge\ \forall\ (s,k) \in ST.\,P \times 1..\#P,\ \ PC_P(s,k) = Ctrl(P,s,k)$

$\wedge\ \forall\ (s,k) \in ST.\,P \times 1..\#P,\ \ \wedge\ St(s,k) \equiv s \wedge Pos(s,k) \equiv k$.

2.4.4 Theorem: Every *Rel*-relationship has a Boolean expression for that relationship, and type equivalence is an equivalence relation. More precisely, let $L \in IPLS$, $P \in \mathcal{P}_L$, $Val \equiv L_3$, $tru \equiv L_6$, $R \in Rel \equiv L_3$, and $s \in ST.\,P$.

(1) $\forall\ E1, E2 \in EXPR.\,s$, if $Compat(E1,E2,s) = tru$

then $\exists\ bx \in BE.\,s \ni (\ v(bx,s) = tru \iff (v(E1,s), v(E2,s)) \in R\ )$.

(2) *TypeEquiv. s* is an equivalence relation.

Proof: See Appendix 2.

2.4.5 Definition: (types)

$$T \text{ is a type in } s \iff \begin{cases} s \in Seq.VAL \\ \wedge\ T \text{ is an equivalence class determined by } TypeEquiv.\,s \end{cases}$$

2.5 PROGRAM EXECUTION AND PROGRAM CONSTRUCTS

We now define an execution of a program segment as a sequence of pairs in the domain of the program-control function PC_P, where P is the program in which the segment appears. Intuitively, execution starts at the initial token of the segment and moves under program control either forever or until reaching either the last token of the program or the token just beyond end of the segment.

2.5.1 Definition: (executions of segments) Let $L \in IPLS$, $P \in \mathcal{P}_L$, and $S \in SEG.P$.

$$\left.\begin{array}{l} E \text{ is an} \\ \text{execution} \\ \text{of } S \end{array}\right\} \Longleftrightarrow \left\{\begin{array}{l} \exists\, n \in Z_0^+ \cup \{\infty\} \wedge \exists\, s \in ST.P \\ \ni E\colon Z_0^{n+1} \to ST.P \times 1..\#P \wedge E_0 = (s, min.Dom.S) \wedge \forall\, i \in Z_0^n,\ \ E_{i+1} = PC_P.E_i \\ \wedge\ \forall\, k \in Z^+,\ ((Pos.E_k = \#P \in Dom.S \text{ or } Pos.E_k = max.Dom.S+1) \Longleftrightarrow k+1 = \#E) \end{array}\right.$$

$$\left.\begin{array}{l} E \text{ is an execution of} \\ S \text{ that terminates} \end{array}\right\} \Longleftrightarrow \quad (E \text{ is an execution of } S \wedge \#E \in Z_2^+)$$

2.5.2 Notation: $\forall\, L \in IPLS$, $\forall\, P \in \mathcal{P}_L$, $\forall\, S \in SEG.P$,

$$Exec.P.S \equiv \{\, E \mid E \text{ is an execution of } S \}$$
$$\wedge\ EXEC.P \equiv \cup\{Exec.P.S \mid S \in SEG.P\}$$
$$\wedge\ EX_P \equiv \{(\,(S,s), E\,) \in (SEG.P \times ST.P) \times EXEC.P \mid St.E_0 = s \wedge E \in Exec.P.S\}$$

2.5.3: Theorem: Given program P and state s, there exists an execution of P beginning in s; given segment S of P, if there exists an execution of S that begins in s, then that execution is unique, so relation EX_P is actually a function. More precisely:

Let $L \in IPLS$, $P \in \mathcal{P}_L$, and $S \in SEG.P$.

(1) $\forall\, s \in ST.P,\ \exists\, E \in Exec.P.P \ni St.E_0 = s.$

(2) $\forall\, E1, E2 \in Exec.P.S$, if $St.E1_0 = St.E2_0$ then $E1 = E2$.

(3) $EX_P : Dom.EX_P \to EXEC.P$.

Proof: See Appendix 2.

Note in the following definition that, even though we choose to name delimiters so as to suggest a while-loop, there is nothing inherent in the definition to distinguish the construct being defined from an if-then-endif construct.

2.5.4 Definition: (citadels)

Let $L \in IPLS$ and $Tk \equiv L_2$.

$$\left.\begin{array}{l} W \text{ is a} \\ \\ \text{citadel} \\ \\ \text{in } L \end{array}\right\} \Longleftrightarrow \left\{\begin{array}{l} W \in STR.\,Tk \\ \wedge\ \exists\ s \in Seq.\,VAL \ni \exists\ while, do, endwhile, sp, b, C \in UNIT.\,s \\ \quad \ni \#\{while, do, endwhile, sp\} = 4 \wedge \forall\ SeqTok \in while \cup do \cup endwhile \cup sp,\ \ \#SeqTok = 1 \\ \quad \wedge\ W = while\,\hat{}\,sp\,\hat{}\,b\,\hat{}\,sp\,\hat{}\,do\,\hat{}\,sp\,\hat{}\,C\,\hat{}\,sp\,\hat{}\,endwhile \\ \quad \wedge\ \forall\ P \in \mathcal{P}_L,\ \text{if } W \cap SEG.\,P \neq \emptyset \\ \qquad\qquad \text{then } \exists\ s \in ST.\,P \ni W, C \in COM.\,s \wedge b \in BE.\,s \end{array}\right.$$

2.5.5 Notation: $\forall\ L \in IPLS$,

$$\begin{array}{l} \text{if } Tk \equiv L_2 \\ \text{then } Citad.\,L \equiv \{\ W \in STR.\,Tk \mid W \text{ is a citadel in } L\} \\ \quad \wedge\ \forall\ W \in Citad.\,L,\ \forall\ s \in Seq.\,VAL,\ \forall\ while, do, endwhile, sp, b, C \in UNIT.\,s, \\ \qquad\qquad \text{if } W = while\,\hat{}\,sp\,\hat{}\,b\,\hat{}\,sp\,\hat{}\,do\,\hat{}\,sp\,\hat{}\,C\,\hat{}\,sp\,\hat{}\,endwhile \\ \qquad\qquad \text{then } Guard.\,W \equiv b \wedge Body.\,W \equiv C. \end{array}$$

We now define a loop as a citadel with a certain kind of execution. That execution E determines a state sequence St and a position sequence Pos whose terms are the first and second elements of the sequence E of pairs. Intuitively: The term of St when Pos is at *while* is the same as the term of St when Pos is next at *do* (no side effects), and if the value of the guard in those states is false, then Pos moves to the first token beyond *endwhile* and execution terminates. Otherwise, Pos moves to the first token of the body. St is then subject to change as Pos varies through the body to *endwhile* and back to *while*; and the process is repeated. If St has no term in which the value of the guard is false when Pos is at *while*, then the execution is infinite.

In the following definition of a loop: The sequence $s^{(\cdot)}$ denotes that subsequence of the state sequence St consisting of those terms where Pos is at *while*; n denotes the number of iterations of the loop, and $n \in Z_0^+ \cup \{\infty\}$; $Sum.\,k$ denotes the number of steps in the execution of the kth iteration of the loop; and $sum.\,k$ denotes the sum of the steps in the individual parts, with $\beta(k)$ and $\gamma(k)$ denoting the number of steps in the execution of the guard b' and body C', respectively. Note that the length of each of the units *while*, *sp*, *do*, and *endwhile* is 1; so the number, $sum.\,k$, of steps in each iteration of the loop begun in state $s^{(k)}$ is the sum of

1	(for *sp*),
$\#EX_P(b', s^{(k)})$,	(for guard b)
3	(for *sp*, *do*, and *sp*),
$\#EX_P(C', s^{(k)})$,	(for body C) and
3	(for *sp*, *endwhile*, and *while*).

2.5.6 <u>Definition</u>: (while-loop commands) Let $L \in IPLS$, $W \in Citad.L$, $b \equiv Guard.W$, $C \equiv Body.W$, $tru \equiv L_6$, and $fals \equiv L_7$.

W is a loop $\iff$

$\forall P \in \wp_L,\ \forall W', b', C' \in SEG.P,$

if $W' \in W \wedge b' \in b \wedge C' \in C \wedge POS.while \equiv min.Dom.W'$

$\wedge\ POS.do \equiv POS.while{+}1{+}\#b'{+}2 \wedge POS.endwhile \equiv POS.do{+}1{+}\#C'{+}2$

then $\forall E \in Exec.P.W',$

$\exists n \in Z_0^+\cup\{\infty\} \ni \exists s^{(\cdot)} : Z_0^n \to ST.P \ni s^{(0)} = St.E_0 \wedge E = EX_P(W', s^{(0)})$

$\wedge$ if $n \in Z_0^+$ then $b \in BE.s^{(n)} \wedge v(b, s^{(n)}) = fals$

$\wedge$ if $n > 0$

then $\forall k \in Z_0^n,\ \ b \in BE.s^{(k)} \wedge W, C \in COM.s^{(k)}$

$\wedge$ if $\beta.k \equiv \#EX(b', s^{(k)}) \in Z^+ \wedge \gamma.k \equiv \#EX(C', s^{(k)}) \in Z^+$

$\wedge\ Sum.k \equiv \Sigma^{k-1}{}_{j=0}(1{+}(\beta.j){+}3{+}(\gamma.j){+}3)$

then $\forall k \in Z_0^{n-1},\ v(b, s^{(k)}) = tru \wedge E_{Sum.k} = (s^{(k)},\ POS.while\ )$

$\wedge\ E_{(Sum.k)+1} = (s^{(k)}, POS.while{+}1)$

$\wedge\ \forall i \in 1..\beta.k,\ \ E_{(Sum.k)+1+i} = EX_P(b', s^{(k)})_i$

$\wedge\ E_{(Sum.k)+1+(\beta.k)+1} = (s^{(k)}, POS.do{-}1\ )$

$\wedge\ E_{(Sum.k)+1+(\beta.k)+2} = (s^{(k)}, POS.do\ \ \ \ )$

$\wedge\ E_{(Sum.k)+1+(\beta.k)+3} = (s^{(k)}, POS.do{+}1)$

$\wedge\ \forall i \in 1..\gamma.k,\ \ E_{(Sum.k)+1+(\beta.k)+3+i} = EX_P(C', s^{(k)})_i$

$\wedge\ E_{(Sum.k)+1+(\beta.k)+3+(\gamma.k)+1} = (s^{(k+1)}, POS.endwhile{-}1\ )$

$\wedge\ E_{(Sum.k)+1+(\beta.k)+3+(\gamma.k)+2} = (s^{(k+1)}, POS.endwhile\ \ \ )$

$\wedge$ if $n \in Z^+$

then $E_{Sum.n} = (s^{(n)},\ \ POS.while\ \ )$

$\wedge\ \ E_{(Sum.n)+1} = (s^{(n)}, POS.while{+}1)$

$\wedge\ \forall i \in 1..\beta.n,\ \ E_{(Sum.n)+1+i} = EX_P(b', s^{(n)})_i$

$\wedge\ E_{(Sum.n)+1+(\beta.n)+1} = (s^{(n)}, POS.do{-}1)$

$\wedge\ E_{(Sum.n)+1+(\beta.n)+2} = (s^{(n)}, POS.do\ \ \ )$

$\wedge\ E_{(Sum.n)+1+(\beta.n)+3} = (s^{(n)}, POS.endwhile{+}1)$

$\wedge\ \#E = 1{+}(Sum.n){+}1{+}(\beta.n){+}3$

2.6 PROGRAM ASSERTIONS AND CONDITIONS

We distinguish between "assertions" and "conditions". An assertion is a Boolean expression; a condition is a set of states. For every assertion Q, there is a condition consisting of those states s for which $v(Q, s) = tru$. (The converse is not necessarily true.) While at first thought, restricting assertions to be Boolean expressions in the programming language might seem too restrictive, existential-quantifier and universal-quantifier assertions over very large, but finite, sets can be considered shorthand for multiple disjunctions and conjunctions, respectively. This tact seems more than adequate for making assertions about program code to be executed on finite machines. (All program states s might have $s_i = \infty \ \forall\ i > m$, for some m.)

2.6.1 Definition: (conditions and assertions) Let $L \in IPLS$ and $P \in \mathcal{P}_L$.

$$C \text{ is a condition of } P \Leftrightarrow C \subseteq ST.P$$

$$Q \text{ is an assertion about } P \text{ in } s \Leftrightarrow (s \in ST.P \wedge Q \in BE.s)$$

2.6.2 Notation: $\forall\ L \in IPLS,\ \forall\ P \in \mathcal{P}_L,\ \forall\ s \in ST.P,$

$$ASSERT_P.s \equiv \{Q \in STR.Tk \mid Q \text{ is an assertion about } P \text{ in } s\}$$

$\wedge\ \forall\ Q \in STR.Tk$, if $tru \equiv L_6$

$$\text{then } COND_P.Q \equiv \{s \in ST.P \mid Q \in BE.s \wedge v(Q,s) = tru\}.$$

2.6.3 Definition: (assertions true at a position) Let $L \in IPLS$, $P \in \mathcal{P}_L$, and $tru \equiv L_6$.

$$\left.\begin{array}{l} Q \text{ is always true at} \\ \text{position } k \text{ during} \\ \text{execution } E \text{ of } P \end{array}\right\} \Leftrightarrow \left\{\begin{array}{l} Q \in STR.Tk \wedge k \in Dom.P \wedge E \in Exec.P.P \\ \wedge\ \forall\ s \in ST.P,\ \forall\ i \in Dom.E,\ \text{ if } E_i = (s,k) \text{ then } s \in COND_P.Q \end{array}\right.$$

3.0 TRANSLATION

It only remains to indicate how the appealing operational language in the statement and proof of Theorem 1.1.1 can be translated into set-theoretic terms. Phrases that translate directly from the definitions in Section 2 are omitted here, e.g., "execution E terminates" means "$\#E \in Z_2^+$".

The context of the statement of Theorem 1.1.1 is that of "any execution of" a loop in some program, so we can assume that we have a program P in which the loop W appears. Thus, "during any execution of W", means "$\forall\ E \in Exec.P.W'$", where $W' \in W \cap SEG.P$; and we are considering an arbitrary execution E of W'. The loop W of the theorem fits the designation of a citadel in Section 2 provided we make the following identifications.

The loop W in the theorem	The loop W in Section 2
while	*while*
<guard>	*b*
do begin	*do*
<body>	*C*
end	*endwhile*

Note that, in order to make this identification, we have to consider the 8-character string 'do begin' as a string of length 1 (i.e., an equivalence class of singleton sequences having the form $\{(k, \textbf{do begin})\}$, where $k \in Z_0^+$) -- because *do* is defined in our development as a string of length 1, i.e., equivalence class of sequences of tokens, each such sequence of tokens being a singleton set. Other spaces and carriage returns appearing in the loop of the theorem are identified with "*sp*" in our development. We assume that the W of our development fits the definition of a loop as given by Definition 2.5.6.

3.1 THE STATEMENT OF THEOREM 1.1.1

The meaning of the various phrases used in the statement of Theorem 1.1.1 are indicated as follows.

- "an expression that is a function of the program variables" is a member of

  $\cap\{\ EXPR.s^{(k)}\ \mid\ k \in Dom.s^{(\cdot)}\}$.

- "a statement about current values of the program variables" is a member of

  $\cap\{\ ASSERT_P.s^{(k)}\ \mid\ k \in Dom.s^{(\cdot)}\}$.

As indicated in these two interpretations, we consider $s^{(\cdot)}$ to represent the sequence of states with domain Z_0^n given in the definition of a loop in Definition 2.5.6, and we let W', b', and C' be members of $SEG.P$ such that $W' \in W$, $b' \in b$, and $C' \in C$.

Note that $E = EX_P(W', s^{(0)})$.

- "<guard> and <body> terminate" means

  "$\forall\ k \in Dom.s^{(\cdot)},\ \ \#EX_P(b', s^{(k)}), \#EX_P(C', s^{(k)}) \in Z_2^+$ ".

- "no program variable changes value as a result of any evaluation of <guard>" means

  "$\forall\ k \in Dom.s^{(\cdot)}$, if $EE = EX_P(b', s^{(k)})$ then $St.EE_{min.\,Dom.\,EE} = St.EE_{max.\,Dom.\,EE}$".

- For each statement S, "S is true each time program control reaches <guard>" means

  "$\forall\ k \in Dom.s^{(\cdot)},\ \ S \in ASSERT_P.s^{(k)} \wedge s^{(k)} \in COND_P.S$".

- "*Expr* is an expression that is a function of the program variables" means

  "$Expr \in \cap\{\ EXPR.ss\ \mid\ \exists\ k \in Dom.s^{(\cdot)} \ni s^{(k)} = ss\}$".

- " '$Expr \le c$ ' is true each time program control reaches <guard>" means

  $$\text{"} \le\ \subseteq Val \times Val \wedge c \in Val$$
  $$\wedge\ \forall\ k \in Dom.s^{(\cdot)},$$
  $$v(Expr, s^{(k)}) \in Val \wedge Compat(Expr, c, s^{(k)}) = tru \wedge Expr \in EXPR.s^{(k)}$$
  $$\wedge\ 'Expr \le c' \in\ ASSERT_P.s^{(k)} \wedge s^{(k)} \in COND_P.'Expr \le c' \text{ ".}$$

- "whenever program control reaches line 2, if v_0 denotes the value of *Expr* at that time, then the value of *Expr* is at least v_0+1 the next time program control reaches line 3" means

  "$\forall\ t \in Dom.E$,

  if $Pos.E_t = min.Dom.C'-1 \wedge v(Expr, St.E_t) = v_0$

  $\wedge\ tt = min\{i \in Dom.E \mid i > t \wedge Pos.E_i = max.Dom.C'+1\}$

  then $v(Expr, St.E_{tt}) \ge t_0+1$".

3.2 THE PROOF OF THEOREM 1.1.1, PART (1)

Some of the phrases used in the proof of Theorem 1.1.1 have already been translated above. Other phrases not yet translated that are used in the proof of Part (1) are indicated as follows. (The sequences β, γ, and *Sum* used below are those with the same name designated in Definition 2.5.6 of a loop.)

- "program control must reach line 3 subsequent to any time during E that it reaches line 2" means

  "$\forall\ k \in Dom.s^{(\cdot)}$, if $(sum.k)+1+(\beta.k)+3 \in Dom.E$ then $(sum.k)+1+(\beta.k)+3+(\gamma.k)+1 \in Dom.E$"
- "the value of *Expr* must increase by at least 1 each time <body> is executed" means

  "$\forall\ k \in Dom.s^{(\cdot)}$, if $v(\text{<guard>}, s^{(k)}) = tru$ then $v(Expr, s^{(k+1)}) \geq 1 + v(Expr, s^{(k)})$"
- "*Expr* has some value v_0 the first time program control reaches <guard> during E" means

  "$\exists\ v_0 \in Val \ni Compat(Expr, v_0, s^{(0)}) = tru \wedge v(Expr, s^{(0)}) = v_0$".
- "<body> can only be executed finitely many times during E" means

  "$\exists\ m \in Z_0^+ \ni$ if $(sum.m)+1+(\beta.m)+3 \in Dom.E$ then $Pos.E_{(sum.m)+1+(\beta.m)+3} \neq min.Dom.C'-1$"
- "the value of *Expr* [increases] beyond c during some execution of <body>" means

  "$\exists\ k \in Dom\ s^{(\cdot)} \ni v(Expr, s^{(k+1)}) > c$"
- "<guard> can only be executed finitely many times [during E]" means

  "$\exists\ m \in Z_0^+ \ni (sum.m)+1+(\beta.m)+4 \notin Dom.E$

3.3 THE PROOF OF THEOREM 1.1.1, PART (2)

The meaning of the various phrases used in the proof of part (2) of Theorem 1.1.1, not already translated above, are indicated as follows.

- "there is a last time t that program control reaches guard [during E]" means

  "$\exists\ t \in Dom.E \ni Pos.E_t = min.b' \wedge t = max\{i \in Dom.E \mid Pos.E_i = min.Dom.b'\}$".
- "*Inv* is true at time t [which is the last time program control reaches <guard> during E]" and " '**not** <guard>' is true at time t" and "*Post* is true at time t" means

  "$v(Inv, s^{(n)}) = tru$" and "$v(\text{<guard>}, s^{(n)}) = fals$" and "$v(Post, s^{(n)}) = tru$".
- "*Post* must be true when program control reaches line 4" means

  "$v(Post, s^{(n)}) = tru \wedge St.E_{max.Dom.E} = s^{(n)}$".

This completes the translation.

4.0 SUMMARY

The goal of this paper was to legitimize the use of the appealing operational language used in the statement and proof of Theorem 1.1.1 by defining "program control" and other operational concepts in terms of set theory. There were two key elements in the development of these definitions. One was a distinction that is not often made, and the other was a unifying use of a definition not often insisted upon. The distinction was between sequences and strings, with strings defined as equivalence classes of sequences. The unifying use was of the definition of a sequence as a function, a function as a relation, and a relation as a set. Since a sequence is a set, set operations can be performed on a sequence, e.g., taking its cardinality, taking its union with other sequences, using a sequence as the domain of a function, etc.

Once the definition of an imperative programming language system was developed, all of the familiar terms used in discussing programs and programming could be defined as sets. In particular, all of the language used in Theorem 1.1.1 and its proof could be defined in set-theoretic terms. The two most prominent such terms were "program control" and "execution of a program segment". Program control in a particular program turned out to be a function mapping each pair consisting of a position (in the program) and a state into a (possibly different) pair also consisting of a position and state. Execution of a segment of a program turned out to be a sequence of terms in the domain of program control for that program, with a terminating execution being such a sequence that is finite.

Finally, each of the questionable words and phrases used in the statement and proof of fundamental Theorem 1.1.1 was translated into the precise set-theoretic terms defined in Section 2.

1. Dijkstra, Edsger W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18 (August 1975), 453-457.

2. Gries, David. *The Science of Programming.* Second edition. Springer-Verlag, New York, 1987.

3. Gries, David and Wadkins, Jeff. An introduction to proofs of program correctness for teachers of college-level introductory programming courses. TR 90-1102. Department of Computer Science, Cornell University. March 1990.

4. Warren, Alexander Z. Letter and personal communication. September 1988.

APPENDIX 1

A MINIMAL EXAMPLE DEMONSTRATING THE CONSISTENCY OF THE DEFINITION OF IMPERATIVE PROGRAMMING LANGUAGE SYSTEMS

We employ the standard notation of single quotes delimiting terms of sequences to represent strings, e.g., 'β' denotes the string of sequences each of which has length 1 and whose only term is β, etc. We use tuple notation for defining sequences, indicating with an initial subscript what the minimum of the domain of the sequence is, e.g., we use "$S = {}_1(a,b,c)$" as meaning "$S_1 = a \wedge S_2 = b \wedge S_3 = c \wedge \#S = 3$", and we use "$S = {}_0(a,b,c)$" as meaning "$S_0 = a \wedge S_1 = b \wedge S_2 = c \wedge \#S = 3$".

SYNOPSIS OF THE EXAMPLE

Four tokens, built from sets of integers, form the set *Tk* of all tokens. Two of them are used to define *start* and *stop*. Another token, α, is used to define the string-of-tokens delimiter *sp*, which has length 1. The remaining token is used as the term for constant sequences of various lengths to form notable strings, two of which are *tru* and *fals*, which are also used as the only elements of *Val*.

Two more notable strings, *x* and *y*, form a set *Var*, intended to suggest "variable". The only other notable strings are *gets* and *Eq*. String *gets* is used in the definition of two commands (each suggestive of an assignment statement), and "*gets*" is intended to suggest "gets the value of". String *Eq* is used as the middle unit in strings consisting of the concatenation of other units -- strings intended to suggest a statement of a relation between the first and last unit. A single relation, *ID* (the identity function on *Val*), is defined and *Rel* is defined to be the singleton set containing *ID*.

The only three commands in the system are the empty command, an assignment of *y* to *x*, and an assignment of *x* to *y*. The only three possible programs in the system use these commands, respectively, as their only components other than *start* and *stop*. All three programs have the same address space $Z_2{}^+$, all store value *tru* at address 0 and value *fals* at address 1, and all use location 2 to store values of receiver *x* and all use location 3 to store values of receiver *y*. The definitions of the seven required functions are designed to satisfy the eight requirements. The function *Compat* is defined in such a way that *Val* forms the only type in the system.

DEFINITION OF THE EXAMPLE

Define: $\infty \equiv \emptyset \ \wedge start \equiv Z^+ \wedge stop \equiv Z \wedge \alpha \equiv \{Z^+\} \wedge \beta \equiv \{Z\} \wedge Tk \equiv \{start, stop, \alpha, \beta\}$.

Note A3.1: $start, stop \in Tk \wedge start \neq stop \wedge STR.\,Tk = \{ \ [t] \mid t \in \cup(Tk^{m..n}) \ | m,n \in Z_0{}^+\}$.

Define: $sp \equiv \text{'}\alpha\text{'} \wedge tru \equiv \text{'}\beta\text{'} \wedge fals \equiv tru^\frown\text{'}\beta\text{'} \wedge x \equiv fals^\frown\text{'}\beta\text{'} \wedge gets \equiv x^\frown\text{'}\beta\text{'} \wedge y \equiv gets^\frown\text{'}\beta\text{'} \wedge Eq \equiv y^\frown\text{'}\beta\text{'}$

$\wedge\ Val \equiv \{tru, false\} \wedge VAL \equiv Val \cup \{\infty\} \wedge Var \equiv \{x,y\} \wedge ID \equiv \{(tru,tru), (fals,fals)\} \wedge Rel \equiv \{ID\}$.

Note A3.2: $tru, fals \in Val \wedge tru \neq fals \wedge \infty \notin Val \wedge Val \subseteq STR.Tk \wedge VAL = Val \cup \{\infty\} \wedge Rel \subseteq 2^{Val \times Val}$
$\wedge\ Seq.VAL = \{ \ s \mid s : Z_0{}^+ \to \{tru, fals, \infty\} \ \}$

Define: $P^{(1)} \equiv {}_1(start, \alpha, stop) \wedge P^{(2)} \equiv {}_1(start, \alpha, x, \alpha, gets, \alpha, y, \alpha, stop)$
$\wedge P^{(3)} \equiv {}_1(start, \alpha, y, \alpha, gets, \alpha, x, \alpha, stop)$

$\wedge \wp \equiv \{P^{(1)}, P^{(2)}, P^{(3)}\} \wedge ST\wp \equiv \{s \in Seq.VAL \mid s_0 = tru \wedge s_1 = fals\}$
$\wedge \forall i \in 1..3,\ ST.P^{(i)} \equiv ST\wp \wedge ADR.P^{(i)} \equiv Z_2^+ \wedge FVal.P^{(i)} \equiv Val$

$\wedge E \equiv \{S1 \hat{}\, sp \hat{}\, Eq \hat{}\, sp \hat{}\, S2 \mid S1, S2 \in STR.Tk\} \wedge COM \equiv \{sp, \text{'}x\alpha gets\alpha y\text{'}, \text{'}y\alpha gets\alpha x\text{'}\}$

Define f, g: $ST\wp \to VAL$ by: $\forall\ s \equiv {}_0(tru, fals, s_2, s_3, s_4, \ldots) \in ST.\wp,\ f(s) \equiv {}_0(tru, fals, s_3, s_3, s_4, \ldots)$
$\wedge\ g(s) \equiv {}_0(tru, fals, s_2, s_2, s_4, \ldots)$.

Define $Ctrl$: $Fsq.Tk \times Seq.VAL \times Z^+ \to Fsq.Tk \times Seq.VAL \times Z^+$ by:

$\forall\ (S', s, k) \in Fsq.Tk \times Seq.VAL \times Z^+$,

if $S' \notin \wp$ or $s \notin ST.\wp$ then $Ctrl(S', s, k) \equiv (S', s, k)$

$\wedge$ if $S' \in \wp \wedge s \in ST.\wp \wedge S'_k = stop$ then $Ctrl(S', s, k) \equiv (S', s, k)$

$\wedge$ if $S' \in \wp \wedge s \in ST.\wp \wedge ($ $S'_k \in \{start, \alpha\}$
or $(S' = P^{(1)} \wedge k \in 1..2)$
or $(S' \in \{P^{(2)}, P^{(3)}\} \wedge k \in 1..6 \cup \{8\})$
$)$
then $Ctrl(S', s, k) \equiv (S', s, k+1)$

$\wedge$ if $S' \in \wp \wedge s \in ST.\wp \wedge S' \in \{P^{(2)}, P^{(3)}\} \wedge k = 7$

$$\text{then } Ctrl(S', s, k) \equiv \begin{cases} (S', f(s), 8) \text{ if } S' = P^{(2)} \\ (S', g(s), 8) \text{ if } S' = P^{(3)} \end{cases}$$

$\wedge$ otherwise, $Ctrl(S', s, k) \equiv (S', s, k)$.

Define $Com, Rcvr, OprCorr : STR.Tk \times Seq.VAL \to \{tru, fals\}$ and
define Loc: $STR.Tk \times Seq.VAL \to Z_0^+ \cup \{\infty\}$ and
define v: $STR.Tk \times Seq.VAL \to \{tru, fals\} \cup \{\infty\}$ by:

$\forall\ (S, s) \in STR.Tk \times Seq.Val$,

$$Com(S, s) \equiv \begin{cases} tru & \text{if } S \in COM \\ fals & \text{otherwise} \end{cases}$$

$$\wedge \quad Rcvr(S, s) \equiv \begin{cases} tru & \text{if } S \in Var \\ fals & \text{otherwise} \end{cases}$$

$$\wedge\ OprCorr(S,s) \equiv \begin{cases} tru & \text{if } S \in Val \cup Var \cup COM \cup \{\text{'start'}, \text{'stop'}\} \cup \wp \cup E \\ fals & \text{otherwise} \end{cases}$$

$$\wedge\ Loc(S,s) \equiv \begin{cases} 2 & \text{if } S = x \\ 3 & \text{if } S = y \\ \infty & \text{otherwise.} \end{cases}$$

$\wedge$ if $S \notin Val \cup Var \cup E$ or $s \notin ST\wp$ then $v(S,s) \equiv \infty$
$\wedge$ if $S \in Val \cup Var \cup E\ \wedge\ s \in ST\wp$ then

$$v(S,s) \equiv \begin{cases} S & \text{if } S \in Val \\ s_2 & \text{if } S = x \\ s_3 & \text{if } S = y \\ tru & \text{if } \exists\ S1, S2 \in STR.Tk \ni S = S1 \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ S2 \wedge v(S1,s) = v(S2,s) \\ fals & \text{if } \exists\ S1, S2 \in STR.Tk \ni S = S1 \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ S2 \wedge v(S1,s) \neq v(S2,s) \end{cases}$$

Define *Compat*: $STR.Tk \times STR.Tk \times Seq.VAL \rightarrow \{tru, fals\}$ by:

$$\forall\ (S1, S2, s) \in STR.Tk \times STR.Tk \times Seq.VAL,\quad Compat(S1,S2,s) \equiv \begin{cases} tru & \text{if } v(S1,s),\ v(S2,s) \in Val \\ fals & \text{otherwise.} \end{cases}$$

Define: $L \equiv (Tk, Val, Rel, start, stop, tru, fals, OprCorr, Com, Compat, Rcvr, v, Loc, Ctrl)$

$\wedge\ \wp_L \equiv \wp$.

Requirement (1) follows from Notes A3.1, A3.2.

Requirement (2) follows from the definitions of *OprCorr* and *v*.

Requirement (3) follows from the definitions of *Com* and *OprCorr*.

Requirement (4) follows from the definitions of *Rcvr*, *OprCorr*, *Loc*, and *v*.

Requirement (5) follows from the definitions of *Compat* and *OprCorr*.

For requirement (6): Let $c \in Val$ and define $S = x$ and $s = {}_0(tru, fals, c, c, c, \ldots)$, i.e, $s_i = c\ \forall\ i \in Z_2^+$.

The result then follows from the definitions of *Rcvr*, *Compat*, *v*, and *Loc*.

Requirement (7) follows from the definitions of functions *ADR*, *ST*, and *FVal* -- all with domain $\wp_L = \wp$, and from the definition of the function *Ctrl*.

For requirement (8): Let $R \in Rel$ and $s \in Seq.VAL$. Then $R = ID = \{(tru, tru),\ (fals, fals)\}$.

Define function *b* as follows. Let $E1, E2 \in EXPR.s$. Then $v(E1,s)$, $v(E1,s) \in Val$, and:

$$\exists\ m, k \in Z^+ \ni \exists\ f1 : (RCV.s)^m \rightarrow STR.Tk \wedge \exists\ f2 : (RCV.s)^k \rightarrow STR.Tk$$

$$\ni E1 \in Rng.f1 \wedge E2 \in Rng.f2,$$

so $\exists\ r1 : 1..m \rightarrow RCV.s \wedge \exists\ r2 : 1..k \rightarrow RCV.s$

$$\ni f1(r1_1,...,r1_m) = E1 \wedge f2(r2_1,...,r2_k) = E2.$$

Thus, define $n \equiv m+k$ and define $f: (RCV.s)^n \rightarrow STR.Tk$ by

$$\forall\ r : 1..n \rightarrow RCV.s,\ \ f(r_1,...,r_n) \equiv f1(r_1,...,r_m) \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ f2(r_{m+1},...,r_n),$$

and define $b(E1,E2) = f(r1_1,...,r1_m, r2_1,...,r2_k)$.

Then $b(E1,E2) = f1(r1_1,...,r1_m) \hat{}\ Eq \hat{}\ f2(r2_1,...,r2_k) = E1 \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ E2 \in E$, by definition of E.*

Since $v(E1,s)$, $v(E1,s) \in Val$, $v(E1 \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ E2, s) \in Val$ by definition of v. Thus $E1 \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ E2 \in EXPR.s$, which shows that $Rng.r \subseteq EXPR.s$, so $b : (EXPR.s)^2 \rightarrow EXPR.s$.

To prove the two desired implications, let $E1, E2 \in EXPR.s$ and $(v(E1,s), v(E2,s)) \in R = ID$. Then

$v(b(E1,E2), s) = v(E1 \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ E2, s)$ by definition of b; and $E1 \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ E2 \in E$ by definition of E, so

$= tru$ by definition of v,

which completes the proof of the first implication. The proof of the second implication is completely parallel, and it is omitted.

* Note that string $b(E1,E2) = E1 \hat{}\ sp \hat{}\ Eq \hat{}\ sp \hat{}\ E2$ is well defined, even though $f1, f2, r1, r2$ are not necessarily unique, i.e., there might exist $f1', f2', r1', r2'$ with the same properties WITHOUT $f1' = f1$, $f2' = f2$, etc.

APPENDIX 2

PROOF OF THEOREMS 2.4.4 AND 2.5.3

Proof of Theorem 2.4.4:

Proof of Part (1): Let $s \in ST.P$, $E1, E2 \in EXPR.s$, and $Compat(E1, E2, s) = tru$. By property (8) of Definition 2.4.2,

(*) $\exists\ b : (EXPR.s)^2 \rightarrow EXPR.s$

$\ni \forall\ EE1, EE2 \in EXPR.s$, if $(v(EE1, s), v(EE2, s)) \in R$) then $v(b(EE1, EE2), s) = tru$

$\wedge$ if $(v(EE1, s), v(EE2, s)) \notin R$) then $v(b(EE1, EE2), s) = fals$

Define $bx = b(E1, E2)$. We first establish the desired equivalence and then show that $bx \in BE.s$.

If $v(bx, s) = tru$ then $v(b(E1, E2), s) = tru$ so "$(v(E1, s), v(E2, s)) \notin R$" cannot be true, because by line (*) that would imply $v(b(E1, E2), s) = fals$, so tru would equal $fals$, which would be a contradiction. This proves "$\Rightarrow$" of the desired equivalence.

If $(v(E1, s), v(E2, s)) \in R$ then, by line (*), $v(b(E1, E2), s) = tru$, i.e., $v(bx, s) = tru$, which proves the other desired implication, and the desired equivalence is established.

It only remains to show that $bx \in BE.s$. Since $bx = b(E1, E2) \in Rng.b \subseteq EXPR.s$, $bx \in EXPR.s$ so we need only show that $v(bx, s) \in \{tru, fals\}$. But this follows from an argument parallel to the equivalence argument just given, because either "$(v(E1, s), v(E2, s)) \in R$" is true or not, so there are only two cases to consider. In case this is true, $v(bx, s) = tru$, and in case it is false, $v(bx, s) = fals$. Since these are the only two possible cases, $v(bx, s) \in \{tru, fals\}$ in any case. Thus $bx \in BE.s$, which completes the proof of Part (1).

Proof of Part (2): Let $s \in Seq.VAL$. By the first conjunct of requirement (5) of Definition 2.4.2,

$\forall\ c \in Val,\ Compat(c, c, s) = tru,$

so $(c, c) \in TypeEquiv.s$, which proves that $TypeEquiv.s$ is reflexive.

Let $(c, d) \in TypeEquiv.s$. Then $Compat(c, d, s) = tru$, so by the second conjunct of requirement (5),

$Compat(d, c, s) = tru$, i.e., $(d, c) \in TypeEquiv.s$, which proves that *TypeEquiv.s* is symmetric.

Let $(c, d), (d, e) \in \mathit{TypeEquiv.s}$. Then

$$\mathit{Compat}(c, d, s) = \mathit{tru} = \mathit{Compat}(d, e, s) = \mathit{Compat}(e, d, s)$$

by both the definition of *TypeEquiv.s* and the second conjunct of requirement (5). By the third conjunct,

$$\mathit{Compat}(c, e, s) = \mathit{tru}, \text{ i.e., } (c, e) \in \mathit{TypeEquiv.s},$$

which proves that *TypeEquiv.s* is transitive, which completes the proof of Part (2).

<u>Proof of Theorem 2.5.3</u>:

<u>Proof of Part (1)</u>: $\forall\ s \in ST.P$, define E inductively by defining

$$E_0 = (s,1)$$

$$\wedge\ \forall\ i \in Z_0{}^+, \text{ if } E_i \text{ is defined and } Pos.E_i \neq \#P$$

$$\text{then define } E_{i+1} = PC_P\,.\,E_i\,.$$

Then $E \in \mathit{Exec}.P.P$ and $St.E_0 = s$, which is what we wanted to prove.

<u>Proof of Part (2)</u>: Let $E1, E2 \in \mathit{Exec}.P.S$ and $St.E1_0 = St.E2_0$. The proof that $E1 = E2$ is by induction. Define K by:

$$k \in K \iff \begin{cases} k \in Z_0{}^+ \\ \wedge \text{ if } \forall\ i \in 0..k,\ \ i \in (\mathit{Dom}.E1) \cup (\mathit{Dom}.E2) \\ \text{then } \forall\ i \in 0..k,\ \ i \in (\mathit{Dom}.E1) \cap (\mathit{Dom}.E2) \wedge E1_i = E2_i \end{cases}$$

$0 \in K$ because: $0 \in Z_0{}^+$ and $E1_0 = (St.E1_0, \mathit{min}.\mathit{Dom}.S) = (St.E2_0, \mathit{min}.\mathit{Dom}.S) = E2_0$.

Let $k \in K$ and we must prove that $k+1 \in K$. Then $k \in Z_0{}^+$, which implies $k+1 \in Z^+$, so assume

$$\forall\ i \in 0..k+1,\ \ i \in (\mathit{Dom}.E1) \cup (\mathit{Dom}.E2)$$

and we must prove that $\forall\ i \in 0..k+1$, $i \in (\mathit{Dom}.E1) \cap (\mathit{Dom}.E2) \wedge E1_i = E2_i$. Thus we

let $i \in 0..k+1$

and we must show that $i \in (Dom.E1) \cap (Dom.E2) \wedge E1_i = E2_i$. If $i \in 0..k$ then the desired result holds because $k \in K$, so we need only consider the case where $i = k+1$.

Case I. $k+1 \in Dom.E1$. Then $k \in Dom.E1$ so, since $k \in K$, $k \in (Dom.E1) \cap (Dom.E2) \wedge E1_k = E2_k$, so

(*) $PC_P.E1_k = PC_P.E2_k$ because PC_P is a function.

If we can show that $k+1$ also belongs to $Dom.E2$, line (*) will convert to $E1_{k+1} = E2_{k+1}$.

Suppose $k+1 \notin Dom.E2$. Then $k = \#E2$, so by definition of execution of S,

$\exists\ kk \in Z_0^+ \ni k = kk+1$, so $kk \in K$ and $E1_{kk} = E2_{kk}$ and

$Pos.E2_{kk} = \#P$ or $Pos.E2_{kk} = max.Dom.S+1$, i.e.,

$Pos.E1_{kk} = \#P$ or $Pos.E1_{kk} = max.Dom.S+1$, because $E1_{kk} = E2_{kk}$ --

and in either case $kk+1$ cannot be a member of $Dom.E1$, i.e., k cannot be a member of $Dom.E1$. This contradicts the fact that $k \in K$ (which implied that $k \in (Dom.E1) \cap (Dom.E2)$, as observed earlier). Thus our supposition is false,

so $k+1 \in Dom.E2$. Thus by (*), $E1_{k+1} = E2_{k+1}$, which completes the argument in this case that $k+1 \in K$.

Case II. $k+1 \in Dom.E2$. The argument here that $k+1 \in K$ is completely parallel to that of Case I, and it is omitted.

Thus $k+1 \in K$ in both cases, which completes the proof, by the induction principle, that $K = Z_0^+$. Thus, by definition of K, $E1_k = E2_k\ \forall\ k \in (Dom.E1) \cup (Dom.E2)$, i.e., $E1 = E2$.

Proof of Part (3): We must prove that any "two" members of EX_P that have the same first element must have the same second element as well, so let $(\ (S,s),\ E1\ ),\ (\ (S,s),\ E2\ ) \in EX_P$, and we wish to prove that $E1 = E2$. Then by definition of EX_P, $E1, E2 \in Exec.P.S$ and $St.E1_0 = s = St.E2_0$, which satisfies the hypothesis of (2), so the conclusion holds, namely, $E1 = E2$. q.e.d.