Draft 1, 2 July 2015
This is the short version, omitting proofs and many discussions.

Theory of Programs

Bertrand Meyer

"Computer science" (informatics) is really *program* science since a computer, by itself too general a machine to be of practical interest, yields useful machines through programs that people write for it. While the theoretical study of programs fills volumes, few people realize that a handful of concepts from elementary set theory suffice to establish a clear and practical basis.

Among the results:

- To describe a specification or a program, it suffices to define one relation and one set.
- To describe the concepts of programming, concurrent as well as sequential, three elementary operations on sets and relations suffice: union, composition and restriction.
- These techniques suffice to derive the axioms of classic papers on the "laws of programming" as straightforward consequences.
- To define both program correctness and refinement, the ordinary subset operator "$\subseteq$" suffices.

Paragraphs labeled "*Intuition*" relate the concepts to the experience of readers having done some programming. Readers with knowledge of previous views of theoretical informatics will find comparisons in "*Comment*" paragraphs. Section 5 provides more discussion.

1 Programs

A program is a simple mathematical object: a *constrained relation over a set of states*.

Definition: program, specification, precondition, postcondition.

A **program**, also known as a **specification**, over a state set S, consists of:
- A relation $post$: $S \leftrightarrow S$, the program's **postcondition**.
- A set $Pre \subseteq S$, the program's **precondition**.

Notation: $A \leftrightarrow B$ is the set of binary relations between A and B, that is, $\mathbb{P}(A \times B)$. The domain of a relation r is written $\underline{r}$ and its range $\overline{r}$.

Intuition: a program starts from a certain state and produces one of a set of possible states satisfying properties represented by $post$. Pre tells us which states are acceptable as initial states.

In the general case, more than one resulting state can meet the expectation expressed by $post$. Correspondingly, $post$ is a relation rather than just a function.

The definition covers continuously running programs, such as those embedded in devices, since they are just repetitions of individual state transformations. Particular choices for S and for acceptable $post$ and Pre determine particular styles of programming, such as the following.

Definition: deterministic, functional, imperative, object-oriented, object, procedural

A program p is:

- **Deterministic** if $post_p$ is a function, and **non-deterministic** otherwise.
- **Functional** if every subset C of S is disjoint from $post_p\ (C)$, and **imperative** otherwise.
- **Object-oriented** if S is of the form $0\,..\,n \to O$ for an integer n and a set O of "**objects**", and **procedural** otherwise.

Notation: For a relation r in $A \leftrightarrow B$ and subsets X and Y of A and B respectively, $r\ (X)$ denotes the image of X, and $r^{-1}\ (Y)$ the reverse image of Y, by r. The relation is a "function" (short for "possibly partial function") if $r\ (\{x\})$, for any element x of A, has at most one element. If it always has one, r is "total". $A \to B$ is the subset of $A \leftrightarrow B$ containing total functions only. An integer interval is written $m..n$. Section 4.2 will present a more elaborate structure for S in which the above characterizations apply to the "store" part.

S_p, Pre_p *and* $post_p$ are the state set, precondition and postcondition of a program p. In addition, discussions of an indexed set of programs p_i will use S_i, Pre_i and $post_i$ for the i-th program.

The principal concepts of programming, studied in the rest of this presentation, are independent of such choices of style and of the properties of S.

Definition: feasibility

A program p is **feasible** if $Pre_p \subseteq \underline{post_p}$.

Intuition: Pre_p tells us when we *may* apply the program, and $post_p$ what kind of result it *must* then give us. A program/specification is safe for us to use if it meets its obligation whenever we meet ours. Feasibility expresses this property: for any input state satisfying Pre_p, at least one output state satisfies $post_p$.

Definition: program equality

Two programs are **equal** if they have the same Pre and the same $post$ / Pre.

Notation: For a relation r and subsets X and Y of its source and target sets, $r\ /\ X$ and $r \setminus Y$ are r restricted to domain X (meaning $r \cap (X \times S)$) and corestricted to codomain Y (meaning $r \cap (S \times Y)$). Two straightforward properties (restriction and corestriction theorems) are that $\underline{r\ /\ X} \subseteq X$ and $\overline{r \setminus Y} \subseteq Y$.

Comment: while it is customary to distinguish between programs and specifications, all definitions of the purported difference are vague, for example that a specification describes the "what" and a program the "how". The reason for the vagueness is that the difference does not exist. It is impossible to assign a given artifact solely to one of the two categories. An assignment instruction is implementation to the application programmer and specification to the compiler writer. (See also section 3.) Any useful notion has to be *relative*: artifact 1 "specifies" artifact 2.

Definition: refines, specifies, abstracts

A program/specification p_2 **refines** another, p_1, and p_1 **specifies** (or **abstracts**) p_2, if:

P1	S_2	$\supseteq$	S_1	-- Extension
P2	Pre_2	$\supseteq$	Pre_1	-- Weakening
P3	$post_2$	$\subseteq_{Pre_1}$	$post_1$	-- Strengthening

Notation: $r \underset{X}{\subseteq} r'$ means $(r / X) \subseteq r'$; in other words, whenever r maps an element of X to a result, r' maps it to the same result. The same conventions applies to other operators on relations, as in $r \underset{X}{=} r'$. Note the names (extension, weakening, strengthening) associated with the three conditions of the definition.

Intuition: a refinement of p gives more detail than p, but still satisfies all properties of p relevant to users of p. So it must cover all of p's states, accept all the input states p accepts and, for these states, only yield results that p could also yield. It may have more states, a more tolerant precondition, and yield only some of the results that p could yield (reduce non-determinism).

Theorem: Refinement Theorem

P4	Refinement is an order relation.

Definition: implementation

An **implementation** of p is a feasible refinement of p.

Intuition: not every refinement of a specification is feasible. For example the infeasible program having the empty relation as its postcondition and S as its precondition refines every specification over S. Hence the importance of finding feasible refinements, also known as implementations. This concept still does not provide a distinction between programs and specifications.

Theorem: Implementation Theorem

P5	A specification/program having an implementation is feasible.

2 Operations on specifications and programs

The fundamental operations of elementary set theory yield fundamental operations on specifications and programs. Only the first three (those of 2.1) refer directly to the basic concepts defined so far; all the rest follow as combinations of these three.

2.1 Basic constructs

Definition: choice, composition, restriction

Name	Notation	Mathematical definition		Programming intuition
		Postcondition	Precondition	
Choice (or: union)	$p_1 \cup p_2$ (Dijkstra: $p_1 \,[]\, p_2$)	$post_1 \cup post_2$	$Pre_1 \cup Pre_2$	Performs like p_1 or like p_2
Composition (or: sequence, compound, block)	$p_1 \,;\, p_2$	$(post_1 \setminus Pre_2) \,;\, post_2$	$Pre_1 \cap post_1^{-1}\,(Pre_2)$	Performs first like p_1 then like p_2
Restriction (guarded command)	$C : p$ (Dijkstra: $C \to p$)	$post_p / C$	Pre_p	Performs like p on C

Notation: In the "postcondition" column, the semicolon ";" denotes composition of functions or relations, in the order of application, so that $(r \,;\, s)\,(X)$ is $s\,(r\,(X))$. (Mathematical texts often use $s \circ r$ for $r \,;\, s$.) "Dijkstra" means the notation of [3].

For a known set of states S, $<post, Pre>$ is the program of postcondition *post* and precondition *Pre*.

Theorem

P6	For feasible operands and arbitrary conditions, the above operators yield feasible programs.

Theorems

Choice, like union of sets, is commutative; composition of programs, like composition of relations, is not. Choice and composition are associative, so we may apply them without parentheses to any number of operands, as in $p_1 ; p_2 ; \ldots ; p_n$. In addition:

P7	$C_1 : (C_2 : p)$	$= C_2 : (C_1 : p)$	-- Restriction is commutative. In fact:
P8	$C_1 : (C_2 : p)$	$= (C_1 \cap C_2) : p$	
P9	$C : (p_1 \cup p_2)$	$= (C : p_1) \cup (C : p_2)$	-- Restriction distributes over choice.
P10	$C : (p_1 ; p_2)$	$= (C : p_1) ; p_2$	-- Composition absorbs restriction.
P11	$q ; (p_1 \cup p_2)$	$= (q ; p_1) \cup (q ; p_2)$	-- Composition distributes left…
P12	$(p_1 \cup p_2) ; q$	$= (p_1 ; q) \cup (p_2 ; q)$	-- … and right over choice.

The following programs are of interest, all of them feasible, the first two total: *Skip*, the identity over S, with postcondition $\lambda x : S \mid \{x\}$ (always applicable, changes nothing); *Havoc*, with postcondition $S \times S$ (always applicable, but we may not assume anything about the result); and *Halt*, defined as $<\varnothing, \varnothing>$ (empty relation as postcondition and, for feasibility, empty set as precondition).

Notation: generalized lambda notation serves to define relations in $A \leftrightarrow B$, using either $\lambda x : A \mid Y$ where Y is a subset of B (as here for *Skip*), or $\lambda x_1 : A; x_2 : B \mid p\ (x_1, x_2)$ where p is a two-variable predicate. A program/specification is **total** if its precondition is S.

Theorems

P13	$(p ; Skip)$	$= (Skip ; p)$	$= p$	
P14	$(p \cup Halt)$	$= (Halt \cup p)$	$= p$	-- Does not hold in the demonic theory.
P15	$(Halt ; p)$	$= (p ; Halt)$	$= Halt$	
P16	$(p \cup Havoc)$	$= (Havoc \cup p)$	$= Havoc$	
P17	$(p ; Havoc)$	$= (Pre_p : Havoc)$		
P18	p	$\subseteq (C : p)$		-- (Reminder: $\subseteq$ on programs is refinement.)
P19	If $D \subseteq C$, then $(C : p) \subseteq (D : p)$.			-- Order reversal (precondition weakening).
P20	If $q \subseteq p$, then $(C : q) \subseteq (C : p)$.			-- Refinement safety, see below.
P21	If $q_1 \subseteq p_1$ and $q_2 \subseteq p_2$, then $(q_1 \cup q_2) \subseteq (p_1 \cup p2)$ and $(q_1 ; q_2) \subseteq (p_1 ; p_2)$.			
P22	$p \subseteq (Pre_p : Havoc)$ for any p.			
P23	$p \subseteq Havoc$ for any total p.			
P24	$p \subseteq Halt$ if and only if $p = Halt$			-- *Halt* is refined only by itself…
P25	$Halt \subseteq p$ if and only if $p = Halt$			-- … and refines only itself.

Notation: "$\cup$" for choice is a new example (after "$\subseteq$" for refinement) of extending set operators to programs. The following application of this idea is also useful:

Name	Notation	Postcondition	Precondition	Programming intuition
Corestriction	$p \setminus C$	$post_p \setminus C$	$Pre_p \cap post_p^{-1}(C)$	p, applied only when results satisfy C

(There is no need for a restriction notation p / C since we already have $C: p$.) The first of the following properties shows that corestriction can be defined from restriction and composition.

Theorems

P26	$(p \setminus C)$	$=$	$(p ; (C: Skip))$	
P27	$(p_1 \cup p_2) \setminus C$	$=$	$(p_1 \setminus C) \cup (p_2 \setminus C)$	-- Compare with P9.
P28	$(p \setminus C)$	$\subseteq$	C	-- Refinement. Compare with P18.
P29	If $D \subseteq C$, then $(p \setminus D)$	$\subseteq$	$(p \setminus C)$	-- Compare with P19.

The restriction and corestriction theorems apply to programs: $\underline{C: p} \subseteq C$ and $\overline{p \setminus C} \subseteq C$.

Notation: in the same spirit, the range and domain notations apply to programs: $\underline{p}$ is a synonym for Pre_p; and (more importantly) $\overline{p}$ is a synonym for $post_p\,(\underline{p})$, the set of values that p can actually yield.

Definition: refinement safety

An operator § on programs is **refinement-safe** if $q_1 \subseteq p_1$ and $q_2 \subseteq p_2$ implies $(q_1 \,§\, q_2) \subseteq (p_1 \,§\, p_2)$.

Theorem

P30 All the operators on programs introduced in this article are refinement-safe.

In a corresponding sense, the program properties "functional" and "object-oriented" are refinement-safe (but not their contraries, "imperative" and "procedural").

2.2 Atomic concurrency

Definition: concurrency

Name	Notation	Definition	Programming intuition
Atomic concurrency	$p_1 \parallel p_2$	$(p_1 ; p_2) \cup (p_2 ; p_1)$	Performs once like each of p_1 and p_2

Theorems: concurrency is commutative, associative and refinement-safe. In addition:

P31	$p_1 \parallel (p_2 \cup p_3)$	$=$	$(p_1 \parallel p_2) \cup (p_1 \parallel p_3)$	-- Concurrency distributes over choice, left…
P32	$(p_1 \cup p_2) \parallel p_3$	$=$	$(p_1 \parallel p_3) \cup (p_2 \parallel p_3)$	-- … and right.
P33	$C: (p_1 \parallel p_2)$	$=$	$(C: p_1) \parallel (C: p_2)$	-- Restriction distributes over concurrency…
P34	$(p_1 \parallel p_2) \setminus C$	$=$	$(p_1 \setminus C) \parallel (p_2 \setminus C)$	-- … and so does corestriction.
P35	$(p_1 ; p_2)$	$\subseteq$	$(p_1 \parallel p_2)$	-- Sequential composition refines concurrency…
P36	$(p_2 ; p_1)$	$\subseteq$	$(p_1 \parallel p_2)$	-- … in any order.

Concurrency generally does not refine composition, but in one particular case it does.

Definition: commuting programs

Two specifications/programs commute if $(p_1 ; p_2) = (p_2 ; p_1)$.

Example and counter-example: if S is the set of functions $PERSON \to \mathbb{Z}$, recording people's bank account balances, consider an infinite set of programs, defined for any person p and any integer n: the postcondition of $deposit_{p,n}$ expresses that the output differs from the input only by having the balance of p increased by n, and similarly for $withdraw_{p,n}$. All these programs commute with each other. They do not commute, however, with the program $reset_p$ setting p's balance to zero.

Theorem

P37 If p_1 and p_2 commute, then $(p_1 \parallel p_2) = (p_1 ; p_2)$.

(Not just refinement, but equality. Immediate generalization to more than two programs.)

Intuition: commuting programs are a boon for concurrent computation, since they open up many possible realizations for "computing" program results (finding values satisfying $post_p$) on actual "computers" (the physical devices that ensure postconditions). Assume for example a large number of *deposit* and *withdraw* operations with various clients and amounts. If the specification is that at the end of the trading day the balance of each should be correct, any assignment of the operations among any number of computers in any order is suitable.

Theorem

P38 For deterministic programs with identical preconditions, refinement and abstraction preserve commuting.

2.3 Non-atomic concurrency

The atomic concurrency operator has a fixed level of granularity, defined by its operands: if they are themselves complex programs built out of simpler components, it will not interleave these components. For example let *on* be "switch on the light", *off* "switch it off" and p "say whether the light is on". Assuming that in the initial state the light is on, $(on ; off) \parallel p$ will always say no, regardless of which of the operands of "$\parallel$" goes first, since $(on ; off)$ is equal to *Skip*.

The practice of concurrency often calls for finer-grain control on concurrency. Here you might want p to execute at the beginning, in the middle (between *on* and *off*), or at the end. Such flexibility causes much of the difficulty of concurrent programming, since it opens up the possibility of "data races" (inconsistent orderings of operations, in some executions only); but a general theory of programming must provide a model for it, given here by a ternary operator.

Name	Notation	Definition	Programming intuition
Non-atomic concurrency	$(p_1 , p_2) \parallel q$	$((p_1 \parallel q) ; p_2) \cup$ $(p_1 ; (p_2 \parallel q))$	Performs once like each operand, with p_1 before p_2

Notation: the only new symbol is the comma, used at a place where the semicolon of composition could also appear. The reuse of "$\parallel$" is only for convenience: the "Notation" entry describes a new three-operand operator. Its "Definition" entry relies on the previously defined atomic concurrency operator "$\parallel$". No confusion arises since the non-atomic operator only occurs in conjunction with the comma.

Non-atomic concurrency is associative on its first two operands p_1 and p_2, so you may use commas to separate any number of program operands of non-atomic concurrency.

Theorems

P39	$(p_1 , p_2) \parallel q$	$= (q ; p_1 ; p_2) \cup (p_1 ; q ; p_2) \cup (p_1 ; p_2 ; q)$	
P40	$(p_1 ; p_2) \parallel q$	$\subseteq (p_1 , p_2) \parallel q$	-- Coarser-grained refines finer-grained.
P41	$p_1 ; (p_2 \parallel q)$	$\subseteq (p_1, p_2) \parallel q$	-- First "law of exchange" of [8].
P42	$(p \parallel q_1) ; q_2$	$\subseteq p \parallel (q_1, q_2)$	-- Second "law of exchange" of [8].

It is straightforward to symmetrize the non-atomic concurrency notation to $(p_1 , p_2) \parallel (q_1 , q_2)$, yielding the generalized law of exchange from [8]: $(p_1 \parallel q_1) ; (p_2 \parallel q_2) \subseteq (p_1 , p_2) \parallel (q_1 , q_2)$.

2.4 Conditionals

Definition: conditionals

Name	Notation	Definition	Programming intuition
Guarded conditional	**if** C_1: p_1 [] C_2: p_2 **end**	$(C_1: p_1) \cup (C_2: p_2)$	Performs like p_1 on C_1, like p_2 on C_2
If-then-else	**if** C **then** p_1 **else** p_2 **end**	$(C: p_1) \cup (C': p_2)$	Performs like p_1 on C, like p_2 elsewhere

Notation: C', for a subset C of S, is its complement: $S - C$. The usual programming notation is "**not** C" (see 2.5 below). The guarded conditional is in fact not new since **if** p_1 [] p_2 **end** was introduced in 2.1 as a synonym for $p_1 \cup p_2$, but it highlights the important case of p_1 and p_1 being restrictions.

Theorems: the guarded conditional is commutative; the corresponding property for if-then else is that (**if** C **then** p_1 **else** p_2 **end**) = (**if** C' **then** p_2 **else** p_1 **end**). Both operators are associative; as a consequence they can be applied to more than two operands (if-then-else uses **elseif** for the second to next-to-last branches, as in **if** C_1 **then** p_1 **elseif** C_2 **then** p_2 **else** p_3 **end**), and to just one: for the guarded conditional, **if** C: p **end** is the same as C: p; for if-then-else, by convention, **if** C **then** p **end** is an abbreviation for **if** C **then** p **else** *Skip* **end**.

Theorems: both forms are distributive over choice and concurrency, but not over composition. The guarded conditional is commutative, but not if-then-else. In addition:

P43	If $D_1 \subseteq C_1$ and $D_2 \subseteq C_2$, then (**if** D_1: p [] D_2: q **end**)	$\subseteq$	(**if** C_1: p [] C_2: q **end**).	
P44	If $q_1 \subseteq p_1$ and $q_2 \subseteq p_2$, then (**if** C: q_1 [] C: q_2 **end**)	$\subseteq$	(**if** C: p_1 [] C: p_2 **end**).	
P45	If $q_1 \subseteq p_1$ and $q_2 \subseteq p_2$, then (**if** C **then** q_1 **else** q_2 **end**)	$\subseteq$	(**if** C **then** p_1 **else** p_2 **end**).	
P46	(C: p)	=	(**if** C: p **end**)	
P47	(**if** C_1: p_1 [] C_2: p_2 **end**)	$\subseteq$	C_1: p_1	-- A conditional refines any of its branches
P48	(D: (**if** C_1: p [] C_2: q **end**))	=	(**if** ($D \cap C_1$): p [] ($D \cap C_2$): q **end**)	-- Distributivity.
P49	(**if** C **then** p_1 **else** p_2 **end**)	=	(**if** C: p_1 [] C': p_2 **end**)	
P50	(**if** C **then** p_1 **else** p_2 **end**)	=	(**if** C' **then** p_2 **else** p_1 **end**)	

2.5 Conditions

Two special conditions are useful for building programs. *True* is another name for *S*, and *False* another name for the empty set. They should not be confused with the similarly named constants of propositional calculus: *True* and *False* are, like all conditions, sets (subsets of *S*). In fact the theory of programs relies on set theory rather than directly on logic, although it is easy to define boolean-like operators on conditions: **and** and **or** as other names for "$\cap$" and "$\cup$", **not** as another name for complement (in P50 we may write *C'* as **not** *C*), **implies** or "$\Rightarrow$" as other names for "$\subseteq$", and so on. Here, in addition to P19, are some properties involving operations on conditionals.

Theorems

P51	(*True*: *p*)	=	*p*	-- And correspondingly for conditionals.
P52	(*False*: *p*)	=	*Halt*	
P53	*p* \ *True*	=	*p*	-- Here "\" is corestriction on programs.
P54	*p* \ *False*	=	*Halt*	
P55	(**if** *True* **then** p_1 **else** p_2 **end**)	=	p_1	
P56	(**if** *False* **then** p_1 **else** p_2 **end**)	=	p_2	-- And similarly for guarded conditionals.
P57	**and**, **or**, **not**, **implies** distribute over choice, restriction and conditionals.			

2.6 Loop

Definition: repetition constructs

Name	Notation	Definition	Programming intuition
Fixed repetition	p^i for any natural integer i	$p^0 = \underline{p}$: *Skip* $p^{i+1} = (p\ ;\ p^i)$	*p* repeated *i* times
Arbitrary repetition	**loop** *p* **end** (or p^*)	$\bigcup_{i \geq 0} p^i$	*p* repeated any number of times
"While loop"	**from** *a* **until** *C* **loop** *b* **end** (or *a*; **while not** *C* **loop** *b* **end**)	*a* ; (**loop** *C'*: *b* **end**) \ *C* or equivalently: $a\ ;\ (\bigcup_{i \geq 0} (C' : b)^i \setminus C)$	*a*, then *p* repeated until *C* holds

Notation: in the second definition of the while loop, it does not matter how we parenthesize the "\"; see P27. Since composition is associative, the inductive expression for fixed repetition can also be written $(p^i ; p)$.

Intuition: **loop** *p* **end** is the program that performs like *p* repeated some finite (but unknown) number of times. Cyclic programs, such as those on embedded devices, follow this pattern. The rest of the present discussion concentrates on the **from** *a* **until** *C* **loop** *b* **end** loop, which starts like *a* then performs like *b*, the loop's "body", as many times as needed (possibly zero) until reaching a state satisfying *C*. In slightly different terms: for the loop to yield a result from a given input

state x, that result must be the first element of C reached by successive executions of b after a. All the previous states are not in C, so they are in C', meaning that what we are iterating is not the whole b but just C': b.

From distributivity follows another expression of the loop:

Theorem: Loop Lemma

P58 The loop l = **from** a **until** C **loop** b **end** can be written $\bigcup_{i \geq 0} q_i$, where q_i is $a \ ; \ (C' : b)^i \setminus C)$.
As a consequence, $\overline{l} = \cup \ \overline{q_i}$.

Intuition: q_i represents a restricted version of the loop, which yields a result (satisfying C) after exactly i iterations. The loop is the union of all such partial versions of it.

Comment: the loop does not automatically get feasibility from that of its operands: it is possible for a, b and all q_i to be feasible, while l is not. A loop is feasible if and only if for every suitable state s there exists an integer i (typically not the same for different s) such that $a \ ; \ (C' : b)^i (\{s\})$ contains an element in C. The feasibility condition relies on the notion of invariant.

2.7 Invariants

Definition: invariant

A condition I is an **invariant** of a program/specification p if $post_p (\underline{p} \cap I) \subseteq I$.

Intuition: an invariant is called that way because if it holds before application of p it will hold afterwards. More precisely, for the initial condition we need not the whole of I but just $\underline{p} \cap I$, since results of p only matter when it starts from the precondition. The following two theorems ensue directly from the definition.

Theorems

P59 Any I disjoint from $\underline{p}$ is an invariant of p.
P60 If I is an invariant of p, so is J if $J \subseteq I$.
P61 If I and J are invariants of p, so are $I \cup J$ and $I \cap J$.

Theorem: Invariant Refinement Theorem

P62 If I is an invariant of p_1 and $p_2 \subseteq p_1$, then I is an invariant of $p_2 \ / \ Pre_1$.

Comment: in practice, the precondition often stays the same under refinement, but in the general case p_2 might have a broader precondition; there is no guarantee that the original invariant will hold for the new states, hence the restriction to Pre_1.

Definition: invariant-preserving operator

An operator on programs is **invariant-preserving** if any invariant of all its program operands is also an invariant of the operator's result.

Example: program composition is invariant-preserving.

Theorem: General Invariant Theorem

P63 All the program operators defined so far are invariant-preserving.

Every element of the infinite unions that define loops is made out of basic operators and, by induction, is invariant-preserving. Since union maintains this property, the loops themselves possess it. They benefit, however, from a more specific form of the notion of invariant.

Definition: loop invariant

A **loop invariant** of **from** *a* **until** *C* **loop** *b* **end** is a subset of $\overline{a}$ that is an invariant of *C'*: *b*.

Theorem: Loop Correctness Theorem

P64 If *I* is a loop invariant of the loop *l* = (**from** *a* **until** *C* **loop** *b* **end**), then $\overline{l} \subseteq C \cap I$

Intuition: The theorem characterizes the fundamental property of loops [11] [5]: the goal of a loop is to obtain on exit ($\overline{l}$) a combination of the exit condition (*C*) and a judiciously chosen invariant (*I*, a weakening of the desired result).

Theorem: Loop Feasibility Theorem

P65 For feasible *a* and *b*, the loop **from** *a* **until** *C* **loop** *b* **end** is feasible if both:

- $\underline{b} \cup C$ is a loop invariant.
- *C'*: $post_b$ is well-founded.

Notation: a "well-founded" (or "Noetherian") relation is one that admits no infinite chain.

Definition: loop variant

A **loop variant** of **from** *a* **until** *C* **loop** *b* **end** is a total function *v* from *S* to a set *V* equipped with a well-founded relation "<", such that $v\ (post_b\ (s)) < v\ (s)$ for any *s* in *C'*.

(Strictly speaking, *v* only needs to be total on $(\cup\ \overline{q_i}) \cup (C' \cap (\cup\ \underline{q_i}))$.) The existence of a variant shows that $post_b$ itself is well-founded, fulfilling the second condition of the Loop Feasibility Theorem. The most frequent choice for *V* is the set of natural integers.

3 Contracted programs

There is, as noted, no difference of principle between specifications and programs. In practice we are used to different connotations for these terms. Since the distinction is so commonly accepted, let us see if we can find a justification serious enough to earn it a place in the theory of programs.

Just like the distinction between abstract and concrete is relative, the distinction between descriptive and executable shifts with the evolution of language and compiler technology. To find a true difference, we must look elsewhere.

The relevant criterion is correctness. As captured by the notion of feasibility, a specification can be inconsistent (if it tells you that the result must be zero and also that it can be one) or consistent; but it makes no sense to ask whether it is correct. Correct with respect to what? Probably

with respect to the customers' desires, or to their actual needs, but these would have to be written down as another, higher-level specification, only pushing the problem further. We do know, however, what it means for a program to be correct: it performs according to its specification. Correctness is a relative notion.

Definition: contracted program, specification part, contract, implementation part, correctness

The notation **require** *Pre* **do** *b* **ensure** *post* **end**, a **contracted program**, states that *i* is an implementation of the specification/program *p* = <*post*, *Pre*>.

Then *p* is the **specification part**, or **contract**, and *b* the **implementation part**.The contracted program is also said to be a **correct program**.

Intuition and comment: the notion of contracted program simply introduces a programming notation for the concept of refinement. Since a program is useless without a precise understanding of what it is supposed to do, program authors should only produce contracted programs. Regrettably, this practice is not yet universal.

The above definition provides a final clarification of what programs in the usual sense of the term (*contracted programs* in the present theory) really are: **a program is a proof obligation**. Writing **require** *Pre* **do** *b* **ensure** *post* **end** is a way to state that *b* must refine *p*, and requires the author, before clicking "Compile", let alone clicking "Run", to click "Verify".

Theorem

P66 If *post* ⊆ *post'*, *Pre'* ⊆ *Pre*, and **require** *Pre* **do** *b* **ensure** *post* **end** is a contracted program, so is **require** *Pre'* **do** *b* **ensure** *post'* **end**.

Comment: in this case, since we keep the implementation and go to a new specification, we can only strengthen the precondition and weaken the postcondition.

The following concepts are defined for given *Pre*, *post* and *b*.

Definitions: weakest precondition, strongest postcondition

$post_b$ / *Pre*, also written *b* **sp** *Pre*, is the **strongest postcondition** of *b* for *Pre*.

$\underline{b} - \underline{post_b - post}$, also written *b* **wp** *post*, is the **weakest precondition** of *b* for *post*.

Intuition: $post_b - post$ is a set difference between two relations, giving us the set of pairs that belong to the first but not to the second. Its domain, $\underline{post_b - post}$, is the set of states for which *b* produces at least one result that *post* could never produce. Subtracting this domain from $\underline{b}$, the domain of *b*, gives us the set of states on which *b* is guaranteed to agree with *post*.

The following property justifies the terms "strongest" and "weakest".

Theorem

P67 If **require** *Pre* **do** *b* **ensure** *post* **end** is a correct program, then (*b* **sp** *Pre*) ⊆ *post* and *Pre* ⊆ (*b* **wp** *post*).

As a corollary, we get a compact definition of program correctness.

Theorem

P68 **require** *Pre* **do** *b* **ensure** *post* **end** is correct if and only if $Pre \subseteq \underline{b} \text{ —} \underline{post_b \text{ —} post}$.

Theorems

P69	b **sp** *False*	=	*Halt*
P70	b **wp** *Halt*	=	*False*
P71	*Halt* **sp** C	=	*Halt*
P72	*Halt* **wp** p	=	*False*
P73	b **sp** $(p \cup q)$	=	$(b \textbf{ sp } p) \cup (b \textbf{ sp } q)$
P74	b **wp** $(p \cup q)$	$\supseteq$	$(b \textbf{ wp } p) \cup (b \textbf{ wp } q)$

Definition: generalizing refinement to contracted programs

If $q \subseteq p$ (q refines p), **require** *Pre* **do** q **ensure** *post* **end** refines **require** *Pre* **do** p **ensure** *post* **end**.

Definition and theorem: Most Abstract Implementation

P75 For feasible p, **require** $\underline{p}$ **do** p **ensure** $post_p$ **end**, the **most abstract implementation** of p, is a correct program, which every implementation of p refines.

Intuition: The most abstract implementation is the specification used as its own implementation.

4 States and environments

The exact nature of S, the state set, varies considerably between application domains and the formalisms supporting programming (*programming languages* as defined next in section 5). Some properties, however, are common to most variants.

4.1 Mappings

The state tracks the evolution, during the computation, of certain elements of information relevant to the results. As a consequence, a state almost always includes (as its essential components) one or more mappings between these elements and their current values. "Mapping" is a general term roughly equivalent to "function"; in programming, since the memories of both humans and computers are finite, these functions will also be finite. S, then, includes components of the form $Name \twoheadrightarrow Value$ for appropriate sets of names and values.

> *Notation*: $A \leftrightarrow B$ is the set of possibly partial functions, and $A \twoheadrightarrow B$ the set of finite functions, from A to B. Inclusions are: $(A \twoheadrightarrow B) \subseteq (A \leftrightarrow B) \subseteq (A \leftrightarrow B)$ and $(A \to B) \subseteq (A \twoheadrightarrow B)$.

4.2 Environment and store

It is common for the state two have two clearly identified components: the environment and the store, also known as the static and dynamic parts. In a simple variant, with a set *Var* (for "variables") of names and a set *Type* representing the types of possible values, the environment is of the form $Var \twoheadrightarrow Type$ and the store of the form $Var \twoheadrightarrow Value$. This division reflects the typical process of executing programs on a computer:

- A first step known as **compilation** creates the environment.
- The actual computation, known as **execution**, takes place in the second step, which builds and transforms the store, constrained by environment built in the first step.

One of the advantages of this approach is that it requires programmers to define types for every variable, making it possible to detect mistakes (such as applying a boolean operation to integer variables) in the first step; in that case the second step does not take place until the programmer has corrected the mistake. Such a process limits the risk of erroneous computation. Another advantage is that it is not necessary to repeat the first step once it has succeeded: subsequent executions of the same program, applied to different input states, can use the result of the compilation.

Definitions: declaration, instruction

A function in $S \twoheadrightarrow S$ is a **declaration** if it leaves the store part unchanged, and an **instruction** if it leaves the environment part unchanged.

Intuition: it is good practice to separate the two kinds of operation; declarations set up the environment; instructions, working in a defined environment, change only the store.

Comment: the characterization of programming styles (functional, object-oriented) in section 1 properly applies to the store component of the state.

5 Languages and programming

"Programming" is the act of writing correct programs according to the preceding definitions. Such a program has two parts: the contract represents the goal of the program, as advertised to its users; the implementation represents the operations that will run on the computer. The definition ensures that the implementation matches the contract.

5.1 Programming languages

If the contract is given, in the form of *Pre* and *post*, programming consists of solving **require** *Pre* **do** *b* **ensure** *post* **end**, viewed as an equation of which *b* is the unknown.

The Most Abstract Implementation, as defined above, yields a trivial solution, often non-deterministic, to the equation: $post_b = post \,/\, Pre$, $Pre_b = Pre$. The reason why that solution is generally of little use, and programming an interesting endeavor, is the *practical* difference between contract and implementation. For *b* we seek a relation $post_b$ that a material computer can process (not necessarily directly, but through the services of tools such as "compilers"). For the specification, since the goal is to describe the problem, we can rely on a broader set of mathematical mechanisms.

In both cases we need a repertoire of mathematical tools to build programs and specifications.

Definition: programming language, specification language

A **programming language** over a state set S, also known as a **specification language** over S, is a set of feasible programs over S. In practice it is given by:.

- A finite set of base programs, obtained from a finite set of relations in $S \leftrightarrow S$ (serving as base postconditions) and a finite set of subsets of S (serving as base preconditions).
- A finite set of operators for deriving new correct programs from previously defined ones.

Intuition: a programming language is a set of possible programs. Any useful programming language is infinite, but it is derived from a few basic postconditions and preconditions, and a few operators to combine them. Many of these basic elements, introduced in the earlier sections of this presentation, can be used by programming languages regardless of the application domain:

- *Havoc*, *Skip* and *Halt* as base programs, with *True* and *False* (S and $\varnothing$) as base preconditions.
- The program construction operators of section 2, including the three basic ones (choice, composition and restriction) and those derived from them (concurrency, conditionals, loops).

Beyond these universal elements, a language will offer specific mechanisms for the intended application domain, beginning with a suitable set S of states and a suitable set of operations over S.

With a single S and a single specification and programming language, the language description will identify, among the language's mechanism, the subset suitable for implementation. Then the requirement on program authors is simply to produce a final version **require** *Pre* **do** *b* **ensure** *post* **end** of the program in which the implementation part b only relies on that subset. Establishing correctness means establishing:

- Refinement: $b \subseteq \langle post, Pre \rangle$.
- Feasibility: $Pre \subseteq post$ (or alternatively, thanks to the implementation theorem P5, $Pre_b \subseteq post_b$ if the preceding condition holds).

One can express these properties convincingly, and prove them, since all three components, *post*, *Pre* and b, are part of the same mathematical framework, even if the last one restricts itself to a subset of that framework's mechanisms.

5.2 Approaches to programming

The most common approach to programming today ignores the *Pre* and *post* elements of the definition, concentrating only on building implementations b from a programming language with the hope that in some informal sense they will match the corresponding user needs. We may call this the "hacking approach"; it has little to commend itself if correctness is part of the objectives.

At the other extreme, a "refinement approach" [17] [1] [15] has made its mark in informatics research and led to such developments methods as B. If we set out to implement a given contract, the Most Abstract Implementation theorem P75 tells us that we may use the contract itself — specifically, $\langle post, Pre \rangle$ — as its own first implementation. Refinement as a software development

method starts with this first version and repeatedly takes advantage of theorems to choose a "refinement" in the sense of the formal definition, P2 and P3, of the previous implementation until reaching an implementation that belongs to the implementation part of the language.

The ideal process should combine the best elements of the "hacking" and "refinement" approaches, retaining the practicality of the first and the rigor of the second. It is not the goal of the present discussion to present such a process, but a general definition helps set the stage.

Definition: programming

Programming is the process of devising interesting contract-implementation pairs and discharging the associated proof obligations.

The starting point for any step in the process may indifferently be:

- A contract element, for which we have to devise a satisfactory implementation (top-down).
- Existing implementation elements (bottom-up). Ideally these elements already have full contracts. In practice, they often have no contracts, or incomplete ones; part of the process then involves uncovering the precise intent of the components and writing the contracts.

This approach seems to yield the necessary flexibility while accommodating the need for rigor and proofs. It yields a useful view of programs.

Slogan: program

Program = Contract + Implementation + Proof obligation

6 Discussion

This article applies to programming the standard method on which science and engineering rely to solve practical problems in any application domain:

- Develop a mathematical model resulting in equations (in the present case, the feasibility equation $Pre \subseteq post$ and the program equation **require** *Pre* **do** *b* **ensure** *post* **end**, where *b* is the unknown).
- Solve the equation.
- Build the solution in the application domain.

The main argument for the model developed in the preceding sections is the simplicity of its premises: the mathematical baggage is elementary set theory, learned in high school around the age of 15; the construction relies on just three mechanisms from that theory: union, composition and restriction. The approach seems to have the potential to cover all the relevant concepts of programming, although the present article takes only a first dig.

6.1 Axioms or theorems?

In theoretical informatics the habit has often been different: devising axiomatic theories. The most developed example is the admirable work of Hoare and colleagues [7][8]. A notable property of these efforts is that they postulate their laws; then "*of course, the mathematician should also design a model of the language, to check completeness and consistency of the laws, to pro-*

vide a framework for the specifications of programs, and for proofs of correctness" [7]. The justification for this method — postulate your ideal laws, the model will follow — is that it has, in Russell's words cited in [9], "*the advantages of theft over honest toil*".

However good the wisecrack, this is not how normal mathematics works. Unless your last name is Euclid or Peano, or your first name Alfred or Bertrand (and even in this last case, only if you have a hereditary peerage), few people will pay attention to axioms you assert on them as if walking down from Mount Sinai. Imagine a world where every mathematical concept were defined axiomatically; in trigonometry, sine and cosine would be postulated as functions satisfying certain properties — the sum of their squares is 1, the derivative of the former is the latter, and so on; and similarly for every important notion. People would quickly tire of having to make incessant leaps of faith.

We expect instead, when presented with new results, to see them *derived*, in the form of definitions and theorems, from what we already know. True, it is often a mark of elegance, for the presenter of a theory and of the laws that it satisfies, to prove that it is the simplest possible construction satisfying these laws; but it is a mark of politeness to perform this feat only as a bonus step, coming after an explanation relying only on material already familiar to the reader.

Stretching Russell's aphorism, we may note that even if Balzac's observation ("*The secret of great fortunes without apparent cause is a forgotten crime*") may explain the *origin* of some hereditary peerages, just as axioms explain the foundations of mathematics, in practice most hereditary peers find it less bothersome to obtain the objects of their daily desires through "honest toil", or at least honest means, than by stealing.

These observations do not rule out occasional reliance on the axiomatic method in the introduction of theories. Aphorisms aside, however, it is hard to justify asserting properties as postulates when they can be proved as theorems. When a manageable mathematical derivation from known concepts exists, it should be the first choice.

As the presentation of the theory of programs has attempted to show, such exactly is the situation with programming. Programs are just relations over sets. An informal and non-exhaustive review of the axioms of classic articles such as [7] and its extension to concurrency [8] (not considering properties specific to individual calculi), as well as [6] and [10], suggests that most of the properties they introduce can be derived, often straightforwardly, from the framework of this article; many indeed appear above as theorems.

Many authors seem to have a suspicion, conscious or not, of the set-theoretical basis of programming; but most — an important exception is Hehner with his "predicative programming" [6] — resist the obvious solution of explicitly building the theory on that basis. They prefer to throw in axioms, even if these axioms mimic the elementary properties of set operators. A dizzying example is the seminal "Laws of Programming" article [7] (together with the more recent [8]), whose authors axiomatically introduce operators with names such as "$\cup$" for non-deterministic choice and "$\subseteq$" for refinement. They never suggest that these could actually *be* the standard mathematical operators bearing the same names; but they cover several pages of *Communica-*

tions of the ACM with such fascinating "axioms" as $P \cup (Q \cup R) = (P \cup Q) \cup R$. One wonders whether the thought ever arose that if it associates like union, commutes like union, distributes like union, and typographically uses the exact symbol of union, perhaps it is union.

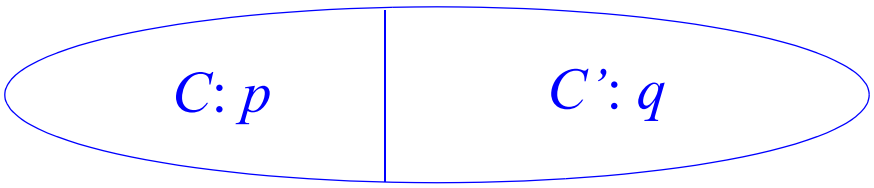
6.2 Keeping simple things simple

Because informatics already struggles to describe inherently complex phenomena, we should not introduce complexity of our own making. Programming theory does not always keep the complexity of the descriptions commensurate with the complexity of the described. Another seminal paper of great elegance [10] introduces the "natural semantics" of the if-then-else conditional thus:

$$\frac{\rho \mid\!- (E_2 \Rightarrow \alpha)}{\rho \mid\!- (\textbf{if } \mathit{True} \textbf{ then } E_2 \textbf{ else } E_3 \textbf{ end}) \Rightarrow \alpha}$$

with a similar rule for the *False* case. In words: if in the environment ρ the expression E_2 evaluates to α, then in ρ the expression **if** *True* **then** E_2 **else** E_3 **end** also does. The companion rule tells us that if E_3 evaluates to β the expression with *False* instead of *True* evaluates to β.

In reality, if-then-else is a very simple concept. It expresses that one may solve a problem by partitioning the domain into two parts and using a different solution in each. Euler would undoubtedly have explained it to his 15-year-old princess pupil [4] by a little illustration:



and she would have understood on the spot. (A pedagogical presentation of the theory of programs' concepts should indeed use Euler-Venn diagrams throughout, although this article has shunned them under the presumption that its putative audience does not need pictures.)

Instead, the above "natural" semantics refers to advanced concepts of mathematical logic and notions such as the "environment" (ρ), which are a distraction from the idea of a conditional instruction. These observations do not put into question the value of [10] and other classic semantic articles, which were conceived as research advances, not tutorials. But they highlight the benefit, as a domain gets understood better, of seeking simplicity and trimming down the set of prerequisite concepts to the indispensable.

6.3 De-emphasizing the program text

One source of complication in theories of programming is reverence for the program text.

Almost every discussion of programming — where "almost" is just to be on the safe side — starts by defining a programming language. (Denotational or operational semantics often starts with *two* languages, one to express programs and the other to express their meanings.)

This attitude seems to be a leftover from the early days when parsing was the difficult problem. Programmers and theorists were awe-struck when Backus, Bauer, Hopper and others showed that instead of coding with zeros and ones it was possible to use a human-readable notation and have it translated automatically. The program text became the alpha and omega of programming. But it is only an artifact. A computer is a mathematical machine for computing pairs in relations. All the rest is decoration.

Programming is no more about programs than electricity is about plugs.

Parsing is the original computer science problem and even though it has long lost its theoretical difficulty it remains our unconscious template for all others. Semantic specification, for example, often looks like a smarter kind of parsing, also starting from program texts and deriving its properties — just more interesting properties. Denotational semantics, in particular, defines "meaning functions" operating on program texts. Electrical engineers, if they worked that way, would start from plugs, dutifully noting how different Swiss, French and Italian plugs are from each other. In reality, of course, what counts is the electrical current — the same in all three countries, with their interconnected networks — and specifically the relevant equations.

In programming too a more productive approach — the application to semantics of the idea of *unparsing*, the reverse of parsing — is to start from an analysis of what we need mathematically: what kinds of postconditions and preconditions give rise to useful specifications and realistic implementations. From this analysis we construct programming notations, not the other way around. For example we do not start from if-then-else as a given construct of interest, but identify the union of two relations as a relevant concept. We consequently derive suitable notations to express it, each adapted to different mathematical situations: if the relations' domains are provably disjoint, **if** *C* **then** *p* **else** *q* **end**; otherwise, the guarded conditional **if** *C*: *p* [] *D*: *q* **end**.

Far from lessening the value of the traditional objects of interest in informatics, such as programs and programming languages, this reversal of perspective makes them even more interesting, turning them from arbitrary products of taste and circumstance into rationally justified modes of expression for useful mathematical concepts.

7 Perspectives

The thesis of this article is that it is possible to found all of programming on a small set of concepts from elementary set theory. The discussion has shown the basic applications, but is only a start. (Also note that the theorems have not been mechanically checked.) Future tasks include:

- Reconstructing entire programming languages on that basis.
- Using the theory to build a "Formal Language Innovation Platform" (FLIP) for experimenting with programming language mechanisms.
- Developing it towards specific approaches to programming, particularly object-oriented.
- Assessing whether the approach can produce effective program verification tools.
- Assessing whether it can help teach programming, including at the elementary level.

8 Acknowledgments

The authors invoked explicitly or not in section 6 (Hoare and coauthors, Kahn, Dijkstra, Scott/Strachey/Plotkin and other pioneers of denotational semantics), complemented by Abrial for his work on Z and B and by Mills and Gries, deserve deep acknowledgments for pioneering the formal approach to programs and programming. Back's and Morgan's seminal work on refinement (following Wirth's) is another fundamental inspiration. Hehner's Predicative Programming is a comprehensive theory of programming based on binary relations, corresponding to the postconditions of the present work. Also influential have been informal comments by David Parnas on the merits of different assertion styles. A note by Shaoying Liu [16], criticizing a purported deficiency in classical refinement approaches (the risk of refining into an unfeasible program), suggested the need for a proper notion of feasibility.

9 References

[1] Back, refinement papers.

[2] Michael Butler, personal communication.

[3] Dijkstra, *A Discipline of Programming*.

[4] Euler, *Lettres à une Princesse d'Allemagne sur divers Sujets de Physique et de Philosophie*, 1760-1762.

[5] Furia, Meyer, Velder, *Computing Surveys* invariant article.

[6] Hehner, *Predicative Programming*

[7] Hoare, original paper on Laws of Programming.

[8] Hoare and van Staden, newer article.

[9] Hoare and van Staden, slides accompanying [8].

[10] Kahn, *Natural Semantics*.

[11] Meyer, IFIP 1980 paper.

[12] Meyer, ETL.

[13] Meyer, OOSC.

[14] Meyer, *Multirequirements*.

[15] Morgan, *Programming from Specifications*.

[16] Shaoying Liu, paper and slides from the 2014 Futatsugi Festschrift.

[17] Wirth, stepwise refinement.