Programming as a Mathematical Exercise [and Discussion]

J. R. Abrial, J. C. Shepherdson, J. S. Hillmore and R. L. Constable

Phil. Trans. R. Soc. Lond. A 1984 **312**, 447-473
doi: 10.1098/rsta.1984.0070

Phil. Trans. R. Soc. Lond. A **312**, 447–473 (1984) [447]
Printed in Great Britain

Programming as a mathematical exercise

By J. R. Abrial
26 *rue des Plantes*, 75014 *Paris*, *France*

This paper contains a formal framework within which logic, set theory and programming are presented together.

These elements can be presented together because, in this work, we no longer regard a (procedural) programming notation (such as PASCAL) as a notation for expressing a computation; rather, we regard it as a mere extension to the conventional language of logic and set theory. The extension constitutes a convenient (economical) way of expressing certain relational statements.

A consequence of this point of view is that the activity of program construction is transformed into that of proof construction.

To ensure that this activity of proof construction can be given a sound mechanizable foundation, we present a number of theories in the form of some basic deduction and definition rules. For instance, such theories compose the two logical calculi, a weaker version of the standard Zermelo–Fraenkel set theory, as well as some other elementary mathematical theories leading up to the construction of natural numbers. This last theory acts as a paradigm for the construction of other types such as sequences or trees. Parallel to these mathematical constructions we axiomatize a certain programming notation by giving equivalents to its basic constructs within logic and set theory. A number of other non-logical theories are also presented, which allows us to completely mechanize the calculus of proof that is implied by this framework.

0. Introduction

In the past few years, specification of computer programs has become the subject of intense activity among computer scientists; numerous specification languages and similar methods have been proposed to 'solve this problem' (see, for example, Jones (1980)). After some years of activity in this area, we have now reached the conclusion that the problem is not so much that of stating (specifying) what a computer program is supposed to do; but rather studying the *mathematical framework* within which each program is supposed to behave. For instance, the specification of a sorting program is not sufficient to construct one; it is certainly necessary to also study certain mathematical properties enjoyed by sorted sequences and by permuted ones.

Experience shows that the number of logical and mathematical tools to be used in the construction of most computer programs is quite small. In fact, such tools are essentially made up of the two logical calculi, the elementary mathematical entities (sets, relations, functions) and, finally, the first mathematically constructed types (finite subsets, natural numbers, finite sequences, finite trees). As can be seen, these are in no way comparable to what needs to be mastered by the professional mathematician.

However, once the mathematical framework in question has been established, and once the program has been formally specified within this framework, the development process by which a corresponding real program can be reached is by no means a trivial task (Jones 1980; Bjørner & Jones 1982). This is the reason why an increasing number of people think, as we do, that the gap must be bridged between a program specification (that is, a declarative statement expressing a *relation* between a program input and some output) and a program content (that is, an algorithmic statement expressing the transformation by finite means of a program input into some output). To do so, it has been proposed to make specifications possibly computable (R. Kowalski and D. A. Turner, this symposium). Compared with Professor Hoare's contribution to this symposium, what we envisage in this paper is slightly different. *We propose to make programs not necessarily executable.* In other words, we no longer regard a programming language as a notation for expressing a computation (however abstract the corresponding computer might be); rather, we regard it as a notation for expressing a relation. That some of these relations are indeed computable is then simply a happy accident.

Of course, all specifications (that is, all logical and set-theoretic relational statements) cannot fit into this 'programming' notation. This is the reason why the activity of program construction has not disappeared; it has even, we think, been clarified in that it now consists of transforming certain formal statements into others, related to the former by equivalence or implication. Such transformations can be done owing to the mathematical framework established, at the initial stage, together with the first specification of the program. In other words, and to summarize at this point, the activity of program construction has been transformed into that of *proof construction*, to be done in the realm of logic and set theory.

This paper is intended to present a *unified theoretical basis* within which this activity of 'program' construction can be developed. As already stated, this basis is essentially made up of logic and set theory together with a 'programming' notation, the constructs of which have equivalent counterparts in logic and set theory. The purpose of such a formal basis is, among others, to ensure that the activity of 'program' construction, that is, again, that of proof construction, can be *mechanically aided*. To do so, we shall also present a number of small extra theories, the role of which is to formalize what is usually written, in the form of English comments besides formal statements (i.e. that certain variables are not free in a formula, that all variables of a declaration are distinct from each other, that some expression is to be substituted for some variable in a formula, etc.). Among these small theories, one is of particular importance: the theory of variable declarations (also called schema (Morgan & Sufrin 1984)), which allows simplification and unification of set theory and programming.

The theory of sets presented here is, in a sense, weaker than the standard set theory of Zermelo and Fraenkel (herinafter called Z.F.); a noticeable difference lies in the absence of the axiom of pairing (there exists a set, the members of which are two given sets) and in that of the axiom of union (there exists a set, the members of which are the members of the members of a given set). As a consequence, the concept of ordered pair cannot be defined by using the traditional trick of Kuratowski, nor can the construction of natural numbers be done, as usual, by a cumulative hierarchy. The reason for these limitations is our intent to give to each formalized object a *type*, constructed from more elementary types through the only operations of Cartesian product, power set, set comprehension and set-theoretic fixpoint.

In fact, the theory of types presented here is just a part of set theory. It starts with the (Whitehead) constructive (fixpoint) definition of the set of finite subsets of a set, which allows one to define eventually the concept of infinity (an infinite set is one that is not a member of the set of its finite subsets); we then postulate (as in Z.F.) the existence of an infinite set. The corresponding axiom, however, does not take an existential form; rather it is expressed through a *symbol*, 'U', denoting the infinite set in question. This set constitutes, together with the empty

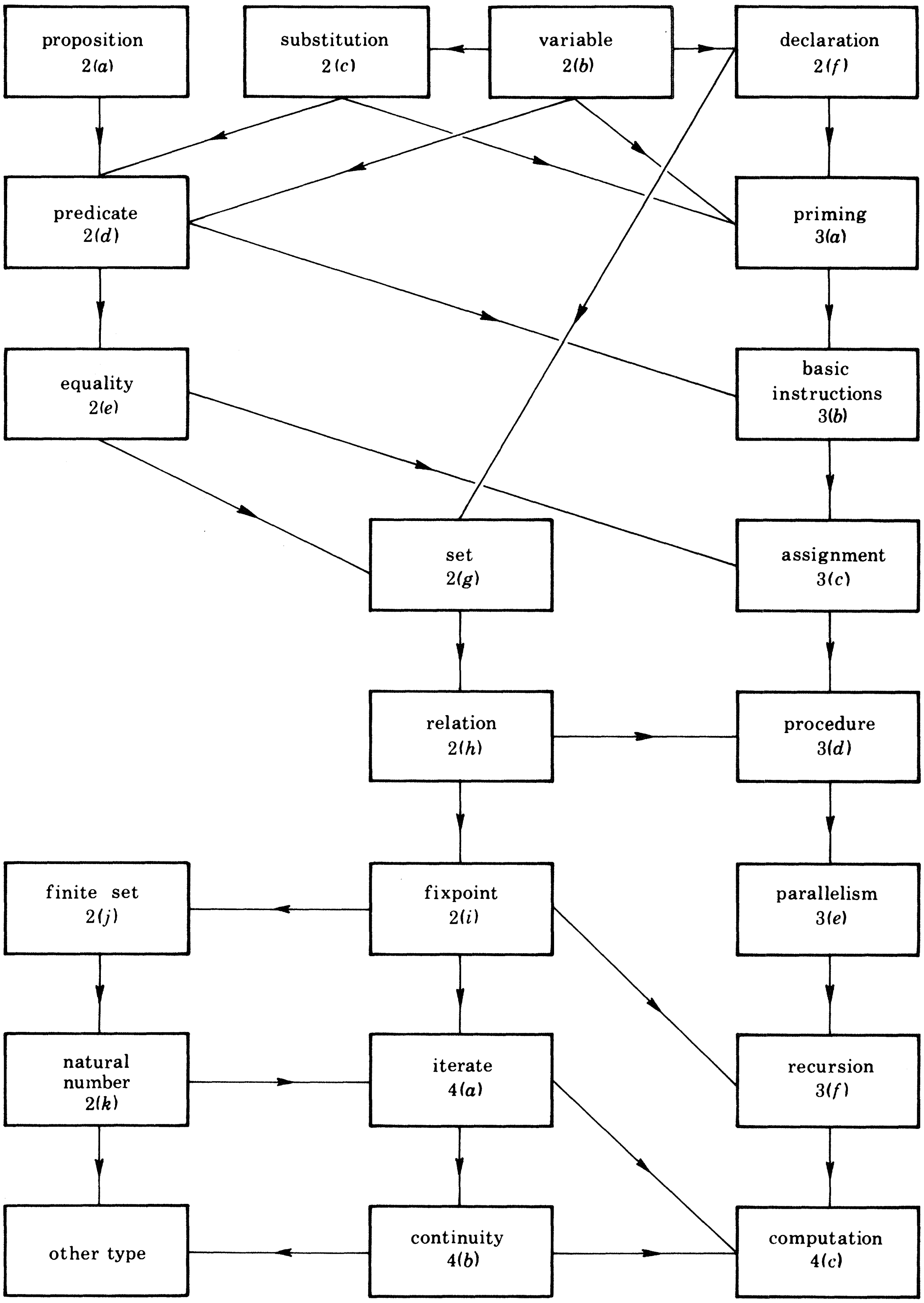

FIGURE 1. Relations between the theories.

set, our only given set. We then construct (again, as a fixpoint) the set of natural numbers as a certain subset of the power set of 'U'. From this point on, other types such as finite sequences and finite trees are easily definable.

Parallel to this presentation of logic and set theory, we define the first constructs of the 'programming' notation (namely guarding, sequencing, alternation, and local variables) (Dijkstra 1975). All these constructs are expressed solely in terms of logic (i.e. propositional calculus and predicate calculus). The 'programming language' is then extended to include an assignment and a '**skip**' (no-op) operation. These, of course, are expressed in terms of equality. A further extension deals with more 'advanced' concepts such as procedures, procedure calls (with various modes of parameter passing), parallelism and recursion. These are expressed in terms of set theory.

As will be seen later, we have carefully and intentionally avoided the possibility of writing spurious 'programs'. For instance, procedures cannot have global variables (although parameters can be passed to them 'by reference'). This precludes the 'dangerous' phenomenon of aliasing, by which the same programming variable can be reached through different paths.

It should be noted that the definition of this 'programming' notation in terms of logic and set theory *does not constitute a form of denotational semantics* (Scott & Stratchey 1970). Again, we should bear in mind that this notation is just a convenient (i.e. economical) way of formally writing certain statements of logic and set theory. This is precisely the reason that we feel it so important to clarify (and, perhaps, simplify) the definition of set theory.

Figure 1 summarizes the relations that exist between the various theories we shall present in later sections.

The remainder of this paper comprises four sections. In the first of them, we shall present some metalinguistic considerations. Section 2 contains logic and set theory. In §3 we present a theory of programming, and a theory of computation is given in §4.

1. Some metalinguistic considerations

In this section we shall present various rules that allow one to correctly 'read' a formal theory. Mostly, such rules correspond to well accepted traditions in logic and in formal mathematics. Our purpose is only to make precise practices that are often implicitly followed by the working logician and mathematician.

This section is divided into four subsections covering the following topics: (*a*) lexical considerations, (*b*) syntactical considerations, (*c*) metatheoretic considerations, describing the structures of deduction rules and of definitions that constitute a theory, (*d*) linguistic considerations, describing conventions for the well-formedness of linguistic constructs.

These considerations will purposely take the form of rather 'dry' statements, as one might find them in the description of the rules of a game. The hurried reader may skip this section (at his own risk!).

(*a*) *Lexical considerations*

(i) A formula is a non-empty sequence made up of any symbol *except for* ' ', '$\vdash$', '$\triangleq$' (blank, turnstile, definition).

(ii) In a formula, each individual symbol is meaningful, *the only exception* being bold lower case letters. In fact, one or more such letters, arranged in a continuous sequence, stand (collectively and in that order) for an individual symbol. In what follows, by extension, we shall call such a sequence also a symbol. For instance, 'P', '→', '**proc**' and '**rec**' are all symbols.

(b) *Syntactical considerations*

(i) All symbols have an arity that is one of 0, 1 or 2.

(ii) Upper case and non-italic letters are symbols of arity 0 (but there exist other symbols of arity 0).

(iii) In a formula, all symbols of arity 1 are prefixed, *except the symbol* '′' (prime), which is postfixed.

(iv) In a formula, all symbols of arity 2 are infixed.

(v) All symbols have the same priority of 0, *except six symbols, which are*: '|', ';', ':', '[', '{' and '′' (bar, semicolon, colon, left square bracket, left curly bracket, prime).

(vi) When priority is the same, association is from left to right. For instance, the formula '$P \wedge Q \rightarrow R \vee P$' is to be understood '$((P \wedge Q) \rightarrow R) \vee P$'. Parentheses (i.e. '(' and ')') can always be added to enhance readability or to circumvent the left to right rule.

(vii) The three symbols '|', ';' and ':' have priority −3, −2 and −1 respectively (and, as we shall see, arity 2). For instance, the formula '$X:S;Y:S \rightarrow T;Z:T|P \vee Q$' is to be understood '$(((X:S);(Y:(S \rightarrow T)));(Z:T))|(P \vee Q)$'.

(viii) The two symbols '{' and '[' have priority 1 (and, as we shall see, arity 1 and 2 respectively). Moreover, these symbols have corresponding closing symbols '}' and ']'. For instance, the formula '$P[S/X] \vee Q[S/X]$' is to be understood '$(P[(S/X)) \vee (Q[(S/X))$'.

(ix) The symbol '′' has priority 1 (and, as we have seen, arity 1).

(c) *Metatheoretical considerations*

A theory is defined by means of a finite number of rules of two different kinds: deduction rules and definition rules.

(i) A deduction rule is made up of two parts: first, the antecedent, which comprises zero, one or more formulae separated by a 'blank' symbol and secondly, the consequent, which is made of a single formula. Antecedent and consequent are separated by the turnstile symbol '⊢'. For instance, the following are deduction rules:

$$\text{`}P \quad P \rightarrow Q \vdash Q\text{'}, \quad \text{`}\vdash (P \vee P) \rightarrow P\text{'}.$$

(ii) A definition rule is also a rule comprising two parts: the antecedent (as for deduction rules), and the definition. Antecedent and definition are separated by the turnstile symbol '⊢'. The definition part comprises two formulae separated by the definition symbol '≙'. For instance, the following are definition rules:

$$\text{`}X \backslash A \vdash A[S/X] \triangleq A\text{'}, \quad \text{`}\vdash P \wedge Q \triangleq \sim(\sim P \vee \sim Q)\text{'}.$$

(iii) A rule, be it a deduction or a definition, stands for an infinite number of other rules called its *instances*. An instance of a rule can be obtained mechanically by simultaneously replacing all occurrences of some (or all) upper case non-italic letters contained in it by corresponding formulae. For instance, the rule '$P \vee Q \quad (P \vee Q) \rightarrow (Q \vee P) \vdash Q \vee P$' is an instance of the rule '$P \quad P \rightarrow Q \vdash Q$' because the former has been obtained by simultaneously replacing all occurrences of 'P' and 'Q' in the latter by '$P \vee Q$' and '$Q \vee P$' respectively.

(d) *Well-formedness*

(i) Five non-logical theories, which are fully described in Appendix 1, are mainly concerned with the following unary symbols: '**pred**', '**term**', '**vrbl**', '**decl**', and '**inst**'.

(ii) Formulae such as '**pred**(P)', '**term**(T)', '**vrbl**(X)', '**decl**(D)' and '**inst**(I)' are respectively to be read 'P is a well formed predicate', 'T is a well formed Term', 'X is a well formed variable', 'D is a well formed declaration', 'I is a well formed instruction'.

(iii) From now on, and to simplify other theories, we shall suppose that occurrences of certain upper case letters in a rule imply that certain corresponding formulae such as those described are implicitly present in its antecedent. The association between these unary symbols and the upper case letters is given in table 1.

Table 1. Association between unary symbols and upper case letters

pred	P	Q	R	—
term	S	T	U	V
vrbl	X	Y	Z	N
decl	D	E	F	—
inst	I	J	K	—

For instance, a rule such as '$\vdash (S = T) \rightarrow (P[S/X] \rightarrow P[T/X])$' stands for the more complete rule:

$$\text{'}\mathbf{pred}(P) \quad \mathbf{term}(S) \quad \mathbf{term}(T) \quad \mathbf{vrbl}(X) \vdash (S = T) \rightarrow (P[S/X] \rightarrow P[T/X])\text{'}.$$

2. Logic and set theory

In this section we introduce logic and set theory through a number of embedded theories, starting with propositional calculus and ending with natural numbers. Each theory will correspond to a separate subsection organized as follows.

We shall first explain informally what the subject of the theory is. To do so, we shall introduce the specific symbols of the theory together with their English meaning. We shall also explain the main properties of each symbol. All this will be formalized through a series of rules that constitute the theory.

As already mentioned (see §1 (*c*)), we have two kinds of rules: deduction rules and definition rules. Each deduction rule may have an antecedent, in which case it is a rule of inference, or no antecedent, in which case it is an axiom. We have already mentioned (see §1 (*d*)) that the formulae involved in the antecedent of a rule may quite well be non-logical statements (i.e. statements that do not belong to the language of logic or set theory).

The definition part of a definition rule introduces a (symmetric) *syntactic equivalence* between the formulae that lie on each side of the '$\triangleq$' symbol. Thus in this paper, a definition rule (which can be *basic*, that is given in a theory, or *derived*, that is proved within a theory) means more than the simple introduction of an abbreviation, which is traditionally implied by the word 'definition'. Moreover a definition rule may also have an antecedent; in this case, the syntactic equivalence implied by the definition is subjected to a number of conditions that must be checked before applying the definition in a proof. For instance, the following definition rule (in fact a derived definition rule of predicate calculus):

$$\text{'}X\backslash P \vdash EX.(P \wedge Q) \triangleq P \wedge (EX.P)\text{'}$$

means that the formula '$EX.(P \wedge Q)$' may be replaced by '$P \wedge (EX.P)$' (and *vice versa*) in the midst of any logical formula; this transformation, however, is subjected to the condition '$X \backslash P$', which means 'X is not free in P'. As can be seen, '$X \backslash P$' is not a logical statement. It is, however, a formal statement to which there corresponds a theory, the theory of variables (see §2(*b*)), which allows one to formally prove statements of this kind.

In some theories, the introduction of new symbols may necessitate the extension of previously defined theories.

Each section will finish with statements of the most useful results of the theory. These statements take the form of other (derived) deduction or definition rules. To keep this paper to a reasonable length, we have decided not to give any formal proof. Instead, we occasionally give some hints.

(*a*) *Propositional calculus*: P

Propositional calculus formalizes elementary reasonings involving the four basic Boolean connectives. The theory presented here is known as the Hilbert–Ackermann system; however, these authors recognized that it was essentially due to A. N. Whitehead and B. Russell. We have only added conditional definition P6, which cannot be proved within the logical system alone.

symbol	$\sim$	$\vee$	$\rightarrow$	$\wedge$
arity	1	2	2	2

Main rules

P1 $\vdash P \rightarrow (P \vee Q)$

P2 $\vdash (P \vee Q) \rightarrow (Q \vee P)$

P3 $\vdash (P \vee P) \rightarrow P$

P4 $\vdash (Q \rightarrow R) \rightarrow ((P \vee Q) \rightarrow (P \vee R))$

P5 $P \quad P \rightarrow Q \vdash Q$

P6 $P \rightarrow Q \quad Q \rightarrow P \vdash P \triangleq Q$

Abbreviations

P7 $\vdash P \rightarrow Q \triangleq \sim P \vee Q$

P8 $\vdash P \wedge Q \triangleq \sim (\sim P \vee \sim Q)$

Among the standard results of propositional calculus, two seem to be quite significant. These are the rule of excluded middle and the rule of double negation.

$$\vdash P \vee \sim P$$
$$\vdash P \triangleq \sim\sim P$$

(*b*) *Theory of variables*: V

The theory of variables (our first non-logical theory), axiomatizes expressions or assertions such as 'a variable that does not occur free in formulae A and B' or 'the variable X is not free in A' or 'the variables X and Y are distinct variables'. To do so, we introduce two symbols of arity 2, '^' and '\'. The formula '$A \hat{} B$' is the formal translation of the first quoted English sentence; consequently '$A \hat{} B$' *is* a variable. The formula '$X \backslash A$' corresponds to the second quoted sentence. When 'A' is 'Y', we obtain '$X \backslash Y$', which corresponds to the formal translation of the third quoted sentence.

As can be seen in Appendix 1, if 'X' and 'Y' are well formed variables and if 'X' is distinct from 'Y' (i.e. 'X\Y') then 'X,Y' *is also a well formed variable.* Consequently, we have to explain in this theory under which circumstances the variable 'X,Y' is not free in the formula 'A'. This is obviously when both 'X' and 'Y' are not themselves free in 'A'.

symbol	ˆ	\	,
arity	2	2	2

Main rules

V1 $\vdash$ AˆB $\triangleq$ BˆA
V2 $\vdash$ Aˆ(BˆC) $\triangleq$ (AˆB)ˆC
V3 $\vdash$ (AˆB)\A
V4 $\vdash$ X\Y $\triangleq$ Y\X
V5 X\A Y\A $\vdash$ (X, Y)\A

Unless otherwise stated, all symbols that we shall introduce will be subjected to one of the following rules depending upon their arity. In these rules, 'ω' stands for any such symbol.

V6 $\vdash$ X\ω
V7 X\A $\vdash$ X\ωA
V8 X\A X\B $\vdash$ X\(AωB)

Note that rule V7 is already applicable to '$\sim$' and rule V8 to '$\vee$'.

(*c*) *Theory of variable substitution*: S

The theory of variable substitution (again, a non-logical theory) axiomatizes expressions such as 'the formula A where the term S has been substituted for all free occurrences of the variable X'. To do so, we introduce the symbol '[' (which, as has been seen already (see §1 (*b*)), is a symbol of arity 2, priority 1 and with a corresponding closing symbol ']'), and the symbol '/' so that the formula 'A[S/X]' is the formal translation of the above quoted sentence.

The first rule of this theory explains the essence of substitution, namely that 'S' substituted for 'X' in 'X' is 'S'. We, then, give general rules showing under which circumstances a substitution has no effect (i.e. when the variable is substituted for itself, when the variable is not free in the formula). Finally, we give rules for successive substitutions as well as for simultaneous ones.

symbol	[	/
arity	2	2

Main rules

S1 $\vdash$ X[S/X] $\triangleq$ S
S2 $\vdash$ A[X/X] $\triangleq$ A
S3 X\A $\vdash$ A[S/X] $\triangleq$ A
S4 Y\A $\vdash$ A[Y/X][S/Y] $\triangleq$ A[S/X]
S5 X\Y X\T $\vdash$ A[S/X][T/Y] $\triangleq$ A[T/Y][S[T/Y]/X]
S6 X\Y Y\S $\vdash$ A[(S, T)/(X, Y)] $\triangleq$ A[S/X][T/Y]

Extensions to theory V

V9 $\quad X\backslash S \vdash X\backslash A[S/X]$

V10 $\quad X\backslash A \quad X\backslash S \vdash X\backslash A[S/Y]$

As for theory V, *and unless otherwise stated*, all symbols that we shall introduce thereafter will be subjected to one of the following rules depending upon their arity. In these rules 'ω' stands for any such symbol.

S7 $\quad \vdash (\omega A)[S/X] \triangleq \omega(A[S/X])$

S8 $\quad \vdash (A\omega B)[S/X] \triangleq (A[S/X])\ \omega(B[S/X])$

The main result of this theory is the commutativity of simultaneous substitution. It can be stated as

$$X\backslash Y \vdash A[(S,T)/(X,Y)] \triangleq A[(T,S)/(Y,X)].$$

This result can be proved by using rules S4, S5 and S6 together with the auxiliary variables '$A\hat{\ }S\hat{\ }X$' and '$(A\hat{\ }T\hat{\ }Y)\hat{\ }(A\hat{\ }S\hat{\ }X)$'.

(*d*) *Predicate calculus*: Q

Predicate calculus formalizes reasonings that involve objects. Formulae denoting objects are called terms. For the moment, well formed terms are variables only (see Appendix 1 for a complete definition of well formed terms).

symbol	*E*	*A*	.
arity	1	1	2

Main rules

Q1 $\quad \vdash P[S/X] \rightarrow (EX\,.\,P)$

Q2 $\quad P \vdash AX\,.\,P$

Q3 $\quad X\backslash Q \vdash (AX\,.\,(P \rightarrow Q)) \rightarrow ((EX\,.\,P) \rightarrow Q)$

Abreviations

Q4 $\quad X\backslash Y \vdash E(X,Y)\,.\,P \triangleq EX\,.\,(EY\,.\,P)$

Q5 $\quad \vdash AX\,.\,P \triangleq \sim(EX\,.\,\sim P)$

Extension to theory V

V11 $\quad \vdash X\backslash(EX\,.\,P)$

V12 $\quad X\backslash P \vdash X\backslash(EY\,.\,P)$

Extension to theory S

S9 $\quad Y\backslash X \quad Y\backslash S \vdash (EY\,.\,P)[S/X] \triangleq EY\,.\,P[S/X]$

Rule V11 defines the *scope* of the quantifier '*E*' by explaining that the variable 'X' is not free in '*E*X . P'. Rule S9 explains under which circumstances a substitution can be performed within a quantified formula, namely when the quantified variable, here 'Y', is not free in the substituting term 'S'.

Among the many results that can be proved within predicate calculus, four seem to be particularly useful. They deal with the distribution of the quantifier 'E' (resp. 'A') over the Boolean connectives '$\vee$' and '$\wedge$', with the commutativity of ',' within the quantifier 'E' (resp. 'A'), and with changes of variable within quantified formulae.

$$\vdash E\mathrm{X}\,.\,(\mathrm{P} \vee \mathrm{Q}) \triangleq (E\mathrm{X}\,.\,\mathrm{P}) \vee (E\mathrm{X}\,.\,\mathrm{Q})$$
$$\mathrm{X}\backslash\mathrm{P} \vdash E\mathrm{X}\,.\,(\mathrm{P} \wedge \mathrm{Q}) \triangleq \mathrm{P} \wedge (E\mathrm{X}\,.\,\mathrm{Q})$$
$$\mathrm{X}\backslash\mathrm{Y} \vdash E(\mathrm{X},\mathrm{Y})\,.\,\mathrm{P} \triangleq E(\mathrm{Y},\mathrm{X})\,.\,\mathrm{P}$$
$$\mathrm{X}\backslash\mathrm{Y} \quad \mathrm{Y}\backslash\mathrm{P} \vdash E\mathrm{Y}\,.\,(\mathrm{P}[\mathrm{Y}/\mathrm{X}]) \triangleq E\mathrm{X}\,.\,\mathrm{P}$$

(*e*) *Equality theory*: E

Equality theory introduces, as expected, the equality symbol '$=$', which is not to be confused with the definition metasymbol '$\triangleq$'.

symbol	$=$	$\neq$
arity	2	2

Main rules

E1 $\mathrm{X}\backslash\mathrm{S} \vdash E\mathrm{X}\,.\,((\mathrm{X} = \mathrm{S}) \wedge \mathrm{P}) \triangleq \mathrm{P}[\mathrm{S}/\mathrm{X}]$

E2 $\vdash (\mathrm{S},\mathrm{T}) = (\mathrm{U},\mathrm{V}) \triangleq (\mathrm{S} = \mathrm{U}) \wedge (\mathrm{T} = \mathrm{V})$

Abbreviation

E3 $\vdash \mathrm{S} \neq \mathrm{T} \triangleq \sim(\mathrm{S} = \mathrm{T})$

As can be seen, we have not defined equality theory using the more traditional rules

$$\vdash (\mathrm{S} = \mathrm{T}) \rightarrow (\mathrm{P}[\mathrm{S}/\mathrm{X}] \rightarrow \mathrm{P}[\mathrm{T}/\mathrm{X}])$$
$$\vdash \mathrm{S} = \mathrm{S}$$

instead of rule E1. The reason for this choice is not only because these results can be proved (Hao Wang) from E1, but also because we wanted to emphasize the importance of rule E1 in what follows.

(*f*) *Theory of declarations*: D

The theory of declarations is our last non-logical theory. It formalizes the concept of declaration.

An elementary declaration is made of a variable together with a term that is called the *type* of the variable; this association is expressed by the binary symbol ':'. Two declarations can be combined (put together) by using the binary symbol ';'. Moreover, a declaration can be restricted by a predicate by using the symbol '|'. The theory of declarations is first concerned with the basic properties of these symbols.

symbol	:	;	|
arity	2	2	2

Main rules (*part* 1)

D1 $\vdash \mathrm{D};(\mathrm{E};\mathrm{F}) \triangleq (\mathrm{D};\mathrm{E});\mathrm{F}$

D2 $\vdash \mathrm{D};(\mathrm{E}|\mathrm{P}) \triangleq (\mathrm{D};\mathrm{E})|\mathrm{P}$

D3 $\vdash (\mathrm{D}|\mathrm{P});\mathrm{E} \triangleq (\mathrm{D};\mathrm{E})|\mathrm{P}$

D4 $\vdash (\mathrm{D}|\mathrm{P})|\mathrm{Q} \triangleq \mathrm{D}|(\mathrm{P} \wedge \mathrm{Q})$

As can be seen, a declaration can always be put in a *normalized form* made up of the left association of the combinations of all its elementary declarations followed by conjunction of all their restrictions.

We now introduce two more symbols, namely 'α' and 'σ', which, when applied to a declaration, yield its *alphabet* and *signature* respectively. The alphabet of a declaration is a variable comprising the individual variables of its elementary declarations. The signature of a declaration is a predicate involving a new binary symbol '$\in$', the *membership symbol*.

symbol	α	σ	$\in$
arity	1	1	2

Main rules (*part* 2)

D5 $\vdash \alpha(X:T) \mathrel{\hat{=}} X$
D6 $\vdash \alpha(D;X:T) \mathrel{\hat{=}} (\alpha D),X$
D7 $\vdash \alpha(D|P) \mathrel{\hat{=}} \alpha D$
D8 $\vdash \sigma(X:T) \mathrel{\hat{=}} X \in T$
D9 $\vdash \sigma(D;E) \mathrel{\hat{=}} (\sigma D) \wedge (\sigma E)$
D10 $\vdash \sigma(D|P) \mathrel{\hat{=}} (\sigma D) \wedge P$

As can be seen, alphabets are only defined for normalized declarations so that two declarations having the same normalized form have the same alphabet.

Finally, we introduce yet another form of declaration, the *empty* declaration, denoted by the symbol '**skip**'. Its obvious axiomatic properties are given.

symbol	**skip**
arity	0

Main rules (*part* 3)

D11 $\vdash \mathbf{skip};D \mathrel{\hat{=}} D$
D12 $\vdash D;\mathbf{skip} \mathrel{\hat{=}} D$
D13 $\vdash \sigma(\mathbf{skip})$

We now extend the theory of variables to deal with variables of the form '$\alpha(D;E)$' or '$\alpha(\mathbf{skip})$'.

Extension to theory V

V13 $\vdash \alpha(D;E)\backslash A \mathrel{\hat{=}} (\alpha D,\alpha E)\backslash A$
V14 $\vdash \alpha(\mathbf{skip})\backslash A$
V15 $\vdash \alpha(\mathbf{skip}), X \mathrel{\hat{=}} X$
V16 $\vdash X,\alpha(\mathbf{skip}) \mathrel{\hat{=}} X$

As can be seen, '$\alpha(\mathbf{skip})$' is the 'non-variable'. We also extend predicate calculus so that quantification may be applied to declarations as well.

Extension to theory Q

Q6 $\vdash ED\,.\,P \mathrel{\hat{=}} E(\alpha D)\,.\,(\sigma D \wedge P)$
Q7 $\vdash E\alpha(D;E)\,.\,P \mathrel{\hat{=}} E(\alpha D,\alpha E)\,.\,P$
Q8 $\vdash AD\,.\,P \mathrel{\hat{=}} {\sim}ED\,.\,{\sim}P$

The following results can be proved.

$$\vdash \alpha(\mathrm{D};\mathrm{E})[(\mathrm{S},\mathrm{T})/(\alpha\mathrm{D},\alpha\mathrm{E})] = \mathrm{U} \mathrel{\hat{=}} (\alpha\mathrm{D},\alpha\mathrm{E})[\mathrm{U}/\alpha(\mathrm{D};\mathrm{E})] = (\mathrm{S},\mathrm{T})$$
$$\vdash E\alpha(\mathbf{skip})\,.\,\mathrm{P} \mathrel{\hat{=}} \mathrm{P}$$
$$\vdash A\alpha(\mathbf{skip})\,.\,\mathrm{P} \mathrel{\hat{=}} \mathrm{P}$$

To prove the first of these results one may use rules V13, Q7 and E1.

(*g*) *Set theory*: T

Set theory, as defined here, involves essentially two kinds of terms introduced by the unary symbols '{' and '*P*'. (Remember (§1 (*b*)) that '{' is a symbol of arity 1 having a closing symbol '}'.) A term such as '{T|D}', where 'T' is a (well formed) term and 'D' is a (well formed) declaration (see Appendix 1 for the rigorous definition of well-formedness) is to be read 'the set of objects of the form T indexed by D'. A term such as '*P*T' is to be read 'the set of all subsets of T'.

The purpose of this theory is to give equivalents for predicates such as 'S∈{T|D}' and 'S∈*P*T'. Of course, set equality is also axiomatized as usual, by using the operation of set inclusion '⊂'. Finally, we axiomatize the empty set '∅'.

symbol	{	*P*	∅
arity	1	1	0

Main rules

T1 $\vdash (\mathrm{S} \subset \mathrm{T}) \wedge (\mathrm{T} \subset \mathrm{S}) \rightarrow (\mathrm{T} = \mathrm{S})$
T2 $\alpha\mathrm{D}\backslash\mathrm{S} \vdash \mathrm{S} \in \{\mathrm{T}|\mathrm{D}\} \mathrel{\hat{=}} E\mathrm{D}\,.\,(\mathrm{S} = \mathrm{T})$
T3 $\vdash \mathrm{S} \in P\mathrm{T} \mathrel{\hat{=}} \mathrm{S} \subset \mathrm{T}$
T4 $\vdash \mathrm{S} \notin \varnothing$

We shall now define a number of abbreviations for set inclusion, set comprehension, singleton, Cartesian product and non-membership.

symbol	⊂	×	∉
arity	2	2	2

Abbreviations

A1 $\mathrm{X}\backslash\mathrm{S} \quad \mathrm{X}\backslash\mathrm{T} \vdash \mathrm{S} \subset \mathrm{T} \mathrel{\hat{=}} A(\mathrm{X}:\mathrm{S})\,.\,(\mathrm{X} \in \mathrm{T})$
A2 $\vdash \{\mathrm{D}\} \mathrel{\hat{=}} \{\alpha\mathrm{D}|\mathrm{D}\}$
A3 $\vdash \{\mathrm{T}\} \mathrel{\hat{=}} \{\mathrm{T}|\mathbf{skip}\}$
A4 $\vdash \mathrm{S} \times \mathrm{T} \mathrel{\hat{=}} \{\mathrm{X}:\mathrm{S};\mathrm{Y}:\mathrm{T}\}$
A5 $\vdash \mathrm{S} \notin \mathrm{T} \mathrel{\hat{=}} {\sim}(\mathrm{S} \in \mathrm{T})$

The main result at this point shows that a set such as '{T|D}' can be typed whenever the term 'T' can itself be typed. Other traditional results have to do with the simpler forms of set as defined in A2 and A4.

$$\alpha\mathrm{D}\backslash\mathrm{S} \vdash (A\mathrm{D}\,.\,(\mathrm{T} \in \mathrm{S})) \rightarrow (\{\mathrm{T}|\mathrm{D}\} \in P\mathrm{S})$$
$$\mathrm{X}\backslash\mathrm{S} \vdash \mathrm{T} \in \{\mathrm{X}:\mathrm{S}|\mathrm{P}\} \mathrel{\hat{=}} (\mathrm{T} \in \mathrm{S}) \wedge \mathrm{P}[\mathrm{T}/\mathrm{X}]$$
$$\vdash (\mathrm{U},\mathrm{V}) \in (\mathrm{S} \times \mathrm{T}) \mathrel{\hat{=}} (\mathrm{U} \in \mathrm{S}) \wedge (\mathrm{V} \in \mathrm{T})$$

It is also necessary to extend the theory of variables and the theory of substitution. This is done in an obvious way as follows.

Extension to theory V

V17 $\vdash \alpha D \backslash \{T|D\}$
V18 $X\backslash T \quad X\backslash D \vdash X\backslash\{T|D\}$

Extension to theory S

S10 $\alpha D\backslash X \quad \alpha D\backslash S \vdash \{T|D\}[S/X] \triangleq \{T[S/X]|D[S/X]\}$

(*h*) *Theory of relations and functions*: R

We shall now introduce the symbols '$\nrightarrow$' and '$\rightarrow$' to define, as usual, the set of all partial functions and the set of all total functions from one set to another. To conveniently formalize the set of partial functions, we use another binary symbol, '$\lceil$', which, when applied to a predicate and to a variable, as in '$P\lceil X$', is to be read 'predicate P is functional in X'. Finally, we introduce the quantifier 'λ' used to define total functions by abstraction.

symbol	$\nrightarrow$	$\rightarrow$	$\lceil$	λ
arity	2	2	2	1

Main rules

R1 $Y\backslash P \quad Z\backslash P \vdash P\lceil X \triangleq A(Y,Z).(P[Y/X] \wedge P[Z/X] \rightarrow (Y = Z))$
R2 $\vdash S \nrightarrow T \triangleq \{Z:P(S\times T)|A(X:S).((X,Y\in Z)\lceil Y)\}$
R3 $\vdash S \rightarrow T \triangleq \{Z:S\nrightarrow T|A(X:S).(EY.(X,Y\in Z))\}$
R4 $\vdash \lambda D.T \triangleq \{\alpha D,T|D\}$

We shall now extend set theory to formalize the notion of a total function *evaluation*. If 'Y' is a total function from 'S' to 'T', then the term 'Y[X]' (where 'X' is supposed to be a member of 'S') is called the *value* of 'Y' at 'X'.

Extension to theory T

T5 $\vdash A(X:S;Y:S\rightarrow T).(X,Y[X]\in Y)$

The main results at this point concern the typing of a function evaluation, that of a function defined by lambda abstraction, and the correspondence between evaluation and term substitution.

$$\vdash A(Y:S\rightarrow T;X:S).(Y[X]\in T)$$
$$\alpha D/S \vdash (AD.(T\in S)) \rightarrow (\lambda D.T \in (\{D\}\rightarrow S))$$
$$\alpha D/S \vdash (AD.(T\in S)) \rightarrow (A(X:\{D\}).((\lambda D.T)[X] = T[X/\alpha D])$$

The last result is of particular importance; it shows under which circumstances one is able to equate a lambda-abstraction evaluation (i.e. '$(\lambda D.T)[X]$') with the substitution of the actual parameters (i.e. 'X') for the formal ones (i.e. 'αD') in its body (i.e. '$T[X/\alpha D]$'). This is (1) when the body in question is well typed for all typed values of the formal parameters (i.e. '$AD.(T\in S)$') and (2) when the actual parameters agree with the type of the formal ones (i.e. '$A(X:\{D\})$').

(*i*) *Set-theoretic fixpoint*: FP

In this extension to set theory, we introduce four symbols, namely '$\cap$', 'ϕ', 'M' and 'μ', which correspond respectively to the concepts of intersection of a set of sets, fixpoint and monotonicity of a set function, and minimalization.

symbol	$\cap$	ϕ	M	μ
arity	1	1	1	1

Main rules

FP1 $\vdash A(\mathrm{Z}:PP\mathrm{S}|\mathrm{Z} \neq \varnothing) . (\cap \mathrm{Z} = \{\mathrm{X}:\mathrm{S}|A(\mathrm{Y}:\mathrm{Z}) . (\mathrm{X} \in \mathrm{Y})\})$
FP2 $\vdash A(\mathrm{Y}:P\mathrm{S} \rightarrow P\mathrm{S}) . (\phi\mathrm{Y} = \cap \{\mathrm{X}:P\mathrm{S}|\mathrm{Y}[\mathrm{X}] \subset \mathrm{X}\})$
FP3 $\vdash M\mathrm{S} \triangleq \{\mathrm{Z}:P\mathrm{S} \rightarrow P\mathrm{S}|A(\mathrm{X}:P\mathrm{S};\mathrm{Y}:P\mathrm{S}|\mathrm{X} \subset \mathrm{Y}) . (\mathrm{Z}[\mathrm{X}] \subset \mathrm{Z}[\mathrm{Y}])\}$
FP4 $\vdash \mu(\mathrm{X}:P\mathrm{S}) . \mathrm{T} \triangleq \phi(\lambda(\mathrm{X}:P\mathrm{S}) . \mathrm{T})$

The main result (Tarski 1955) at this point shows that the fixpoint of a monotonic set function (definition FP3) is indeed a 'fixed-point', that it is the smallest of them, and consequently, that any subset of it, provided it is also a fixpoint, is indeed equal to it.

$\vdash A(\mathrm{Y}:M\mathrm{S}) . (\mathrm{Y}[\phi\mathrm{Y}] = \phi\mathrm{Y})$
$\vdash A(\mathrm{Y}:M\mathrm{S};\mathrm{X}:P\mathrm{S}|\mathrm{X} = \mathrm{Y}[\mathrm{X}]) . (\phi\mathrm{Y} \subset \mathrm{X})$
$\vdash A(\mathrm{Y}:M\mathrm{S};\mathrm{X}:P\mathrm{S}|\mathrm{X} \subset \phi\mathrm{Y}|\mathrm{X} = \mathrm{Y}[\mathrm{X}]) . (\mathrm{X} = \phi\mathrm{Y})$

The last result is important because it allows one to reason by *induction* to prove universal properties for sets defined by the fixpoint operator 'ϕ' applied to monotonic set functions (Burstall 1969; Park 1969).

(*j*) *Finite subsets of a set*: F

The previous theory has given us the tools that allow us to define inductively the set of all finite subsets of a given set. For this, we only need to introduce the operation '^' of adding an element to a set. The set 'FS' of finite subsets of 'S' is then defined as the smallest subset of 'PS', which contains the empty set '$\varnothing$' and which is closed under the operation defined by the symbol '^'.

symbol	^	F
arity	2	1

Main rules

F1 $\vdash A(\mathrm{X}:\mathrm{S};\mathrm{Y}:P\mathrm{S}) . (\mathrm{X}\hat{}\mathrm{Y} = \{\mathrm{Z}:\mathrm{S}|(\mathrm{Z} = \mathrm{X}) \vee (\mathrm{Z} \in \mathrm{Y})\})$
F2 $\vdash F\mathrm{S} \triangleq \mu(\mathrm{Z}:PP\mathrm{S}) . (\varnothing\hat{}\{\mathrm{X}\hat{}\mathrm{Y}|\mathrm{X}:\mathrm{S};\mathrm{Y}:\mathrm{Z}\})$

The main result here concerns the possibility of proving properties of 'FS' by induction.

$\vdash \varnothing \in F\mathrm{S}$
$\vdash A(\mathrm{X}:\mathrm{S};\mathrm{Y}:F\mathrm{S}) . (\mathrm{X}\hat{}\mathrm{Y} \in F\mathrm{S})$
$\mathrm{P}[\varnothing/\mathrm{Y}] \quad A(\mathrm{X}:\mathrm{S};\mathrm{Y}:F\mathrm{S}|\mathrm{P}) . \mathrm{P}[\mathrm{X}\hat{}\mathrm{Y}/\mathrm{Y}] \vdash A(\mathrm{Y}:F\mathrm{S}) . \mathrm{P}$

An infinite set is one that is not a member of its finite subsets. An interesting property of infinite sets is that they are indeed infinite; in other words, the set difference between an infinite set and one of its finite subsets *is never empty*.

$\vdash \mathrm{S} \notin F\mathrm{S} \rightarrow (A(\mathrm{Y}:F\mathrm{S}) . (\{\mathrm{X}:\mathrm{S}|\mathrm{X} \notin \mathrm{Y}\} \neq \varnothing))$

(*k*) *Natural numbers*: N

To construct the natural numbers, one must postulate the existence of an infinite set. We shall suppose that such a set is (God) given to us; its name is 'U'.

Although we can define the natural numbers without it, it is also convenient to suppose that, for each *non-empty* set 'S', there exists a *privileged member of it*, denoted by 'τS'. These elements constitute the last two rules of set theory.

symbol	U	τ
arity	0	1

Extension to theory T

T6 $\vdash U \notin FU$
T7 $\vdash \mathrm{S} \neq \varnothing \rightarrow (\tau\mathrm{S} \in \mathrm{S})$

Given a finite subset 'X' of 'U' we define its successor 'σ' by adding to it the privileged element of the set difference between 'U' and 'X'. We already know (§2 (*j*)) that such a set difference is not empty because 'U' is infinite and 'X' is one of its finite subsets. The set 'N' of natural numbers is then defined to be the smallest subset of 'FU', which contains the empty set (that is, '0') and which is closed under the successor operation 'σ'.

symbol	σ	N	0
arity	1	0	0

Main rules

N1 $\vdash A(\mathrm{X}:FU)\,.\,(\sigma\mathrm{X} = (\tau\{\mathrm{Y}:U \mid \mathrm{Y} \notin \mathrm{X}\}\hat{}\,\mathrm{X}))$
N2 $\vdash 0 \triangleq \varnothing$
N3 $\vdash N \triangleq \mu(\mathrm{X}:PFU)\,.\,(0\hat{}\{\sigma\mathrm{Y} \mid \mathrm{Y}:\mathrm{X}\})$

As expected, the main results are the five 'axioms' of Peano.

$\vdash 0 \in N$
$\vdash A(\mathrm{X}:N)\,.\,(\sigma\mathrm{X} \in N)$
$\vdash A(\mathrm{X}:N)\,.\,(\sigma\mathrm{X} \neq 0)$
$\vdash A(\mathrm{X}:N;\mathrm{Y}:N \mid \mathrm{X} \neq \mathrm{Y})\,.\,(\sigma\mathrm{X} \neq \sigma\mathrm{Y})$
$\mathrm{P}[0/\mathrm{X}] \quad A(\mathrm{X}:N \mid \mathrm{P})\,.\,\mathrm{P}[\sigma\mathrm{X}/\mathrm{X}] \vdash A(\mathrm{X}:N)\,.\,\mathrm{P}$

The four arithmetic operations can then be defined as well as the minimum of a non-empty subset of 'N'. Other types, such as finite sequences and finite trees, are also easily definable.

3. Theory of programming

As for logic and set theory, the theory of programming is made up of a series of embedded theories. The first of them is an extension to the theory of variables (§ 2 (*b*)). We then find various theories within which the features of the 'programming' notation are gradually introduced.

(*a*) *Theory of priming*

In this section we extend the 'variable language' by introducing the unary symbol ''' (prime) (remember (§2 (*b*)) that it is a postfixed symbol of priority 1). If 'X' is a well formed variable then so is 'X''. The variable 'X'' is distinct from 'X' and if 'X' and 'Y' are distinct variables then so are 'X'' and 'Y''. The priming of the variable 'X,Y' is 'X',Y''.

symbol	$'$
arity	1

Extension to theory V

$$\begin{array}{ll} \text{V19} & \vdash X \backslash X' \\ \text{V20} & \vdash X \backslash Y \mathrel{\hat{=}} X' \backslash Y' \\ \text{V21} & \vdash (X,Y)' \mathrel{\hat{=}} X',Y' \end{array}$$

By extension, the priming of a declaration corresponds to that of its alphabet.

Extension to theory D

$$\text{D14} \quad \vdash D' \mathrel{\hat{=}} D[(\alpha D)'/\alpha D]$$

An 'interesting' result is the following.

$$\vdash \textbf{skip}' \mathrel{\hat{=}} \textbf{skip}$$

(*b*) *Theory of programming*: *basic structures*

An instruction (of the 'programming' notation) is an incomplete predicate that can only be completed by a declaration. This is accomplished by an operation denoted by the binary symbol 'π'.

Given a declaration 'D' and an instruction 'I', '$D\pi I$' is a predicate that indicates how the values stored in the memories bearing the same names and types as those of the variables declared in 'D', are to be affected by the execution of 'I'.

(The use of expressions such as 'value stored', 'memory', 'execution' constitutes an obvious abuse of language. As already mentioned in the introduction, the proposed 'programming' notation does not constitute a genuine programming language; rather, it is a notation for writing certain logical and set-theoretic statements. However, and because certain features of this notation look like those of some existing programming languages, we have taken the liberty of explaining them *informally* in operational terms.)

More precisely, '$D\pi I$' is a predicate involving the variables 'αD' and '$\alpha D'$'. The former denotes the values stored in the corresponding memories *just before* the execution of 'I' (these values are members of the set '$\{D\}$') and the latter denotes the values stored in the *same* memories *just after* the execution of 'I' (these values are also members of '$\{D\}$'). The predicate '$D\pi I$' is obviously independent of the order of the elementary declarations in 'D'.

We now introduce the basic 'control' structures of the notation. Again, we shall give some informal explanation in terms of 'execution' as if these control structures were genuine programming features.

When the predicate 'P' holds, the execution of '$P \rightarrow I$' has the same effect as that of the instruction 'I'. When 'P' does not hold, however, the execution of '$P \rightarrow I$' does not give any result (which is *not* the same as having no effect).

The execution of '$I;J$' corresponds as usual, to the execution of the instruction 'I' followed by that of the instruction 'J'. If one of the two does not give any result, then their sequential composition does not give any result either.

The effect of the execution of '$I \square J$' is that of the execution of instruction 'I' *or* that of the execution of instruction 'J', the choice between the two being arbitrary. If one of the two does not give any result, then the effect of their non-deterministic alternation is that of the other. In this respect, the non-determinism involved in this alternation is said to be 'angelic'.

Finally, in the execution of '**loc**(D) . I', the instruction 'I' may use the variables declared in 'D' as local memories. Note that a non-initialized memory has nevertheless a value stored in it; this value is unknown. Again, we have here a very unrealistic situation.

symbol	$\square$	$\rightarrow$	;	**loc**
arity	2	2	2	1

Main rules

I1 $\vdash (D\pi I) \rightarrow \sigma(D;D')$
I2 $\vdash (D;E)\pi I \triangleq (E;D)\pi I$
I3 $\vdash D\pi(P \rightarrow I) \triangleq P \wedge (D\pi I)$
I4 $\vdash D\pi(I;J) \triangleq EX.((D\pi I)[X/\alpha D'] \wedge (D\pi J)[X/\alpha D])$
I5 $\vdash D\pi(I \square J) \triangleq (D\pi I) \vee (D\pi J)$
I6 $\vdash D\pi(\mathbf{loc}(E).I) \triangleq E\alpha(E;E').((D;E)\pi I)$

It should now be clear that the English expression 'instruction I does not give any result (in the context of the declaration D)' is formalized by '$\sim (D\pi I)$ is a tautology'. The main and obvious results concern the associativity of ';', the commutativity of '$\square$' and the distributivity of ';' over '$\square$'.

$$\vdash D\pi(I;J;K) \triangleq D\pi(I;(J;K))$$
$$\vdash D\pi(I \square J) \triangleq D\pi(J \square I)$$
$$\vdash D\pi((I \square J);K) \triangleq D\pi((I;K) \square (J;K))$$
$$\vdash D\pi(I;(J \square K)) \triangleq D\pi((I;J) \square (I;K))$$

Note that the instruction '**loc**(**skip**|P) . I' has the effect of imposing that the predicate 'P' is an *invariant* of instruction 'I'.

(*c*) *Assignment*

At first, assignment may seem to be easy to formalize: given a variable 'X' and a term 'T', the execution of '$X \leftarrow T$' modifies the contents of the memories associated with the individual variables in 'X' by storing in them values corresponding to the evaluation of 'T'. What we have to formalize, however, is not the instruction '$X \leftarrow T$', which is, as already mentioned, an incomplete predicate; rather, we have to give an equivalent to the predicate '$D\pi(X \leftarrow T)$'. If some variables in 'X' are not declared in 'D', we cannot say anything about this predicate. On the other hand, if some variables declared in 'D' do not occur in 'X', we have to say something about them, namely that *they are not modified by the execution of this instruction*. Note that the values stored initially, and the final values as well, must belong to the set '{D}'. We shall also formalize the instruction '**skip**', which has no effect on any memory whatsoever.

symbol	$\leftarrow$	**skip**
arity	2	0

Extension to theory I

I7 $\vdash \mathrm{D}\pi(\mathbf{skip}) \mathrel{\hat{=}} \sigma\mathrm{D} \wedge (\alpha\mathrm{D}' = \alpha\mathrm{D})$

I8 $\mathcal{A}(\mathrm{D};\mathrm{E}).(\sigma(\mathrm{D};\mathrm{E}))[\mathrm{T}/\alpha\mathrm{D}] \vdash (\mathrm{D};\mathrm{E})\pi(\alpha\mathrm{D} \leftarrow \mathrm{T}) \mathrel{\hat{=}} \sigma\mathrm{D} \wedge (\alpha\mathrm{D}' = \mathrm{T}) \wedge (\mathrm{E}\pi(\mathbf{skip}))$

The main result at this point concerns the sequential composition of an assignment with another instruction.

$$\vdash (\mathrm{D};\mathrm{E})\pi(\alpha\mathrm{D} \leftarrow \mathrm{T};\mathrm{I}) \mathrel{\hat{=}} \sigma\mathrm{D} \wedge ((\mathrm{D};\mathrm{E})\pi\mathrm{I})[\mathrm{T}/\alpha\mathrm{D}]$$

Note that '$(\mathbf{skip})\pi(\mathbf{skip})$' is a tautology.

(*d*) *Procedures and procedure calls*

Given an instruction 'I' and a declaration 'D', the construct '**proc**(D) . I' is a procedure. The formal parameters of this procedure are specified in the declaration 'D'. The body of the procedure is the instruction 'I'.

Given a procedure 'S' and a variable 'X', the construct 'S[X]' is an instruction that is the call of the procedure 'S'. The actual parameters of this call are those individual variables occurring in 'X'. (Thus, actual parameters are passed 'by reference').

A procedure such as '**proc**(D) . I' *is a binary relation*; more precisely, it is the subset of '{D} × {D}', the members of which are such that the predicate 'DπI' holds. This definition dictates that of 'DπS[X]': if 'X' is exactly 'αD' then 'DπS[X]' is equivalent to the membership of the pair '(αD,αD′)' to the binary relation 'S', provided, however, that 'S' is a binary relation of the right type, namely '*P*({D} × {D})' (we shall abbreviate this set by '*R*D'). If some variables in 'D' are not in 'X', then we have a '**skip**' action on these variables, as for assignment. Finally, if some variables in 'X' are not declared in 'D', we cannot say anything about the predicate 'DπS[X]'.

symbol	**proc**	[	*R*
arity	1	2	1

Extension to theory I

I9 $\vdash \mathbf{proc}(\mathrm{D}).\mathrm{I} \mathrel{\hat{=}} \{\alpha\mathrm{D},\alpha\mathrm{D}' | (\mathrm{D};\mathrm{D}'|\mathrm{D}\pi\mathrm{I})\}$

I10 $\alpha\mathrm{D}\backslash\mathrm{S} \quad \mathcal{A}\mathrm{E}.(\mathrm{S} \in \mathit{R}\mathrm{D}) \vdash (\mathrm{D};\mathrm{E})\pi\mathrm{S}[\alpha\mathrm{D}] \mathrel{\hat{=}} ((\alpha\mathrm{D},\alpha\mathrm{D}') \in \mathrm{S}) \wedge (\mathrm{E}\pi(\mathbf{skip}))$

Extension to theory A (*abbreviations for set theory*)

A6 $\vdash \mathit{R}\mathrm{D} \mathrel{\hat{=}} \mathit{P}(\{\mathrm{D}\} \times \{\mathrm{D}\})$

Note that we can unify procedure calls and assignments, as the predicate '$\sigma\mathrm{D} \wedge (\alpha\mathrm{D}' = \mathrm{T})$' involved in rule I8 can be rewritten '$(\alpha\mathrm{D},\alpha\mathrm{D}') \in (\lambda\mathrm{D}.\mathrm{T})$' (provided '$\lambda\mathrm{D}.\mathrm{T}$' is a genuine total function from '{D}' to '{D}'). As a consequence, we have the definition:

I11 $\vdash \alpha\mathrm{D} \leftarrow \mathrm{T} \mathrel{\hat{=}} (\lambda\mathrm{D}.\mathrm{T})[\alpha\mathrm{D}]$.

(It should be clear that the formula '$(\lambda\mathrm{D}.\mathrm{T})[\alpha\mathrm{D}]$', being a term and at the same time an instruction, is nevertheless not ambiguous. This is so because these distinct usages of the same formula never occur within the same linguistic context.) To deal directly with guarded assignments, we have the special case

I11′ $\vdash \mathrm{P} \rightarrow (\alpha\mathrm{D} \leftarrow \mathrm{T}) \mathrel{\hat{=}} (\lambda(\mathrm{D}|\mathrm{P}).\mathrm{T})[\alpha\mathrm{D}]$.

As can be seen, we cannot deal with procedures having global variable assignments. In fact, no genuine predicate can ever be transformed into a construct such as 'DπI', where 'I' would contain an assignment to a variable not declared in 'D' (there does not exist any rule, the backwards application of which would lead to such a construct). Consequently, no 'procedure' such as '**proc**(D) . I' can ever be *constructed* that would contain an assignment to a variable not declared in 'D'.

Actual parameters are systematically passed 'by reference'. The effect of passing actual parameters 'by value' can be obtained for actual parameters corresponding to formal parameters occurring at the end of the declaration. If 'S' is a procedure, the construct 'S[X;T]' is a procedure call with actual parameters 'X' passed 'by reference' and actual parameters 'T' passed 'by value'. The corresponding definition is

$$\text{I12} \quad \vdash \mathrm{S}[\alpha\mathrm{D};\mathrm{T}] \triangleq \mathbf{loc}(\mathrm{E})\,.\,(\alpha\mathrm{E} \leftarrow \mathrm{T};\mathrm{S}[\alpha(\mathrm{D};\mathrm{E})]).$$

Note that the fact that the actual parameters called 'by value' should correspond to the last parameters in formal parameter declaration is not really a limitation, as the order of these parameters may easily be rearranged within an *ad hoc* procedure.

The main result of this theory is the obvious identity

$$\vdash \mathrm{D}\pi(\mathbf{proc}(\mathrm{D})\,.\,\mathrm{I})[\alpha\mathrm{D}] \triangleq \mathrm{D}\pi\mathrm{I}$$

(*e*) *Parallelism*

Parallelism (simultaneous execution) is not defined for instructions in general; rather it is defined for procedure calls. For this reason, in this section, we shall call a procedure call a *process*. Given two processes 'S[α(D; E)]' and 'T[α(E; F)]', their parallel execution denoted by the symbol '$\|$' is also a process.

symbol	$\|$
arity	2

$$\text{I13} \quad \vdash (\mathrm{S}[\alpha(\mathrm{D};\mathrm{E})])\,\|\,(\mathrm{T}[\alpha(\mathrm{E};\mathrm{F})]) \triangleq \mathrm{U}[\alpha\mathrm{A}]$$

where

U is $\{\alpha\mathrm{A},\alpha\mathrm{A}' \mid (\mathrm{A};\mathrm{A}' \mid \mathrm{B} \wedge \mathrm{C})\}$
A is (D; E; F)
B is (D; E)πS[α(D; E)]
C is (E; F)πT[α(E; F)]

The variables in the common declaration 'E' are acting as *communication channels* between the two processes. When 'E' is '**skip**', the two processes do not communicate.

(*f*) *Recursion and loop*

If 'I' is an instruction and 'X' is a variable, then the construct '**rec**(X) . I' is another instruction; the execution of this instruction corresponds to the execution of 'I' within which occurrences of calls to 'X' are replaced by '**rec**(X) . I' itself!

(The limitation of informal 'operational' explanations appears now quite clearly; this English description of what '**rec**(X) . I' is supposed to be is obviously extremely vague, if not misleading.)

To give an equivalent to the predicate '$\mathrm{D}\pi(\mathbf{rec}(\mathrm{X}).\mathrm{I})$', we consider the function '$\lambda(\mathrm{X}{:}R\mathrm{D}).(\mathbf{proc}(\mathrm{D}).\mathrm{I})$'; this is a function of type '$R\mathrm{D}\rightarrow R\mathrm{D}$'; consequently the fixpoint (§2(*i*)) of this function is a member of '$R\mathrm{D}$', that is, a binary relation from '{D}' to '{D}'. The predicate '$\mathrm{D}\pi(\mathbf{rec}(\mathrm{X}).\mathrm{I})$' corresponds to the 'call' of this relation.

We also define the '**loop**' of an instruction 'I' as a certain recursion on 'I'.

symbol	**rec**	**loop**
arity	1	1

Extension to theory I

$$\text{I14}\quad \vdash \mathbf{proc}(\mathrm{D}).(\mathbf{rec}(\mathrm{X}).\mathrm{I}) \triangleq \mu(\mathrm{X}{:}R\mathrm{D}).(\mathbf{proc}(\mathrm{D}).\mathrm{I})$$
$$\text{I15}\quad \vdash \mathrm{D}\pi\mathbf{loop}(\mathrm{I}) \triangleq \mathrm{D}\pi(\mathbf{rec}(\mathrm{X}).((\mathrm{I};\mathrm{X}[\alpha\mathrm{D}])\square(\sim(E\alpha\mathrm{D}'.(\mathrm{D}\pi\mathrm{I}))\rightarrow\mathbf{skip})))$$

We can use previous results of the theory of functions (§2(*h*)) and of the theory of set-theoretic fixpoint (§2(*i*)) to prove the expected results

$$(\lambda(\mathrm{X}{:}R\mathrm{D}).\mathrm{A})\in M(\{\mathrm{D}\}\times\{\mathrm{D}\}) \vdash \mathrm{B} \triangleq \mathrm{A}[\mathrm{B}/\mathrm{X}]$$

where

A is $\mathbf{proc}(\mathrm{D}).\mathrm{I}$
B is $\mathbf{proc}(\mathrm{D}).(\mathbf{rec}(\mathrm{X}).\mathrm{I})$

$$\mathrm{C} \vdash \mathrm{D}\pi\mathbf{loop}(\mathrm{I}) \triangleq \mathrm{D}\pi((\mathrm{I};\mathbf{loop}(\mathrm{I}))\square(\sim(E\alpha\mathrm{D}'.(\mathrm{D}\pi\mathrm{I}))\rightarrow\mathbf{skip}))$$

where

C is $(\lambda(\mathrm{X}{:}R\mathrm{D}).(\mathbf{proc}(\mathrm{D}).(\mathrm{I};\mathrm{X}[\alpha\mathrm{D}]))\in M(\{\mathrm{D}\}\times\{\mathrm{D}\})$

4. Theory of computation

In this last section we shall formally define the concept of *computation*. To do so, we shall introduce the concept of nested substitution (§4.1) and that of set-theoretic continuous function (§4.2). In the last subsection we shall see what is meant by a computable instruction.

(*a*) *Nested iterated substitution*: IT

The main purpose of this section is to formally define the concept of *nested iterated substitutions*. Given a term 'T', a natural number 'N', another term 'U', and a variable 'X', the construct $\mathrm{T}[\mathrm{N}{\downarrow}(\mathrm{U}/\mathrm{X})]$ corresponds to '$\mathrm{T}[\mathrm{T}[\ldots\mathrm{T}[\mathrm{U}/\mathrm{X}]\ldots/\mathrm{X}]/\mathrm{X}]$', where the innermost substitution is embedded '$\mathrm{N}-1$' times. For instance, '$\mathrm{T}[2{\downarrow}(\mathrm{U}/\mathrm{X})]$' is '$\mathrm{T}[\mathrm{T}[\mathrm{U}/\mathrm{X}]/\mathrm{X}]$', '$\mathrm{T}[3{\downarrow}(\mathrm{U}/\mathrm{X})]$' is '$\mathrm{T}[\mathrm{T}[\mathrm{T}[\mathrm{U}/\mathrm{X}]/\mathrm{X}]/\mathrm{X}]$', etc.

To give a rigorous definition to such a construct we first need to define the iterate '**itrt**' of a total function from a set to itself. Given such a function 'Y', '**itrt**(Y)' is another function that, when applied to a number 'N', yields the 'Nth' iterate of 'Y' (this is denoted by '$\mathrm{Y}{\uparrow}\mathrm{N}$'). The definition of '**itrt**' uses that of the identity function on a set 'S' (this is denoted by 'IS') and that of the composition '$\circ$', of two total functions.

symbol	I	$\circ$	**itrt**	$\uparrow$	$\downarrow$
arity	1	2	1	2	2

Main rules

IT1 $\vdash I\mathrm{S} \triangleq \lambda(\mathrm{X}:\mathrm{S}).\mathrm{X}$

IT2 $\vdash A(\mathrm{X}:\mathrm{S}\to\mathrm{T};\mathrm{Y}:\mathrm{T}\to\mathrm{U}).(\mathrm{Y}\circ\mathrm{X} = (\lambda(\mathrm{Z}:\mathrm{S}).\mathrm{Y}[\mathrm{X}[\mathrm{Z}]]))$

IT3 $\vdash A(\mathrm{Y}:\mathrm{S}\to\mathrm{S}).(\mathbf{itrt}(\mathrm{Y}) = \mathrm{A})$

where A is $\mu(\mathrm{Z}:P(N\times(\mathrm{S}\to\mathrm{S}))).((0,I\mathrm{S})\hat{}\{\sigma\mathrm{N},(\mathrm{Y}\circ\mathrm{X})\,|\,(\mathrm{N},\mathrm{X}):\mathrm{Z}\})$

IT4 $\vdash A(\mathrm{Y}:\mathrm{S}\to\mathrm{S},\mathrm{N}:N).(\mathrm{Y}\uparrow\mathrm{N} = (\mathbf{itrt}(\mathrm{Y}))[\mathrm{N}])$

Extension of theory S

S11 $\vdash \mathrm{T}[\mathrm{N}\downarrow(\mathrm{U}/\mathrm{X})] \triangleq ((\lambda(\mathrm{X}:\mathrm{S}).\mathrm{T})\uparrow\mathrm{N})[\mathrm{U}]$

The expected intuitive results are

$\vdash A(\mathrm{Y}:\mathrm{S}\to\mathrm{S}).(\mathbf{itrt}(\mathrm{Y})\in(N\to(\mathrm{S}\to\mathrm{S})))$

$\vdash A(\mathrm{Y}:\mathrm{S}\to\mathrm{S}).(\mathrm{Y}\uparrow 0 = I\mathrm{S})$

$\vdash A(\mathrm{Y}:\mathrm{S}\to\mathrm{S};\mathrm{N}:N).(\mathrm{Y}\uparrow\sigma\mathrm{N} = (\mathrm{Y}\circ(\mathrm{Y}\uparrow\mathrm{N})))$

$\vdash \mathrm{A}\to(\mathrm{T}[0\downarrow(\mathrm{U}/\mathrm{X})] = \mathrm{U})$

$\vdash \mathrm{A}\to(A(\mathrm{N}:N).(\mathrm{T}[\sigma\mathrm{N}\downarrow(\mathrm{U}/\mathrm{X})] = \mathrm{T}[\mathrm{T}[\mathrm{N}\downarrow(\mathrm{U}/\mathrm{X})]/\mathrm{X}])$

where A is $(A(\mathrm{X}:\mathrm{S}).(\mathrm{T}\in\mathrm{S}))\wedge(\mathrm{U}\in\mathrm{S})$.

The first of these results is a consequence of rule IT3. It can be proved by mathematical induction and by using the fourth Peano axiom, which states that the successor function 'σ' is 'one–one' on 'N'.

(*b*) *Set-theoretic continuous functions*: C

When a set function (i.e. a function from 'PS' to itself) is *continuous*, its fixpoint (§2(*i*)) can be equated to certain iterates. More precisely, when the set function in question is defined by lambda abstraction (§2(*h*)), its fixpoint is equal to the (infinite) union of all the nested substitutions of the empty set in its body.

To define 'CS', the set of continuous set functions built on 'S', we need to introduce the union '$\cup$' of a set of sets and also the set of increasing infinite sequences of subsets of a set 'S' (this is denoted by '**inc**(S)').

symbol	$\cup$	**inc**	C
arity	1	1	1

Main rules

C1 $\vdash A(\mathrm{Z}:PP\mathrm{S}).(\cup\mathrm{Z} = \{\mathrm{X}:\mathrm{S}|E(\mathrm{Y}:\mathrm{Z}).(\mathrm{Z}\in\mathrm{Y})\})$

C2 $\vdash \mathbf{inc}(\mathrm{S}) \triangleq \{\mathrm{Y}:N\to P\mathrm{S}|A(\mathrm{N}:N).(\mathrm{Y}[\mathrm{N}]\subset\mathrm{Y}[\sigma\mathrm{N}])\}$

C3 $\vdash C\mathrm{S} \triangleq \{\mathrm{Y}:P\mathrm{S}\to P\mathrm{S}|A(\mathrm{X}:\mathbf{inc}(\mathrm{S})).\mathrm{A}\}$

where A is $\cup\{\mathrm{Y}[\mathrm{X}[\mathrm{N}]]|\mathrm{N}:N\} = \mathrm{Y}[\cup\{\mathrm{X}[\mathrm{N}]|\mathrm{N}:N\}]$

Note how the operations '$\cup$' and '[' commute in the definition of 'CS'. The main result (Kleene 1952) is the one given at the beginning of this section.

$$(\lambda(\mathrm{X}:P\mathrm{S}).\mathrm{T})\in C\mathrm{S}\vdash\mu(\mathrm{X}:P\mathrm{S}).\mathrm{T} \triangleq \cup\{\mathrm{T}[\mathrm{N}\downarrow(\varnothing/\mathrm{X})]|\mathrm{N}:N\}$$

This result can be put in another form.

$$(\lambda(\mathrm{X}:\mathit{P}\mathrm{S}).\mathrm{T}) \in \mathit{C}\mathrm{S} \vdash \mathrm{U} \in (\mu(\mathrm{X}:\mathit{P}\mathrm{S}).\mathrm{T}) \triangleq \mathit{E}(\mathrm{N}:\mathit{N}).(\mathrm{U} \in \mathrm{T}[\mathrm{N}\!\downarrow\!(\varnothing/\mathrm{X})])$$

In other words the membership of a term 'U' to a fixed-point defined set is equivalent to an *existential quantification* (over the natural number) of the membership of the same term to a term obtained by nested substitution.

Another important result (J. W. de Bakker & D. Scott, 'A theory of programs' (unpublished notes (1969))) allows one to prove universal properties of fixpoints of continuous functions.

(*c*) *Computation*

In this subsection, at last, we deal with the concept of *computation*. As expected, this concept is related to that of proof; when all other means have been exhausted, the only way to prove a conjecture might be to undertake an exhaustive search (a computation) through the problem space.

Where the conjecture in question takes the form of an *existential quantification*, the exhaustive search is, by definition, guaranteed to be successful *if the conjecture is provable*.

If we look at the various forms taken by the predicate 'DπI' (§3) for the various instructions 'I' of the 'programming' notation, we see that the only one that is not an existential form is 'Dπ(**rec**(X).I)' (and of course 'Dπ(**loop**(I))', as it is defined in terms of it). All other predicates 'invoke' similar predicates through existential quantifications (or by simple Boolean connections; this is so for 'Dπ(P→I)' or 'Dπ(I□J)').

We shall see that, provided a certain condition (of continuity) is met, we can also give an existential (computable) form to the predicate 'Dπ(**rec**(X).I)'.

To do so, we need to extend the 'programming' notation by introducing four more instructions. First, the instruction replacement instruction, the construct 'I[J/X]', where 'I' *and* 'J' are instructions, is an instruction, the execution of which corresponds to that of 'I', except that 'J' is executed upon 'encountering' the variable 'X'. Second, the nested generalization of the previous instruction; this is denoted by 'I[N↓(J/X)]'. Third, the 'Nth' iterate of instruction 'I', denoted by 'I↑N'. Finally, the '**break**' instruction, *which never gives any result*!

symbol	**break**
arity	0

Extension to theory I

$$\begin{array}{ll}
\mathrm{I16} & \vdash \mathrm{D}\pi(\mathrm{I}[\mathrm{J}/\mathrm{X}]) \triangleq (\mathrm{D}\pi\mathrm{I})[\mathbf{proc}(\mathrm{D}).\mathrm{J}/\mathrm{X}] \\
\mathrm{I17} & \vdash \mathrm{D}\pi\mathrm{I}[\mathrm{N}\!\downarrow\!(\mathrm{J}/\mathrm{X})] \triangleq (\mathrm{D}\pi\mathrm{I})[\mathrm{N}\!\downarrow\!(\mathbf{proc}(\mathrm{D}).\mathrm{J}/\mathrm{X})] \\
\mathrm{I18} & \vdash \mathrm{D}\pi(\mathrm{I}\!\uparrow\!\mathrm{N}) \triangleq \mathrm{D}\pi(\mathrm{I};\mathrm{X}[\alpha\mathrm{D}])[\mathrm{N}\!\downarrow\!(\mathbf{skip}/\mathrm{X})] \\
\mathrm{I19} & \vdash \sim(\mathrm{D}\pi(\mathbf{break}))
\end{array}$$

Now the main and well known result, a direct consequence of the result of the previous section, is

$$\mathrm{A} \vdash \mathrm{D}\pi(\mathbf{rec}(\mathrm{X}).\mathrm{I}) \triangleq \mathit{E}(\mathrm{N}:\mathit{N}).(\mathrm{D}\pi\mathrm{I}[\mathrm{N}\!\downarrow\!(\mathbf{break}/\mathrm{X})])$$

where A is $(\lambda(\mathrm{X}:\mathit{R}\mathrm{D}).(\mathbf{proc}(\mathrm{D}).\mathrm{I})) \in \mathit{C}(\{\mathrm{D}\}\times\{\mathrm{D}\})$.

For the loop instruction, this result reduces to

$$\mathrm{B} \vdash \mathrm{D}\pi(\mathbf{loop}(\mathrm{I})) \triangleq E(\mathrm{N}{:}N)\,.\,\mathrm{C}$$

where

$$\mathrm{C} \text{ is } \mathrm{D}\pi(\mathrm{I}{\uparrow}\mathrm{N}) \wedge \sim(E\mathrm{D}\,.\,(\mathrm{D}\pi\mathrm{I}))[\alpha\mathrm{D}'/\alpha\mathrm{D}]$$

$$\mathrm{B} \text{ is } (\lambda(\mathrm{X}{:}R\mathrm{D})\,.\,(\mathbf{proc}(\mathrm{D})\,.\,(\mathrm{I};\mathrm{X}[\alpha\mathrm{D}]))) \in C(\{\mathrm{D}\}\times\{\mathrm{D}\}).$$

5. Conclusion

In this paper, we have presented a foundation for logic, set theory and programming. We have obviously not proved any new result (we have not proved any result at all!). Our only (limited) goal was to present this material in a certain style, characterized by a high degree of formalism together with a number of English explanations at the same time. We think that such a style, where the extremes are brought together, is perhaps indispensable for presenting this kind of work.

In being *completely formal* (Bourbaki 1970) we had in mind the possible mechanization of that part of proof writing that is *completely clerical* and thus extremely tedious and error prone. In fact, we have very serious reasons (Milner, this symposium; Abrial 1984) to believe that a mechanized 'proof assistant' (PA) already exists, or can be easily constructed, and that *all* (missing) proofs in this paper can be written with it.

We now discuss some requirements for the proof assistant. It should first offer a series of housekeeping facilities to allow an interactive user to enter new symbols (their arity, their priority), to create new theories, to enter (and check for syntax correctness) new *basic* deduction or definition rules, and finally to retrieve the material just entered. In this respect, it would only mechanize what has been done (and checked!) by hand in this paper.

A second series of facilities concerns the entering of a new *derived* deduction or definition rule. In fact, to accept such a rule, PA requires a proof. It is, however, able to assist the user in accomplishing this unavoidable task (note that a rule, once accepted by PA, can be used like any other already-entered rule).

In operating the first facilities of this second series, the user provides the proof assistant with a set of formulae together with an existing deduction rule. It is then able to automatically generate the consequent of each instance (see §1 (*c*)) of the given rule, having the given set of formulae as an antecedent. This facility corresponds to the most elementary step taken in the making of a formal proof. A similar facility could be offered for definition rules; upon providing PA with a set of antecedent formulae together with a definition rule and a formula to be transformed, it is able to automatically generate all possible transformations of the given formula compatible with instances of the given rule that have the given set of formulae as an antecedent.

Other modes of operation are mere variants of the two previous facilities. In operating these variants, we provide the proof assistant with *less and less information*. For instance, the user could give only the name of a theory instead of that of a rule. It would then be required to try all possible rules in this theory (experience shows that if theories are not too big, a dozen rules or so, then some elementary criterions easily discard most non-pertinent rules). Another (separate) variant would consist in providing the proof assistant with more formulae than strictly necessary to build a possible antecedent. It would then be required to choose among these candidates, the formulae which would fit the given rule. A third variant uses both previous variants together. A fourth variant would consist of applying repeatedly all definition rules (backwards or forwards) of a theory until no rule is applicable (This variant is particularly useful for a change of variables).

As can be seen, the proof assistant, although 'artificial', is not particularly 'intelligent' (sometimes it can even prove to be a little too enthusiastic in expressing its services!). We think, nevertheless, that it can be of some help in constructing proofs, thus in *constructing programs*.

In this respect, the important thing to keep in mind is that definition rules are able to work on *both sides*. Almost all rules of the theory of programming (§3) are definition rules; when applying them from left to right, we assign meanings to 'programs' (Floyd 1967). However, and perhaps more important, when we apply them from right to left, we also assign 'programs' to meanings.

I am particularly indebted to Tony Hoare; many ideas in this paper are obviously his. The importance of program construction was made clear to me by Michel Sintzoff. Bernard Sufrin convinced me eventually of the importance of the concept of schema. Countless discussions with Cliff Jones have had a great influence on the content of this paper.

During the last five years, several people have been kind enough to comment on some preparatory works to this paper, among whom are D. Scott, D. Gries, R. Milne, R. Burstall, S. Schuman, I. Sorensen, T. Clement, L. Morris, M. Goldsmith, B. Meyer, A. Guillon, C. Morgan, J. C. Shepherdson, as well as some other anonymous commentators. Their remarks have all been studied carefully and profitably.

Last, I warmly thank Cathy de Rudder for her remarkable typing (and patience). This paper could never have been finished on time without her.

Appendix 1. Well-formedness

Well formed predicates

$\vdash \mathbf{pred}(\mathrm{P}[\mathrm{S}/\mathrm{X}])$
$\vdash \mathbf{pred}(\sim \mathrm{P})$
$\vdash \mathbf{pred}(\mathrm{P} \vee \mathrm{Q})$
$\vdash \mathbf{pred}(E\mathrm{X}\,.\,\mathrm{P})$
$\vdash \mathbf{pred}(\mathrm{S} = \mathrm{T})$
$\vdash \mathbf{pred}(\mathrm{S} \in \mathrm{T})$
$\vdash \mathbf{pred}(\sigma \mathrm{D})$
$\vdash \mathbf{pred}(\mathrm{D} \pi \mathrm{I})$

Well formed terms

$\vdash \mathbf{term}(\mathrm{T}[\mathrm{S}/\mathrm{X}])$
$\vdash \mathbf{term}(\mathrm{X})$
$\vdash \mathbf{term}(\mathrm{S},\mathrm{T})$
$\vdash \mathbf{term}\{\mathrm{T}|\mathrm{D}\}$
$\vdash \mathbf{term}(P\mathrm{S})$
$\vdash \mathbf{term}(\varnothing)$
$\vdash \mathbf{term}(\mathrm{S}[\mathrm{T}])$
$\vdash \mathbf{term}(U)$
$\vdash \mathbf{term}(\tau \mathrm{S})$

Well formed variables

$\vdash \mathbf{vrbl}(A\hat{\ }B)$

$X\backslash Y \vdash \mathbf{vrbl}(X,Y)$

$\vdash \mathbf{vrbl}(\alpha D)$

$\vdash \mathbf{vrbl}(X')$

Well formed declarations

$X\backslash T \vdash \mathbf{decl}(X:T)$

$\alpha E\backslash D \quad \alpha D\backslash E \vdash \mathbf{decl}(D;E)$

$\vdash \mathbf{decl}(D|P)$

$\vdash \mathbf{decl}(\mathbf{skip})$

$\alpha D'\backslash D \vdash \mathbf{decl}(D')$

$\vdash \mathbf{decl}(D[X/\alpha(E;D;F)])$

Well formed instructions

$\vdash \mathbf{inst}(I[S/X])$

$\vdash \mathbf{inst}(P \rightarrow I)$

$\vdash \mathbf{inst}(I;J)$

$\vdash \mathbf{inst}(I \,\Box\, J)$

$\vdash \mathbf{inst}(\mathbf{loc}(D) \,.\, I)$

$\vdash \mathbf{inst}(\mathbf{skip})$

$\vdash \mathbf{inst}(X \leftarrow T)$

$\vdash \mathbf{inst}(S[X])$

$\vdash \mathbf{inst}((S[X]) \,\|\, (T[Y]))$

$\vdash \mathbf{inst}(\mathbf{rec}(X) \,.\, I)$

$\vdash \mathbf{inst}(\mathbf{loop}(I))$

$\vdash \mathbf{inst}(I[J/X])$

$\vdash \mathbf{inst}(I[N \downarrow (J/X)])$

$\vdash \mathbf{inst}(I \uparrow N)$

$\vdash \mathbf{inst}(\mathbf{break})$

References

Abrial, J. R. 1984 The mathematical construction of a program. In *Science of computer programming*, p. 4.

Bjørner, D. & Jones, C. B. 1982 *Formal specification and software development*. Englewood Cliffs, N.J.: Prentice-Hall.

Bourbaki, N. 1970 *Théorie des ensembles*. Paris: Hermann.

Burstall, R. M. 1969 Proving properties of programs by structural induction. *Computer J.* **12**, 41–48.

Dijkstra, E. W. 1975 Guarded commands, non-determinacy and formal derivation of programs. *Communs Ass. comput. Mach.* **18** (8).

Floyd, R. W. 1967 Assigning meanings to programs. In *Proc. Symp. Appl. Math.* (ed. J. T. Schwartz), vol. 19, pp. 19–31. Providence, Rhode Island: American Mathematical Society.

Jones, C. B. 1980 *Software development – a rigorous approach*. Englewood Cliffs, N.J.: Prentice-Hall.

Kleene, S. C. 1952 Introduction to meta-mathematics. New York: Van Nostrand.

Morgan, C. & Sufrin, B. 1984 Specification of the UNIX filing system. *IEEE Trans. software Engng* (In the press.)

Park, D. 1969 Fixpoint induction and proofs of program properties. *Mach. Intell.* **5**. Edinburgh University Press.

Scott, D. & Strachey, C. 1970 Towards a mathematical semantics for computer languages. *Tech. Monogr.* PRC-6. Oxford University Press.

Tarski, A. 1955 A lattice-theoretical fixpoint theorem and its application. *Pacif. J. Math.*

Discussion

J. C. Shepherdson (*School of Mathematics, University of Bristol, U.K.*). Is there not a danger when one makes one's first specification in a completely formal language that one may make a mistake in the very first step, that of translating the informal everyday language specification of a program into this language? Is it not safer to proceed in stages, first expressing the specification in a very rich and friendly language?

J. R. Abrial. I certainly agree with Professor Shepherdson that the brutal *translation* of our first specification written in the 'informal every day language' into another one written in a 'completely formal language' is obviously a mistake, and that it is certainly safer to first express our specification in a 'very rich and friendly language'.

However, I do not think that we have to reason in terms of 'translation' from one language, be it 'everyday' or 'very rich and friendly', into another 'completely formal' one. The term 'translation' is too narrow. What we have to do is to obtain a *mathematical representation* of our informally stated problem, and such a representation must be easily understandable. Such a desirable mathematical representation is certainly not the result of a translation process; rather it is the result of deep understanding of the mathematical properties of our problem.

Obviously we have to learn how to write such mathematical representations. For instance, I do not think that predicate calculus is the best tool for such an activity. Far better, in my opinion, is to place ourselves directly within the realm of set theory, where such concepts as functions and relations and their operations are available, and where such mathematical objects as natural numbers, sequences and trees and their operations are usable. Obviously such concepts and objects offer possibilities that are very 'rich' and 'formal' at the same time (i.e. we can formally prove properties of mathematical representations written with such tools).

In fact Professor Shepherdson's question raises an important point. For many years, the issue of language (programming, specification, query, relational, natural, etc.) and that of translation has hidden what is, in my opinion, the real problem; namely that of aproaching the activity of program construction from a mathematical point of view.

J. S. Hillmore (*Polytechnic of North London Computing Service, London, U.K.*). Could Mr Abrial act as a spokesman both for himself and for the previous speakers, and explain the relevance of this theoretical work to those of us who write real programs and also to those of us who teach others to write real programs?

J. R. Abrial. Sorry, I can only act as a spokesman for myself. Firstly, I must say that the subject of this symposium was not the writing of real programs nor the teaching of how to write such programs; it is, as you know: 'Mathematical Logic and Programming Languages'. As a consequence, I felt that I had to concentrate mostly on the main subject, although I have agreed with you for many years that the questions you raise are of utmost importance.

In my paper, I have tried to embed a certain programming notation within a completely formal treatment of logic and set theory. In other words, the thesis of my paper is that programming is nothing but a certain part of logic and set theory rewritten in a convenient way.

This *theoretical* thesis has very important consequences on the *practical* activities you mention.

It means, for instance, that we could teach future programmers mathematical logic and some part of axiomatic set theory as a prerequisite for programming courses. In one way or another we have to teach students how to *reason rigorously while constructing their programs*. Excellent textbooks, such as Jones (1980) or Gries (1981), are already available to help support such teachings.

I am convinced that real programming will never become a mature technical activity unless real programmers use a genuinely mathematical approach to construct their programs. It may take some time for such ideas to spread among the software community.

Reference

Gries, D. 1981 *The science of programming*. Berlin: Springer.

R. L. Constable (*Computer Science Department*, *Edinburgh University*, *U.K.*). Mr Abrial's program of research is quite similar to that of de Bruijn's AUTOMATH project, begun in 1968, and to that of Edinburgh LCF, begun in 1975, and to FOL at Stanford. For instance, LCF has a formalized account of substitution, which is then used in building new user-defined proof rules, as Robin Milner explained earlier in this symposium. Mr Abrial's ideas for a theory of programs and data types seem quite similar to those in the Cornell work on programming logics, *ca.* 1978. All of these projects already have built computer systems to help with the details of proof construction. In what ways do Mr Abrial's ideas go beyond those in the established projects? In what way is his set theory superior to that defined in AUTOMATH, which provides an explicit theory of functions (as typed lambda terms) in each of its theories?

J. R. Abrial. I must admit that for some reason I have not been able to become familiar with the work you mention. As a consequence, it is quite probable that the ideas in my paper do not go beyond those in established projects that started so long ago, and that the set theory presented is in no way superior to that defined in AUTOMATH.

My main sources of inspiration were Bourbaki's treatise on set theory, Hoare's work on the axiomatization of programming languages and Dijkstra's '**do**...**od**' non-deterministic language.