Meanings as programs: Programming Really Is Simple Mathematics

Bertrand Meyer, Reto Weber

Constructor Institute of Technology

Dedicated to Yuri Gurevich on the occasion of his 85th birthday, in homage to his seminal contributions to the theory and practice of software.

Abstract

A re-construction of the fundamentals of programming as a small mathematical theory (“PRISM”) based on elementary set theory. Highlights:

- Zero axioms. No properties are assumed, all are proved (from standard set theory).
- A single concept covers specifications and programs.
- Its definition only involves one relation and one set.
- Everything proceeds from three operations: choice, composition and restriction.
- These techniques suffice to derive the axioms of classic papers on the “laws of programming” as consequences and prove them mechanically.
- The ordinary subset operator suffices to define both the notion of program correctness and the concepts of specialization and refinement.
- From this basis, the theory deduces dozens of theorems characterizing important properties of programs and programming.
- All these theorems have been mechanically verified (using Isabelle/HOL); the proofs are available in a public repository.

This paper is a considerable extension and rewrite of an earlier contribution [24].

1 Overview

The present article is a development of a long-term project to establish computer programming as a mathematical theory. The core ideas were outlined in reference [24], of which it is a revision and extension, with all properties now formally verified.

Caveat: The construction of the theory, starting with section 3, assumes no prior knowledge of programming. While readers may have such knowledge (having possibly, for example, encountered the term “bug” before discovering its definition in section 3.4), the best way to approach this exposition is most of the time to pretend to be reading about a new subject — occasionally breaking this rule by relating the formal view (with the help of “Explanation” and “Justification” paragraphs) to one’s existing understanding of programming concepts.

1.1 Programming and mathematics: two worlds, or one?

Anyone who has both written programs and studied elementary mathematics will have noticed how close the business of writing programs is to the business of proving theorems.

Instead of taking advantage of this closeness, however, most of the work relating programs to mathematics proceeds as if the two worlds were separate. It typically considers a program as if it were some pre-existing object of study, like a bird for an ornithologist or a wave for a physicist, and sets out to explain it in terms of mathematical concepts. Such an approach follows the time-honored process of using mathematics in the

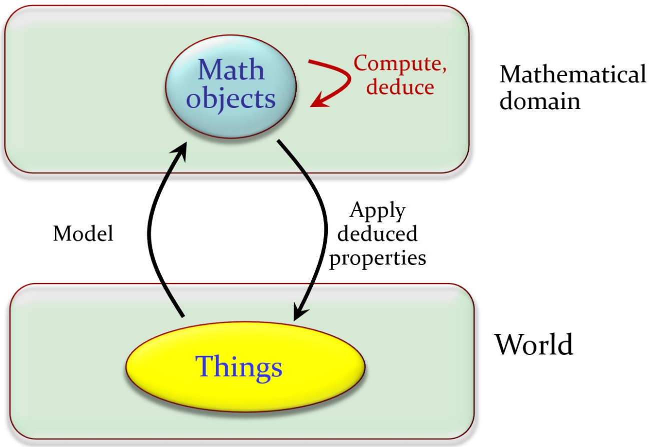

Figure 1: **Modeling through mathematics**

natural sciences: devise a mathematical model of some natural objects; work on the model to deduce interesting properties; then transpose these properties back to the original objects of interest (Fig. 1). Floyd's foundational semantics paper [10] was accordingly titled "*Assigning Meanings to Programs*".

The B book [3] provocatively and famously reversed the pairing, promising to "*Assign Programs to Meanings*": write very abstract programs with clear mathematical properties, then refine them progressively into more concrete programs until they are deemed practical enough for direct use.

We go one step further towards simplification by *removing any distinction* between program and meaning. The corresponding slogan could be "Meanings As Programs". A program is a mathematical object, from a (simple) theory, which happens to be suitable both for execution by a computer and for perusal, under a suitably designed notation ("syntax"), by human readers.

1.2 Staying in the world of mathematics

The key word in the last sentence of the previous paragraph is "is": the mathematical object does not provide a model of some thing endowed with its own existence, it *is* that thing. Unlike the two-world model of Fig. 1, applicable to the natural sciences, the process resembles the simpler one-world model of pure mathematics, which only includes (Fig. 2) the top part of the earlier schema.

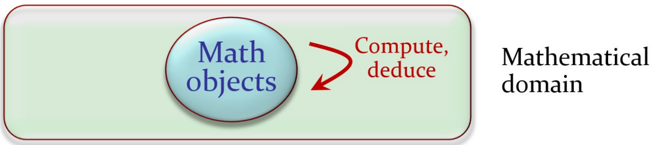

Figure 2: **Reasoning within mathematics**

In other words, we start in the world of mathematics, not programming, and do not ever leave it, even when we go down to concepts that typically appeal to programmers rather than pure mathematicians.

As one does in mathematics, we construct a theory in the usual way, by defining some mathematical objects and proving that they possess certain properties. That theory happens to cover the concepts of programming and its objects are programs, data structures, instructions and other artifacts of programming. As they come from a mathematical theory, they satisfy well-defined properties, proved as theorems.

One difference with the usual (non-software-related) practice of mathematical work has to do with notation. While mathematicians pay considerable attention to notations, they do not generally have the programmer's needs for very large formal texts. In a mathematical article or book, the formal parts each spread over a few lines or, rarely, a few pages, much less than the thousands to millions of lines taken up by a program text, from which concerns arise that are unknown to mathematicians: we need modular constructs (routines, classes, packages...) to organize programs into manageable pieces; to resolve possible ambiguity, we cannot rely on the common sense of a human reader (who readily sees, for example, that one formula extends over two lines but the third line starts a new formula), and must rely instead on rigorously defined syntax, with specific properties (context-free, LL(1), LALR...) enabling compilers to process the texts. To make programs at least moderately readable, debuggable and maintainable, the programming language — even if it is of the esoteric, geeky kind, for example C or Python — must retain some of the verbosity of natural language (usually English) and even some of its words for use as keywords.

As the present work will show, however, programming languages are in the end just notations ("syntactic sugar" is the accepted phrase) for mathematical objects. For example, a standard mathematical concept is the *restriction* of a relation: C: r, for a set C and a relation r, is the relation identical to r except that its domain has been restricted to C. For a programming audience, we allow writing it as **if** C **then** r **end** (see section 7). That form is only an alternative notation for the classical mathematical concept of restriction.

Using this general idea that programs and their components are objects of a mathematical theory couched in a programmer-ready notation, we can pursue the goal of constructing a fully developed programming language, or *re*-constructing an existing language (at least one whose semantics is sufficiently simple and regular), in the form of programming-oriented representations of objects of the mathematical theory.

The present article falls short of attaining this goal, but moves in its direction, up to the fundamental control structures of modern programming languages (sequence, various forms of conditional instruction, various forms of repetition and loops), basic Design by Contract concepts (preconditions, postconditions), and concurrency. The exposition proceeds (starting with section 3) according to the principles defined above: build a theory, starting from extremely simple notions (a relation and a function), derive and prove (with mechanical verification through Isabelle/HOL) theorems capturing important properties of its mathematical objects, and provide programming-like syntax — specifically, Eiffel-like syntax, since at the end of the journey we might reach the entire language — to express them.

1.3 How not to use mathematics for programming (1)

The present work is in part a reaction against some traditional approaches which go in the programs-to-math direction rather than math-to-programs as developed here. Two well-known contributions by prestigious authors typify what we are up against.

Consider first Gilles Kahn's classic "Natural Semantics" paper [19]. It specifies a simple programming language and has a list of definitions for programming constructs, assuming a considerable mathematical baggage. The semantics of the conditional expression ("if ... then ... else ..."), for example, is specified through two inference rules:

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{True} \qquad \rho \vdash E_2 \Rightarrow \alpha}{\rho \vdash \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \Rightarrow \alpha}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{False} \qquad \rho \vdash E_3 \Rightarrow \alpha}{\rho \vdash \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \Rightarrow \alpha}$$

Conditional expression: "natural" (?) semantics, from [19]

(Explanation of this rule: its purpose is to specify how to evaluate, during program execution, the conditional expression **if** *E1* **then** *E2* **else** *E3*. We characterize the state of the program, at the time of evaluation, by a "variable binding" ρ, which maps names of variables to their values at that time; for example, after an assignment of the value True to the variable x, the binding ρ must be such that, under ρ, x is bound to True. Such a property is written $\rho \vdash x \Rightarrow \mathit{True}$, read aloud as "*under* binding ρ, the variable x *evaluates to True*". The rule of Fig. 1.3 has two parts, each of which is an "inference rule" telling us that if the properties above the horizontal line (the hypotheses) hold we may deduce the property below the line (the conclusion). This particular inference rule states that if, under the binding ρ, *E1* evaluates to True, and under that same binding *E2* evaluates to some value α, then, again in that binding ρ, the whole expression **if** *E1* **then** *E2* **else** *E3* will also evaluate to α. And the second rule gives us the corresponding property for the False case. Wow!)

One can only shake one's head: the notion of conditional expression can be explained in full to a 6-year-old: wherever *E1* holds use *E2*, otherwise use *E3*. You may summon Fig. 3 for help. Or realize how easy it is to interpret an "if... then... else" road sign (Fig. 4) even though most drivers would be surprised to hear that if they want to avoid ending up in Bern when they would actually like to get to Basel they first have to study Boolean algebra, variable bindings and inference rules.

If modeling such a basic construct — in a toy functional programming language, which in the cited paper does not even have assignments, routines or loops — requires such a heavy conglomerate of concepts and notations, what would we need for the truly hard mechanisms of programming? That cannot be right. The definition of conditional obtained below will conform to what the last figure suggests: take the *union* of *E2* and *E3*, each *restricted* respectively to *E1* and its complement.

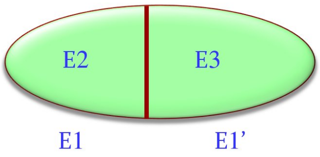

Figure 3: **Visualizing the evaluation of a conditional expression or instruction**

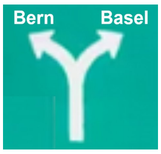

Figure 4: **A "conditional" road sign**

1.4 How not to use mathematics for programming (2)

The second example of how not to approach the relationship between programming and mathematics is another milestone paper, this one by Tony Hoare and eight co-authors [15], extended and refined over the next 25 years later with other coauthors [18]. It introduces "Laws of Programming" intended to serve as a basis for the entire discipline.

There is much to admire in this effort, but one can only balk at the general methodology: *postulating* desired laws as axioms. Some of these laws, together with their comments, appear in Fig. 1.4; they are the ones dealing with the cited authors' "choice" operator, for which they use the union symbol ∪ (or possibly some sign looking very much like it).

(1) Clearly, it does not make any difference in what order a choice is offered. "Tea or coffe?" is the same as " coffee or tea?"

$$P \cup Q = Q \cup P \qquad \text{(symmetry)}$$

(2) A choice between three alternatives (tea, coffee, or cocoa) can be offered as first a choice between on alternative and the other two, followed (if necessary) by a choice between the other two, and it does not matter in which way the choices are grouped.

$$P \cup (Q \cup R) = (P \cup Q) \cup R \qquad \text{(associativity)}$$

(3) A choice between one thing and itself offers no choice at all (Hobson's choice).

$$P \cup P = P \qquad \text{(idempotence)}$$

(4) The ABORT command already allows completely arbitrary behavior, so an offer of further choice makes no difference to it.

$$\perp \cup P = \perp \qquad \text{(zero } \perp\text{)}$$

"Laws of choice", from [15]

If we cease for an instant to believe the authors, who tell us that ∪ is some newly invented operator on programs, called "choice", but chose instead to believe our own lying eyes, which tell us that ∪ is ∪ , meaning union, then the "laws" enunciated here are trivial; they are just the properties of union, which we learned in kindergarten. (We have to assume that "⊥" in this context is but a fancy name for the universal set.)

That is not what the article says. It is taking an entirely axiomatic approach, stating at the outset: "*we suggest that a comprehensive set of algebraic laws serves as a useful formal definition (axiomatization) of a set of related mathematical notations, and specifically of a programming language*"[15]. In other words, define a new mathematical domain through a set of laws, unrelated to anything that already exists. It is inevitable, however, to ask whether an operation that is commutative like the union of subsets, associative like union, idempotent like union, has a unit like union (as well as a zero like union), and for which the authors have chosen what seems to be the same symbol, might not possibly *be* union.

This line of articles includes an amazing number of such proudly shoehorned "laws" that turn out, if interpreted in the framework of elementary set theory, to be obvious properties. As another example, [17] has a "law of consequence" stating (parentheses added here for readability):

$$(p \sqsubseteq p') \quad \wedge \quad ((p' \,;\, q) \sqsubseteq r) \quad \rightarrow \quad (p \,;\, q) \sqsubseteq r$$

where "$\sqsubseteq$" is refinement and ";" is sequential composition. In mathematics, however, "$\subseteq$" is subset and "$r \,;\, s$" is a notation (an alternative to $s \circ r$) for the composition of relations r and s in this order. In such a framework, this "law of consequence (pre)" is a trivial theorem. (See a precise form, */Compose_specialsafety/*, in section 6.5.)

1.5 Other law-guided approaches

Denotational semantics [25] is the quintessential approach considering programs and programming languages as natural objects deserving analysis: for every programming construct c it defines a "meaning" $M\ [\![c]\!]$ defined recursively, starting with special "semantic domains".

Seminal books by Hehner [12], Morgan[26, 27] and Back [2] have devised theories of programming and elucidated many issues. They follow the style also found in the Hoare work cited above: define programming constructs through their properties, in particular refinement. To discuss the conditional instruction, for example, chapter 4 of [27] presents a syntax for the construct, provides an informal operational explanation of its semantics, then throws in a "refinement" law, with no argument that it is sound with respect to some model or consistent with other laws.

1.6 A public incitation to crime?

The authors of the Hoare-style, axiom-based line of work (pursued in successive articles including [15], [13], [15], [18], [14], [17], [16]) proudly advertise its benefits, with [14] invoking Bertrand Russell:

> *The method of postulation has many advantages. They are the same as the advantages of theft over honest toil.*

After reading this citation twice and checking that one is awake, the only reaction — tempered only by the out-and-out impenetrability of British humor to foreigners — is to wonder what kind of argument it offers. If such theft is to be commended, mathematics becomes easy: I can postulate that P is equal to NP (or just as well, if my taste goes the other way, that they are not equal) and win the $1-million Clay prize. Mathematics is not about postulating all you need but (as Russell brilliantly demonstrated when he was doing actual work rather than indulging in dubious professor jokes) to *minimize* the number of axioms and, from them, *prove interesting theorems*, maximizing their number.

A fundamental precept of all sciences, not just mathematics, encourages us to strive for the smallest possible set of hypotheses and to reject unnecessary assumptions (such as specially introduced constants in physics). In the history of science it has been expressed as "Occam's Razor", Newton's and Laplace's "*we are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances*", Einstein's well-known maxim about simplicity etc.

The number of postulated axioms in the PRISM theory presented below is, exactly, zero. We are happy to start from the established properties of set theories, whose axioms we take from credible sources (found in well-known textbooks such as [21]), and set out to prove the properties of programs.

Rejecting axioms, or minimizing their number, is not just a methodological guideline justified by striving for elegance. There is a downside to "postulating" what you want: how do you know the axioms thrown into the kitchen sink are all *consistent* with each other? Consistency proofs are difficult, and when we are dealing with sophisticated constructions such as occur in computer science often teeter on the brink of undecidability. Dijkstra, who cites Hoare's citation of Russell's provocation [8], raises this question even though he is always deferential to Hoare:

> *A possibly more valid doubt may be raised by the question "How do you know that your axioms are consistent?".*
>
> *My inclination — but am I perhaps awfully naive? — is to say "Well, I am pretty sure that they are consistent. And if they are not, well that is my risk! The worst that I could have done is talking about an empty universe, and, although fruitless, that cannot have done much harm either."*

The reader will appreciate the sudden "not much harm" comment from the usually stern disciplinarian of programming methodology, not known for being indulgent towards imprecision. We should remember that a primary goal of applying formal methods in software engineering is to help ensure that future software does not reproduce the behavior of Therac-25 or Ariane 5.

The dangers of an axiomatic approach, to which Dijkstra timidly alluded, are not just theoretical. Shaoying Liu, studying Morgan's laws of refinement ([27]), pointed out [20] that they make it possible to refine a feasible operation into an infeasible one[1]. The lack of a precise mathematical specification and proofs opens the way to such consequences, and does not allow determining whether they are intentional — and then how they one should handle them — or a mistake in the theory. ("Feature or bug?")

The only way to avoid such predicaments is to move away from axiom inebriation and, having woken up sober, to shift the focus to theorems and their mechanically-checked proofs.

1.7 A sober approach

Beyond the natural glee that comes from witnessing great men getting entangled in questionable pronouncements, the lesson of the preceding analysis is twofold:

- In analyzing software we should not pretend programs pre-exist their mathematical models but consider that they *are* mathematical objects.
- A suitable theory should be "as simple as possible but not simpler" in the following sense: *simple programming concepts should have simple mathematical descriptions.* There are complex concepts in software, and we may expect their descriptions to be non-trivial too. But a conditional expression — yes, it is a straightforward concept and no, it cannot possibly require heaps of advanced mathematical concepts (inference rules, bindings, deductions...) to be modeled mathematically. To paraphrase Alan Kay[2], simple things should have simple specifications, and complex things should have specifications.

The rest of this article presents the starting elements of a theory of software based on elementary (high-school-level) set-theoretical concepts — intersection, union, subset, relation, function, composition... — and no new axiom whatsoever. It defines programming constructs as mathematical objects utilizing these concepts, and *proves* all the often remarkable properties that they possess. All the properties and their proofs have been mechanically checked; they are recorded in a repository, which anyone can use to reproduce the proofs. As far as we have been able to determine, all the "laws" and "axioms" of the cited articles by Hoare and colleagues are theorems in the present framework, including laws of concurrency such as the "exchange axiom" and associated exchange laws (see section 10.5).

The PRISM approach of this article leaves the matter of consistency to mathematicians (from Zermelo and Frankel to Gödel and Cohen) and starts from their axioms, proving all new properties along the way.

Much of the formal work in software is axiom-rich and theorem-poor. One builds a complicated mathematical basis for a fairly small Return On Investment. Since that basis cannot be checked for consistency, it is fraught with uncertainty (the reverse of what we normally expect from mathematics!): over the entire edifice hangs the specter of contradiction. Instead, the present approach is not only axiom-poor but **axiom-free**, and **theorem-rich**, hoping to deliver a good ROI to any practicing programmer willing to learn a bit of simple theory.

2 Notations

The discussion relies on standard concepts of elementary set theory and associated notations, sometimes adapted.

> (**Short version of section 2**. A reader familiar with elementary set theory and eager to move to programming concepts may simply skip to section 3 after noting these specifics: $\{x\colon A \mid p\ (x)\}$ to define a set by comprehension, with a colon and a vertical bar, also for $\forall$ and $\exists$ predicates; $\underline{r}$ and $\overline{r}$ for the domain and range of a relation r; and $r\ ;\ s$ for the composition of two relations in the order of their application.)

2.1 Logic and sets

The boolean operators are $\wedge$ (and), $\vee$ (non-exclusive or), $\neg$ (not) and $\rightarrow$ (implies, also written $\Rightarrow$). The usual operations are available on sets: $\in$ (member), $\cap$ (intersection), $\cap$ (union), $\subseteq$ (subset or equal), $\times$ (Cartesian

[1] In the PRISM framework developed in this article, the infeasible version is, per the formal definitions of section 6.2, a refinement but not an implementation of the original version.

[2] Kay's oft-quoted aphorism is: "*Simple things should be simple, complex things should be possible*".

product). Set difference uses a minus sign, “$-$”[3]. Basic sets include $\mathbb{N}$ (natural integers), $\mathbb{Z}$ (all integers) and $\mathbb{B}$ (booleans). The cardinal (number of elements) of a finite set A is written $|A|$. $\mathbb{P}\ (A)$, the powerset of a set A, is the set of all its subsets. Similar conventions apply to universally and existentially quantified predicates: $\forall\ x\colon A \mid p\ (x)$ (all members of A satisfy p), $\exists\ x\colon A \mid p\ (x)$ (some member of A satisfies p)[4].

2.2 Relations and functions

A relation over sets A and B is a subset of $A \times B$: a set of pairs $\{[x_1,\ y_1],\ [x_2,\ y_2], \ldots\}$ where $x_i \in A$ and $y_i \in B$. An alternative notation for the set $\mathbb{P}\ (A \times B)$ of relations between A and B is $A \leftrightarrow B$. The source set of a relation in $A\ \leftrightarrow B$ is A and its target set is B. The inverse of a relation r is written r^{-1}; if r is in $A \leftrightarrow B$, then r^{-1} is in $B \leftrightarrow A$ and its definition is that $[x,\ y] \in r^{-1}$ if and only if $[y,\ x] \in r$. The domain of r, written $\underline{r}$, is the set of elements x of A such that r contains a pair $[x,\ y]$ for some y in B; its range, written $\overline{r}$, is the set of y of B such that r contains a pair $[x,\ y]$ for some x in A[5]. Clearly, $\underline{r} \subseteq A$, $\overline{r} \subseteq B$, $\underline{r} = \overline{r^{-1}}$ and $\overline{r} = \underline{r^{-1}}$.

For a relation r in $A\ \leftrightarrow B$ and a subset X of A, the image of X by r is written simply $r\ (X)$; it is the subset of the target set B made of all the y such that r contains a pair $[x,\ y]$ for some x in A. Properties include $r\ (\varnothing) = \varnothing$, $r\ (X') \subseteq r\ (X)$ whenever $X' \subseteq X$ (referred to in later sections as */Image_subset/*), $r\ (X) \subseteq \overline{r}$ for any X (*/Subset_image/*), $r\ (X \cup\ Y) = r\ (X) \cup r\ (Y)$ (*/Image_union/*), $r\ (X \cap\ Y) \subseteq r\ (X) \cap r\ (Y)$ (*/Image_inter/*), and $r\ (A) = r\ (\underline{r}) = \overline{r}$. A relation r is total if $r\ (X)$ is non-empty for non-empty X. (Two equivalent definitions are: (1) $\underline{r} = S$; (2) for any x in A, r contains at least one pair of the form $[x,\ y]$ for some y.)

The restriction $r\ /\ X$ and corestriction $r \setminus\ Y$ of a relation r in $A\ \leftrightarrow B$, for subsets X of A and Y of B, are the subsets of r containing only the pairs $[x,\ y]$ for which, respectively, $x \in$ X and $y \in$ Y. Clearly, $\underline{r\ /\ X} \subseteq X$ and $\overline{r \setminus\ Y} \subseteq Y$. More precisely, $\underline{r\ /\ X} = \underline{r} \cap X$ and $\overline{r \setminus\ Y} = \overline{r} \cap Y$ (*/Domain_restrict/*). Also, $\overline{r\ (X)} = \overline{r\ /\ X} \subseteq \overline{r\ (A)}$ (*/Image_restrict/*).

Composition of relations uses the operator “;” which lists its operands in the order of application: $r\ ;\ s$ is the set of pairs $[x,\ z]$ such that for some y there is a pair $[x,\ y]$ in r and a pair $[y,\ z]$ in s[6]. (If $r \in A \leftrightarrow B$ and $s \in B \leftrightarrow C$, then $r\ ;\ s \in A \leftrightarrow C$.)

A relation f is a function if it includes, for any given x, at most one pair $[x,\ y]$. The set of functions from A to B, a subset of $A\ \leftrightarrow B$, is written $A \nrightarrow B$. If f is total (as a relation, according to the definition above) then it does have one such pair for every x in A; we then say it is a total function. The set of total functions from A to B, a subset of $A \nrightarrow B$, is written $A \rightarrow B$. A function is said to be partial if it is not total[7]. For a function f and an element x of its domain, we may use the function application notation $f\ (x)$ to denote the single element y such that $[x,\ y] \in f$[8]. If f is total, this notation is always meaningful; otherwise, using it is subject to a **proof obligation** that $x \in \underline{r}$.

2.3 Lambda notation

Similar to the definition of sets by comprehension as seen above, “lambda notation” is available to define specific relations and functions by their properties. For a boolean-valued property $p\ (x,\ y)$ applicable to x and y respectively in A and B, and an expression $e\ (x)$ denoting an element of B for any x in A, we may use:

- $\lambda\ x\colon A \mid e\ (x)$: a (total) function in $A \rightarrow B$ which for every x in A yields $e\ (x)$.

[3] The backslash symbol “\” has another meaning, corestriction of relations, as seen a few paragraphs down.

[4] A dot it often used instead of the vertical bar, but the latter is more visible and avoids confusion for when the theory reaches object-oriented programming.

[5] To avoid any confusion between source set and domain, and between target set and range: stating that a relation r is in $A\ \leftrightarrow B$, with domain A and range B, plays the role of a declaration for r, specifying the sets from which first and second elements (respectively) of pairs in the relation *may* take their values. The domain and range specify which elements *do* appear in r’s actual pairs.

[6] A commonly used operator for composition is “$\circ$”, which lists the operands in reverse order. “;” is used among others by the VDM specification language and goes back to Ernst Schröder in the 19th century)

[7] It is common to use “partial function” for what is called just “function” here: a function that is *possibly* partial. The terminology used in this article is more consistent, with “partial” meaning the opposite of “total”.

[8] The image notation from relations remains available, but since the notation is “typed” (it always specifies, for any object in the text, a set of which it is a member) there is no ambiguity: with f in $A \nrightarrow B$ and an element x of A, the function application $f\ (x)$, if defined, denotes an element of B; for a subset X of A, its image $f\ (X)$, always defined, denotes a subset of B.

- $\Lambda\ x\colon A;\ y\colon B \mid p\ (x,\ y)$: a relation in $A \leftrightarrow B$ which contains the pairs $[x,\ y]$ such that $p\ (x,\ y)$ holds. (It can also be written $\{[x,\ y]\colon A \times B \mid p\ (x,\ y)\}$ but the Λ notation emphasizes that we get a relation.)

Functions as defined so far take a single argument. It can be in a cartesian product: if f is in $A \times B \rightarrow C$, we can apply it to a pair, as in $f\ ([x,\ y])$, but we can also consider it as two-argument function and write the application as just $f\ (x,\ y)$; similarly for more arguments. Lambda notation for functions can similarly use more than one variable. The notation also makes it possible to define multi-level functions and relations, which use functions or relations as arguments or results. For example the composition operator ";" can be defined as a function taking two relations and yielding a relation[9]:

$$\lambda\ r\colon A \leftrightarrow B;\ s\colon B \leftrightarrow C \mid \Lambda\ x\colon A;\ z\colon C \mid \exists\ y\colon B \mid [x,\ y] \in r \wedge [y,\ z] \in s$$

Sometimes it is convenient to restrict functions to one argument. One can resort to this convention without any loss of generality by applying a scheme known as *currying*, which from a two-argument function f in $A \times B \rightarrow C$ yields a function f' in $A \rightarrow (B \rightarrow C)$ such that $(f'(x))\ (y) = f\ (x,\ y)$.

2.4 Closures

The next notions apply to a relation r that is in $A \leftrightarrow A$ for some A (same source and target set). We may define its powers: r^1 is r, r^2 is $r\ ;\ r$ (r composed with itself) and so on[10]; r^0 is the identity relation $Id\ [A]$ on A (defined as $\Lambda\ x\colon A \mid [x,\ x]$). Another special relation is $Univ\ [A]$, defined as $A \times A$, the universal relation, which includes every pair of elements of A. The reflexive transitive closure r^* of r is $\cup\ r^i$ for all natural integers i; excluding from this union of the relations the case $i = 0$ yields the (non-reflexive) transitive closure r^+. A relation r is transitive if $r\ ;\ r \subseteq r$[11]. (Equivalently, if $r = r^+$.) It is symmetric if $r = r^{-1}$ and reflexive if $Id\ [A] \subseteq r$. By construction, the transitive closure (reflexive or not) of any relation is associative, and the reflexive closure is reflexive. Related concepts for functions of two arguments (often expressed as operators, such as ";" on relations) are "associative" ($f\ (f\ (x,\ y),\ z) = f\ (x,\ f\ (y,\ z))$) and "commutative" ($f\ (x,\ y) = f\ (y,\ x)$).

3 Basic programming concepts

Notational convention: All formal elements (definitions, theorems...) have a unique identifier such as */Basic_def/* below, with a fixed structure ("slash" characters bracketing a two-part name with intervening underscore). The same names are used in the publicly available Isabelle/HOL repository [30] associated with this article, which includes the mechanically-verified proofs of all the theorems, enabling any reader to peruse and reproduce them. A reference such as */Basic_def/* is (in the electronic version) a hyperlink to the defining occurrence.

3.1 Definitions: program, specification, precondition, postcondition

A **program** (or **specification**), relative to a set S, the **state space**, consists of: */Basic_def/*

- A relation $post$ in $S \leftrightarrow S$, the **postcondition**. */Post_cond/*
- A subset Pre of S, the **precondition**. */Pre_cond/*

It may be written $\langle post,\ Pre \rangle$.

Explanation: 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. 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. (See 3.2 below about non-determinism.)

Justification: The usual view in software engineering treats "program" and "specification" as distinct concepts, but all definitions of the purported difference are vague: for example, that a specification describes the "what" and a program the "how", not an meaningful distinction since these notions are relative. An assignment is implementation to the application programmer and specification to the compiler writer. $\mathbf{out}^2 \cong in$ may look like a specification; but some "programming" languages — and AI tools — accept it, letting the compiler derive

[9] In reading this example, make sure not to confuse the capital "lambda" Λ and the boolean "and" $\wedge$.

[10] Formally, "and so on" means that r^{i+1} is defined as $r^i\ ;\ r$, or equivalently as $r\ ;\ r^i$.

[11] In other words, whenever r contains two pairs $[x,\ y]$ and $[y,\ z]$ it also contains $[x,\ z]$.

a square-root algorithm. Any useful distinction must be relative: a program/specification "*specifies*" another. The introduction of "contracted program" in section 9) will formalize this idea.

Limitation: The definition covers programs/specifications of the input-to-output kind. It can be extended to continuously running programs such as operating systems, and to reactive systems — extensions that are beyond the scope of the present article.

Notation: $post_p$, and Pre_p (the latter also written $\underline{p}$ per 3.3 below) are the postcondition and precondition of a program p. If p_i is an indexed list of programs, we may use $post_i$ and Pre_i as abbreviations for $post_{p_i}$ and Pre_{p_i}.

Convention: For simplicity, the notations for programs used in this article take the state S for granted, assuming that all programs act on the same global S. A finer-grain analysis would make the state explicit in each case, adding the notation S_p, and specify a program with three components as $\langle post,\ Pre,\ S\rangle$ rather than just the first two. The definitions and proofs in the Isabelle/HOL complement to this article [30] do take this approach of making the state explicit. A few more observations on the nature of states appear in section 3.9.

3.2 Definition: deterministic

A program p is:

- **Deterministic** if $post_p$ is a function. */Deterministic_def/*
- **Non-deterministic** otherwise. */NonDeterministic_def/*

Explanation: For a deterministic program, the postcondition is a function, so that the program always delivers at most one result. (For an input in Pre it delivers *exactly* one result.) In simple sequential programming, programs are usually deterministic; in concurrent and distributed programming, many programs are non-deterministic.

3.3 Definition: domain and range of a program

The following notations generalize to a program p the domain $\underline{r}$ and range $\overline{r}$ notations for a relation. The first is just an abbreviation for the precondition of a program. The second one denotes the set of states that the program can reach (if started in its precondition).

Notation	**Definition**	**Name**
$\underline{p}$	Pre_p	Precondition ("domain") of a program */Pre_notation/*
$\overline{p}$	$\overline{post_p\ /\ \underline{p}}$	Usable range of a program */Prog_range/*

From properties of relations (*/Domain_restrict/*) it follows, for any set C of states, that:

$post_p\ (\underline{p}) = \overline{p}$ */Domain_range/*
-- Notation reminder: left side is image of $\underline{p}$ by relation $post_p$.

3.4 Definitions: feasible, rounded, exact

A program p is:

- **Feasible** if $\underline{p} \subseteq \underline{post_p}$. */Feasible_prog/*
- **Rounded** if $\underline{post_p} \subseteq \underline{p}$. */Rounded_prog/*
- **Exact** if both feasible and rounded. */Exact_prog/*

Terminology: The following terms will denote departures from feasibility and roundedness (see Fig. 5):

- If a program is *infeasible*, it cannot handle some legal input states (those in $\underline{p} - \underline{post_p}$). We say that the program is **buggy**; such unhandled cases signal **bugs**.
- If the program is *unrounded*, the relation could handle some input states (those in $\underline{post_p} - \underline{p}$) that are, however, not allowed. We say that the program has **dead code**.

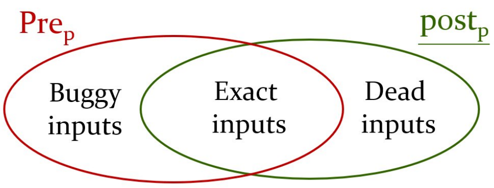

Figure 5: **Categories of input states**

Explanation: $\underline{p}$ 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 $\underline{p}$, at least one output state satisfies $post_p$. Roundness, on the other hand, expresses that the relation is used to its full extent: the program will accept all meaningful input states.

Justification:

> A program/specification describes a transformation between states, through the relation $post$. The definition, however, also includes a precondition Pre. It is legitimate to ask why Pre is necessary: would it not be enough to specify the postcondition? Then the precondition would just be $\underline{post}$, the domain of the postcondition.
>
> This approach would simplify the theory but is unfortunately not adequate in practice since we are often given a previously-known general relation which may not be applicable to all relevant cases; or, conversely, its application is only relevant for some of the possible cases in its domain. As a simple example, a real-square-root program is defined by the relation $\Lambda\ in,\ out\colon \mathbb{R} \mid out^2 \cong in$, a member of $\mathbb{R} \leftrightarrow \mathbb{R}$, which is not applicable to every possible input state, but only for a subset of the source set, the set $\mathbb{R}_{\geq 0}$ of non-negative numbers, which may serve as the precondition.
>
> We could in this example consider that the relation is in fact a member of $\mathbb{R}_{\geq 0} \leftrightarrow \mathbb{R}$, on which it is total (and hence does not require a precondition), but even in such a simple case it is inconvenient to have to define endless special input-restricted variants of general relations. In more advanced examples that approach would be even more impractical.
>
> The square root example illustrates the need for the precondition to guarantee applicability of the relation; formally, the condition is $\underline{p} \subseteq \underline{post_p}$, "feasibility", violated for example if we take Pre to be the whole of $\mathbb{R}$. We may also encounter the converse problem — a precondition that is too small (instead of too large). To avoid this case the program must satisfy the dual condition, roundedness: $\underline{post_p} \subseteq \underline{p}$.

Example: With $\mathbb{R}$ as the set of states S, consider the square root program where $post$ is again $\Lambda\ in,\ out\colon \mathbb{R} \mid out^2 \cong in$, so that $\underline{post}$ is $\mathbb{R}_{\geq 0}$ since all pairs in the relation have non-negative in. Then (using $\mathbb{R}^\bullet$ for the set of non-zero numbers and $\mathbb{R}^+$ for the set of positive numbers):

- $\langle post,\ \mathbb{R}^\bullet\rangle$ is not feasible (it accepts negative numbers but cannot process them) and not rounded (the relation includes the pair $[0,\ 0]$ which is not accepted).
- $\langle post,\ \mathbb{R}^+\rangle$ is feasible but not rounded (because of that same pair) .
- $\langle post\ /\ \mathbb{R}_{\geq 0},\ \mathbb{R}\rangle$ is not feasible but is rounded.
- $\langle post\ /\ \mathbb{R}_{\geq 0},\ \mathbb{R}_{\geq 0}\rangle$ is feasible and rounded.

Comment: Having to prove feasibility or roundness may seem tedious, but the real culprit is the notion of precondition. In light of the preceding definition, we may surmise that the need to include this notion is in line with general results of incompleteness and undecidability in computer science: we cannot limit ourselves to a comfortable world where all postconditions would be total (and all programs would terminate). We must accept that many realistic programs will have a non-trivial precondition and that we cannot escape our obligation to satisfy the precondition when applying the program.

3.5 Definitions: trimming and rounding a program

For a given program p:

- $\lceil p \rceil$, the **feasible version** of p, is $\langle post_p,\ \underline{p} \cap \underline{post_p} \rangle$. *\/Feasible_version/*
- $\lfloor p \rfloor$, the **rounded version** of p, is $\langle post_p\ /\ \underline{p},\ \underline{p} \rangle$. */Rounded_version/*
- $[p]$, the **exact version** of p, is $\langle post_p\ /\ \underline{p},\ \underline{p} \cap \underline{post_p} \rangle$. */Exact_version/*

Terminology: The feasible version is also called the "trimmed" version. "Making a program feasible", or synonymously "trimming" it, means replacing it by its feasible version. "Rounding" it means replacing it by its rounded version.

Explanation: In line with the earlier explanations (section 3.4), we may view $\lceil p \rceil$ as p cleaned of dead code, and $\lfloor p \rfloor$ as p freed from bugs, in other words restricted to input states that it can actually process.

Example: For square root, with $\mathbb{R}$ as the set of states S and $p = \langle post,\ \mathbb{R}^\bullet \rangle$ where $post$ is $\Lambda\ in,\ out\colon \mathbb{R} \mid out^2 \cong in$, so that p, as seen above, is neither feasible nor rounded:

- $\lceil p \rceil$ is $\langle post,\ \mathbb{R}^+ \rangle$ (rejecting "buggy" negative inputs, but retaining "dead code" in the postcondition), feasible and not rounded.
- $\lfloor p \rfloor$ is $\langle post\ /\ \mathbb{R}^\bullet,\ \mathbb{R} \rangle$ (eliminating dead code but retaining buggy input), rounded and not feasible.
- $[p]$ is $\langle post\ /\ \mathbb{R}^\bullet,\ \mathbb{R}^+ \rangle$ (without dead code or bugs), exact. The postcondition could also be $post\ /\ \mathbb{R}^+$.

Feasibility and roundness theorems

For any program p:

$\lceil p \rceil$ is feasible. */Consistent_feasible/*
-- The "feasible version" deserves its name.

$\lfloor p \rfloor$ is rounded. */Consistent_round/*
-- The "rounded version" deserves its name.

$[p]$ is exact. */Consistent_exact/*
-- The "exact version" deserves its name.

If p is feasible, so is $\lfloor p \rfloor$. */Feasible_round/*
-- Rounding preserves feasibility.

If p is rounded, so is $\lceil p \rceil$. */Round_feasible/*
-- Trimming (making feasible) preserves rounding.

Proof idea: */Feasible_round/* follows from a property of restriction: if $A \subseteq \underline{r}$, then $A \subseteq \underline{r\ /\ A}$, applied to the relation $post_p$ and the set $\underline{p}$. */Round_feasible/* follows from the elementary set-theoretical property that if $A \subseteq B$, then $A \subseteq A \cap B$, applied to the sets $\underline{post_p}$ and $\underline{p}$.

Notations:

- $post_p\ /\ \underline{p}$, the postcondition of $\lfloor p \rfloor$, may be written $\lfloor post_p \rfloor$.
- $\underline{p} \cap \underline{post_p}$, the precondition of $\lceil p \rceil$, may be written $\lceil \underline{p} \rceil$.

3.6 Definition: total

A program p is **total** if it is feasible and $\underline{p} = S$. */Total_program/*

Explanation: A total program can process all states. In informal parlance, it "has no precondition". (Formally, it does have a precondition: the entire state set S, also called $True$ per 3.8 below.)

Comment: This definition of "total" for a program is consistent with the definition of "total" for a relation in section 2 in the following sense: p is a total program if and only if it is feasible and $post_p$ is a total relation.

3.7 Definitions: program equality and equivalence

Two programs p and q are:

- **Equal**, written $p = q$, if they have the same postconditions and preconditions (that is, $post_p = post_q$ and $\underline{p} = \underline{q}$). */Equal_program/*
- **Equivalent**, written $p \equiv q$, if their rounded versions are equal (that is, $\lfloor p \rfloor = \lfloor q \rfloor$). */Equiv_program/*

Comments: The first definition is the usual notion of equality (since a program is defined by its postcondition and precondition). As elsewhere in this article, we assume that the state set S is the same for all programs.

Justification: The theorems introduced in the following sections sometimes state that two expressions yield equal programs, but sometimes one or both of the programs have some extra pairs ("dead code") in their $post$. In such cases only the rounded versions are equal and the theorems assert equivalence rather than full equality.

Property: Obviously, equivalence (like any relation of the form "p and q have the same <something>") is an equivalence relation in the usual sense (reflexive, symmetric, transitive).

3.8 Conditions

Terminology: In the rest of this article, the term "condition" will denote a subset (usually called C, D or Pre) of the set of states. S. A precondition is a condition, but a postcondition is not (it is a relation on states). The following notations from logic can be used in lieu of their set-theoretical counterparts.

Notation	**Definition**
$True$	S */True_def/*
$False$	$\varnothing$ */False_def/*
$C \vee D$	$C \cup D$ */Or_def/*
$C \wedge D$	$C \cap D$ */And_def/*
$\neg\ C$	$C = False$ */Not_def/*
$C \Rightarrow D$	$C \subseteq D$ */Implies_arrow/*
C **implies** D	$C \subseteq D$ */Implies_keyword/*

Comment: $C = True$ is equal to C. The rest of this article uses the notations defined here except for "$\vee$" and "$\wedge$" for which the set operators are just as convenient.

Theorems: properties of special conditions

For any program p:

$True$: $p = p$ */Restrict_true/*

$False$: $p \equiv Fail$ */Restrict_false/*

$p \setminus True \equiv p$ for feasible p . */Corestrict_true/*

$p \setminus False = Fail$ */Corestrict_false/*

3.9 The nature of states

Other than in specific illustrative examples, this article ignores what the state set S actually is; it is indeed possible to obtain many properties of programs and programming without considering the structure of S. That is what the following sections do.

In other contexts (and beyond the scope of the present article) it becomes necessary to look into specific kinds of state, which yield specific forms of programming. Here we may simply note that useful forms of states generally include a *binding*, defining the relationship between elements of programs and elements of their executions. A binding b is a function in $Name \nrightarrow Value$ for some sets $Name$ (related to a program's text) and $Value$ (related to the program's executions).Two special kinds of program are particularly interesting:

A program p is:

- **Functional** if $b \subseteq (post\ ;\ b)$. */Functional_def/*
- **Object-oriented** if $Name$ is $1..n$ for some integer n. The elements of $Value$ in this case are called "objects" and the binding is also called a "heap". */OO_def/*

Explanation:

- A functional program does not modify any existing name-value association. In other words, the binding that holds in a state s remains in any of the states in $post\ (s)$: for all s, $b\ (s) \subseteq b\ (post\ (s))$ as expressed by the definition. For the binding and the order relation defined by $\subseteq$, the postcondition is non-decreasing. Execution can enrich the binding with new $[name,\ value]$ pairs but never remove any pair.
- An object-oriented program manipulates a set of values each with a unique "object id".

The two definitions are unrelated, with the consequence that a program can be both functional and object-oriented. The fundamental program constructs and properties reviewed in the rest of this article are entirely independent of the nature of state and the resulting programming style (such as functional or object-oriented).

4 Fundamental program constructs

The following operations on programs are the basic building blocks of programming. They are not typically used directly as programming language constructs (although they could be), but serve as a basis for the actual language constructs introduced from sections 7 on.

The present section also introduces a set of extreme-case programs with simple definitions and distinctive properties, similar to special values (zero, unity) in other mathematical theories.

4.1 Overview of the basic operators

There are three basic constructs. Informally (for programs $p,\ q$ and a set of states C):

- C: p, the **restriction** of p to C executes like p, but is applicable only in C.
- $p \cup q$, the **choice** (or union) between p and q, executes like either p or q whenever one of them is applicable. (If both are, the execution could use either one.)
- $p\ ;\ q$, the **composition** of p and q, executes first like p then like q.

Notation reminder: $r\ /\ C$ and $r \setminus D$ are the restriction and corestriction of a relation r to subsets C of its source set and D of its target set.

4.2 Definitions: restriction, choice and composition

Notation	Name	Postcondition	Precondition
$C\colon\ p$	Restriction	$post\ /\ C$	$Pre \cap C$ */Restrict_def/*
$p \cup q$	Choice (or: union)	$\lfloor post_p \rfloor \cup \lfloor post_q \rfloor$	$\underline{p} \cup \underline{q}$ */Choice_def/*
$p\ ;\ q$	Composition	$post_p\ ;\ \lfloor post_q \rfloor$	$Pre_p \cap \underline{post_p \setminus Pre_q}$ */Compose_def/*

Explanation: See the informal introduction above.

Justification: These definitions embody the core thesis of this work, considering that program operations are essentially applications of classical operations from elementary set theory:

- Restriction on programs is restriction of the postcondition and precondition.
- Union of programs is union of the (rounded) postconditions and precondition.
- Composition on programs is composition of the postconditions, with the necessary narrowing of the precondition and again rounding.

Notation: The operators for programs reuse the symbols of the corresponding set operators. " $\cup$ " and "$;$" in the first column above are union and composition of programs; in the other columns, they are union of sets and composition of relations. Restriction for programs could be written $p \setminus C$ like its relation counterpart; the semicolon notation reflects programmers' practice of writing the condition first, as in "if C holds, execute p". (The actual "conditional instruction", based on restriction and choice, appears below in section 7.)

Comment: The rounding of postconditions is necessary for consistency:

- For choice, using only $post_p \cup post_q$ without rounding may lead (if the original programs are not rounded) to wrongly resuscitating dead code; specifically, applying $post_q$ in a state in $\underline{q} - \underline{p}$ (or conversely). For example, if p is $\langle \{[1, 1], [2, 1]\}, \{1\} \rangle$ and q is $\langle \{[2, 2]\}, \{2\} \rangle$ then the union of the postconditions — without rounding — includes $[2, 1]$, which is applicable under the union of the preconditions $\{1, 2\}$, even though it is applicable under neither of the original programs. The definition of choice removes such anomalies by rounding the two operands.
- Similarly for composition, a dead pair $[b,\ c]$ of $post_q$ (dead because b is not in $\underline{q}$) might find its match in a pair $[a, b]$ of $post_p$, bringing it back to life. For example, if p is $\langle \{[1, 10], [1, 20]\}, \{1\} \rangle$ and q is $\langle \{[10, 10], [20, 1000]\}, \{10\} \rangle$, with the second pair of its postcondition dead, then the composition of the postconditions — without rounding of the second one — will take advantage of that pair to include, along with the expected pair $[1, 10]$, the spurious pair $[1, 1000]$ in the result. The definition of composition removes such anomalies by rounding the second operand.

Notational variants:

- Choice might also be called union and composition sequence, compound or block.
- It would also be possible to use Dijkstra's symbols for "guarded commands" [9]: $p\ [\!]\ q$ for $p \cup q$ and $C \rightarrow p$ for $C\colon\ p$.

Comment: In a version of the theory that makes the state explicit, the respective state sets resulting from the three basic operations are S_p, $S_1 \cup S_2$ and again $S_1 \cup S_2$.

4.3 Basic programs

Notation reminder: $True$ is another notation for S, the set of states, and $False$ for the empty set of states $\varnothing$. $Id\ [X]$ is the identity relation and $Univ\ [X]$in $X \leftrightarrow X$. $\langle post, Pre \rangle$ defines a program by its postcondition and precondition, in that order; $False$ denotes the empty relation in the first position (where it may also be written $\varnothing$), and $\varnothing$ denotes an empty relation in the second position.

Name	Definition
Fail	$\langle \varnothing,\ False \rangle$ */Fail_def/*
Infeasible	$\langle \varnothing,\ True \rangle$ */Infeasible_def/*
Havoc	$\langle Univ\ [S],\ True \rangle$ */Havoc_def/*
Skip	$\langle Id\ [S],\ \ True \rangle$ */Skip_def/*
$Skip_C$ (for $C \subseteq S$)	$\langle Id\ [C],\ \ C\ \rangle$ */Skip_restr/*

Explanation:

- *Fail* is never applicable — and hence never produces any result. This program is sometimes also known as “abort”.
- *Infeasible* is more devious: it accepts any input but never produces any result. Compare with *Fail* which at least tells you right away that something is wrong: its execution can never proceed. *Infeasible* pretends to proceed but after that nothing more can happen.
- *Havoc* accepts any input but does not guarantee anything about the resulting state. Something happens but *anything* can happen.
- *Skip* is always applicable but yields a change identical to what it was. Such a program is also called a “no-op” (short for “no operation”).
- $Skip_C$ is like *Skip* but applicable only on a specific set of states C. Clearly, *Skip* is the same as $Skip_C$.

Fail, *Havoc* and *Skip* are all feasible programs. *Infeasible* is not.

Comment: These definitions, other than for $Skip_C$, assume a single set of states S. A more fine-grain definition is also possible, making the state set explicit and indexing all the basic programs with it, as with $Skip_C$.

Properties: It will follow from the theorems below that the set of programs is a monoid (associative, identity element) for “;” and *Skip*, and a semilattice (monoid, commutative, idempotent) for “ $\cup$ ” and *Fail*.

4.4 Corestriction

Symmetrically with restriction on programs (4.2, derived from restriction on relations 2.2, we may define a corestriction for programs out of corestriction for relations. This operator is less fundamental than the three basic ones studied earlier, but does have its uses.

Notation	Name	Definition
$p \setminus D$	Corestriction	$\langle post_p \setminus D,\ \underline{p} \rangle$ */Corestrict_def/*

Explanation: D is a set of states. $p \setminus D$ the **corestriction** of p to D. executes like p, but only keeps results that belong to D.

Caveat: A feasible program may become infeasible through corestriction, since some initial states satisfying $\underline{p}$ may yield final states not satisfying D. To guarantee feasibility, use $\lceil p \setminus D \rceil$. An alternative definition of corestriction, guaranteeing feasibility of the result, would use (instead of $\underline{p}$) the precondition $\underline{p} \cap \underline{post_p \setminus C}$. See */Compose_prepost/* in 5.3 below.

5 Properties of basic programs and operators

The basic programs and fundamental operators introduced above satisfy important properties, some of which are listed below. As other theorems in this article, these properties have been proved with Isabelle/HOL [28]. To facilitate connecting the proofs and the present article, the names appearing here for every property, such as *Skip_compleft*, also serve as labels of the corresponding proofs in the Isabelle files.

5.1 Properties of basic programs

Theorems: Properties of *Skip*

For a program p and a subset C of S:

$p \; ; \; \mathit{Skip} = p$ /Skip_compleft/
-- *Skip* is a neutral element for composition, left...

$\mathit{Skip} \; ; \; p \equiv p$ /Skip_compright/
-- ... and right.

$\mathit{Skip}_{\mathit{False}} = \mathit{Fail}$ /Skip_empty/
-- Restriction to nothing is failure.

$\mathit{Skip}_C \; ; \; p \equiv C \colon p$ /Skip_comprestrict/
-- Left-composing with restricted *Skip* yields restriction...

$p \; ; \; \mathit{Skip}_C \equiv p \setminus C$ /Skip_composecorestrict/
-- ... and right-composing yields corestriction.

Comment: In contrast with its properties for composition a, *Skip* obeys no general property vis-à-vis choice. $\mathit{Skip} \cup \mathit{Skip} = \mathit{Skip}$ is a consequence of more general properties of choice (see /Choice_idem/ in 5.4).

Theorems: Properties of *Fail* and *Havoc*

For feasible p:

$(\mathit{Fail} \cup p\ ) \equiv (p \cup \mathit{Fail}) \equiv p$ /Fail_choice/
-- *Fail* is a neutral element of choice, left and right.

$(\mathit{Fail} \; ; \; p\ ) = (p \; ; \; \mathit{Fail}) = \mathit{Fail}$ /Fail_comp/
-- *Fail* absorbs composition, left and right.

$(a \cup b) \equiv \mathit{Fail}$ if and only if $a \equiv \mathit{Fail}$ and $b \equiv \mathit{Fail}$ /Fail_choiceonly/
-- Choice can only fail if both operands fail.

$(p \ \cup \ \mathit{Havoc}) = (\mathit{Havoc} \cup p) = \mathit{Havoc}$ /Havoc_choice/
-- *Havoc* absorbs union.

5.2 Properties of restriction

Theorems: Properties of restriction

For programs p, q and conditions C, D:

$(\underline{p}\colon \ p) = p$ /Restrict_own/
-- Restriction to your own precondition is no restriction.

$C \colon (D \colon p) = (C \cap D) \colon p$ /Restrict_inter/
-- Repeated restriction is intersection.

$C \colon (D \colon p) = D \colon (C \colon p)$ /Restrict_commute/
-- As a consequence, restriction is commutative.

$C \colon (C \colon p) = C \colon p)$ /Restrict_idem/
-- Another consequence is that it is idempotent.

$C \colon (p \cup q) \equiv C \colon p \cup C \colon q$ /Restrict_distrib/
-- Restriction distributes over choice. Equivalence only:
-- restriction may introduce dead code that choice removes.
-- Example: $p = q = \langle \{[1, 1]\}, \{1\} \rangle$, $C = \mathit{False}$.

$C\!: (p \; ; \; q) = C\!: p \; ; \; q$ /*Compose_absorb*/

-- Composition absorbs restriction.

5.3 Properties of corestriction

Theorems: Properties of corestriction

For a program p and conditions C and D:

$(p \setminus C) \setminus D = p \setminus (C \cap D)$ /*Corestrict_inter*/

-- Repeated corestriction is intersection.

$(p \setminus C) \setminus D = p \setminus D \setminus (C))$ /*Corestrict_commute*/

-- As a consequence, corestriction is commutative.

$C\!: (C\!: p\ ) = C\!: p$ /*Corestrict_idem*/

-- Another consequence is that it is idempotent.

$(p \cup q) \setminus C = p \setminus C \cup q \setminus C$ /*Corestrict_choice*/

-- Corestriction distributes over choice.

$(p \; ; \; q) \setminus C = p \; ; (q \setminus C)$ /*Corestrict_compose*/

-- Corestriction is only applied to the last element.

If p is feasible then $((\underline{p} \cap \underline{post_p \setminus C})\!: p) \setminus C$ is feasible. /*Compose_prepost*/

5.4 Properties of choice

Theorems: Properties of choice

For programs p, q, v:

$p \cup q = q \cup p$ /*Choice_commute*/

-- Choice is commutative.

$p \cup (q \cup v) = (p \cup q) \cup v$ /*Choice_assoc*/

-- Choice is associative.

$p \cup p \equiv p$ /*Choice_idem*/

-- Choice is idempotent (equivalence only, not equality)
-- Equality does not always hold as choice removes dead code.
-- Example (equivalence, no equality): $p = \langle\{[1,\ 1]\},\ False\rangle$.

$\overline{p \ \cup \ q} \ = \ \overline{p} \ \cup \ \overline{q}$ /*Choice_range*/

-- Choice distributes over the range of postconditions.

Comment: These properties include the "Laws of Choice" of the Hoare theory papers (section 1.4)[12] — but as (mechanically) proved theorems, not axioms.

[12] Except for the *abort*-related property; see 5.7 below.

5.5 Properties of composition

Theorems: Properties of composition

For programs p, q, v and conditions C and D:

$p \; ; \; (q \; ; \; v) \; = (p \; ; \; q) \; ; \; v$
-- Composition is associative.
-- So we may apply ";" to several operands
-- without parentheses, as in $p \; ; \; q \; ; \; v$
-- and to a list of programs with Σ, see below.
/Compose_assoc/

$C\!: (p \; ; \; q) \; = (C\!: p) \; ; \; q$
-- Composition left-absorbs restriction.
/Compose_absorbrest/

$(p \; ; \; q) \setminus D = p \; ; \; (q \setminus D)$
-- Composition right-absorbs corestriction.
/Compose_absorbcorest/

$v \; ; \; (p \cup q) \equiv \; (v \; ; \; p) \cup (v \; ; \; q)$
-- Composition left-distributes over choice.
-- Equivalence only, not equality: choice removes dead code
-- but composition may re-introduce some.
-- Example: $p = q = v = \langle \{[1,\ 1], [2,\ 1]\},\ \{1\} \rangle$
/Compose_choiceleft/

$(p \cup q) \; ; \; v \equiv \; (p \; ; \; v) \cup (q \; ; \; v)$
-- Composition right-distributes right over choice
-- (equivalence only).
/Compose_choiceright/

If q is feasible then $p \; ; \; q$ is feasible.
-- This result is stronger than */Compose_feasible/* (5.6).
/Compose_feasibleleft/

Notation: The composition $p_1 \; ; \; ...; \; p_n$ of a list of programs may be written $\Sigma \; p_i$. Associativity of composition (Compose_assoc) justifies this notation. For an empty list, the sum is *Skip* by convention (see 8.1 for details).

5.6 Properties of roundness and feasibility

Theorems: Fundamental operators preserves roundness

For programs p and q and an arbitrary condition C:

$\lfloor C\!: p \rfloor \; = \lfloor p \rfloor \; / \; C$ */Round_restrict/*
$\lfloor p \cup q \rfloor \; = \lfloor p \rfloor \cup \lfloor q \rfloor$ */Round_choice/*
$\lfloor p \; ; \; q \rfloor \; = \lfloor p \rfloor \; ; \; \lfloor q \rfloor$ */Round_compose/*
$\lfloor p \setminus C \rfloor \; = \lfloor p \rfloor \setminus C$ */Round_corestrict/*

As a consequence, if p and q are rounded:
C: p is rounded. */Restrict_rounded/*
$p \cup q$ is rounded. -- True even if p and q are not rounded. */Choice_rounded/*
$p \; ; \; q$ is rounded. */Compose_rounded/*

Proof idea: For restriction (*/Restrict_rounded/*), the property to prove, assuming $\underline{post_p} \subseteq \underline{p}$, is $\underline{post_p \; / \; C} \subseteq \underline{p} \cap C$. It follows from the property of restriction on relations called */Domain_restrict/* in section 2.2: $\underline{r \; / \; X} = \underline{r} \cap X$. For choice, if $\underline{post_p} \subseteq \underline{p}$ and $\underline{post_q} \subseteq \underline{q}$ then $\underline{post_p} \cup \underline{post_q} \subseteq \underline{p} \cup \underline{q}$; rounding the operands of the union (per the definition of the postcondition of the choice operation) changes nothing since these operands are already rounded. Similarly, for composition, the postcondition is $post_p \; ; \; post_q$ and the theorem in this case follows from the property of composition of relations that $\underline{r \; ; \; s} = \underline{r} \cap r^{-1} \; (\underline{s})$.

Theorems: Fundamental operators preserve feasibility

For feasible p and p_2 and an arbitrary condition C:

C: p is feasible. *⁄Restrict_feasible⁄*

$p \cup q$ is feasible. *⁄Choice_feasible⁄*

p ; q is feasible. *⁄Compose_feasible⁄*

Proof idea: (From definitions in 4.2): for restriction, the precondition gets narrowed down (strengthened), preserving the feasibility property $\underline{p} \subseteq \underline{post_p}$. For choice, the new precondition is the union of the preconditions, and the new postcondition is the union of the rounded postconditions; we apply *⁄Feasible_round⁄* from 3.5.

5.7 A note about choice

The choice operator is “angelic” (as expressed by the theorems *⁄Fail_choice⁄* and *⁄Fail_choiceonly⁄* in 5.1) in the sense that it will yield a result if either operand could do so.

This behavior is what we usually want: if we are writing an Internet routing program which can send a packet through either of two routers A and B, we will succeed if *at least one* of them is up and running. The definition of choice (last column in the table defining basic operators in *⁄Choice_def⁄*, section 4.2) reflects this property by using $\underline{p} \cup \underline{q}$ as the precondition $\underline{p \cup q}$ for the choice.

Another variant is “demonic” choice, which only guarantees a solution if *every* operand can do so. If we are using someone else’s packet delivery mechanism (rather than writing our own), and know that it may use either A or B, we can only assume that the packet will go through if *both* servers are up.

Demonic choice is easy to add to the present theory as a variant of choice in which the precondition of the choice is changed to $\underline{p} \cap \underline{q}$: the intersection, rather than the union, of the preconditions of the operands. A more complete exposition of the theory will include this variant.

With its “internal” and “external” choice operators, the CSP concurrency calculus [29] provides a distinction similar to angelic versus demonic. The explanation is that internal choice is made by the system and external by “the environment”. These concepts are informal and, in practice, difficult to teach. A mathematical distinction — union of preconditions versus their intersection — seems preferable as it removes any uncertainty.

The last axiom of the “Laws of Choice” by Hoare and colleagues cited earlier (1.4) suggests that those authors’ understanding of choice is demonic. This example provides further illustration of the dangers of a purely axiomatic approach: introducing both variants axiomatically would require a repetition of all the other choice axioms, since the operators share most of their other properties including associativity, commutativity and relationship with “;” and other operators). With the axiom-free and theorem-oriented approach of the present work, it suffices to add one definition for a new operator (specifying the new precondition); then its — possibly numerous — properties, instead of being tediously postulated, can be proved as theorems.

6 Refinement and related notions

Programs and specifications are fundamentally the same, but the reason they have traditionally been distinguished from each other is that they may exist at widely varying levels of abstraction. If we consider programs and specifications *relatively* to each other, the distinction makes sense: a program/specification may be a more concrete or more abstract version of another. In that case we might talk of the more concrete version as the program and of the more abstract one as its specification. The notions of refinement, specialization and implementation formalize these observations.

6.1 Refinement: preliminary notions

The following notions help define refinement and specialization.

Definition: state-extending, pre-weakening, post-strengthening

For programs q and p:

q **state-extends** p if $S_p \subseteq S_q$ *⁄Extend_state⁄*

q **pre-weakens** p if $\underline{p} \subseteq \underline{q}$ *⁄Weaken_pre⁄*

q **post-strengthens** p if $(\underline{p}\colon \underline{q}\colon post_q) \subseteq post_{\,p}$ *⁄Strengthen_post⁄*

Caveat: The terms "extends", "weakens" and "strenghtens" are convenient but should not mislead since all relationships use $\subseteq$, not $\subset$. So these terms should be understood as shorthand for "extends or retains" etc.

Comment: Post-strengthening (strengthening the precondition) means replacing the postcondition by a tighter one (in the sense of $\subseteq$, meaning one with as many or fewer input-output state pairs. This relation only applies, however, to input states that satisfy both preconditions. (A similar property applies to contract adaptation in redeclarations under inheritance in object-oriented programming [23].)

Unlike with earlier discussions, the definitions make the state explicit, as refinement and its variants may introduce new (concrete) states in addition to the original's abstract states.

6.2 Refinement, specialization, implementation

With the preceding auxiliary notions we can define three fundamental relations between programs:

Notation	**Name**	**Alternative names**	**Definition**
$q \sqsubseteq p$	q **refines** p	p **specifies** (or **abstracts**) q	q post-strengthens p and q pre-weakens p and q state-extends p */Refine_def/*
$q \subseteq p$	q **specializes** p	p **generalizes** q	q post-strengthens p and p pre-weakens q and p state-extends q */Special_def/*
	q **implements** p		q is feasible and refines p */Implement_def/*

Comment: Because they are based on subsetting, the notations for refinement and specialization use variants of the subset symbol "$\subseteq$". It is always clear from the operands whether "$\subseteq$" denotes "subset" on sets or "refine" on programs.

Justification: The difference between refinement and specialization is subtle but significant. Both strengthen the postcondition. Refinement — like the rules governing inheritance in object-oriented programming — accepts more states and a larger ("weaker") precondition, meaning that the refined version (q where $q \sqsubseteq p$) may have new behavior for states not in covered by the original precondition. With specialization, in contrast, the new version (q where $q \subseteq p$) has fewer behaviors. The literature contains many studies of refinement, but some of them — including by Hoare and colleagues [18, 13] — cover what is here called specialization. Refinements also figures in formal specification approaches such as Z and B; its definition for Z in the main reference on the topic takes up 98 book pages (pages 53 to 150 of [5]), making comparisons hazardous. Refinement as defined here is in line with other approaches, particularly from Dijkstra and Wirth [7, 4, 6, 31, 32].

Notation caveat: The literature often uses $q \sqsubseteq p$ to express that p refines q rather than the converse as here. The convention the present work (where $q \sqsubseteq p$ means that q refines p, and similarly for specialization) is consistent with the defining property: for refinement as well as for specialization, the postcondition of q is a subset of the postcondition of p, not the other way around. (A similarly unfortunate notational paradox exists in logic, where "p implies q" is sometimes written $p \supset q$ even though the implication means, if we view propositions as sets, that p is a subset of q or equal.)

6.3 Properties of refinement, specialization and implementation

The relations just introduced — refinement, specialization, implementation — satisfy some important properties.

Implementation theorem

A program having an implementation is feasible. */Implement_feasible/*

Notation caveat: This important theorem states (from the preceding definitions) that if q refines p and is feasible, then p itself is feasible.

Comment: The Implementation theorem actually works both ways: it can actually be extended to "a program is feasible if and only if it has an implementation", since the "only if" part is trivial (if a program is feasible, then because of the reflexivity of refinement, seen below, it has an implementation — itself).

Terminology:

- "Order relation" as used below denotes a non-strict possibly partial order: a relation "$\leq$" that is reflexive ($x \leq x$ for all x), antisymmetric (whenever $x \leq y$ and $y \leq x$ then $x = y$) and transitive (whenever $x \leq y$ and $y \leq z$ then $x \leq z$).
- The relations under study — "refines", "specializes", "implements" — are antisymmetric only under equivalence. In other words, taking refinement as an example, if $p \sqsubseteq q$ and $q \sqsubseteq p$ both hold, it is not necessarily the case that $p = q$ (because of possible spurious elements, "dead code", on either side), but equivalence $p \equiv q$ does hold. If a relation has this property and is otherwise reflexive and transitive, we will say that, rather than *being* an order relation, it **induces** an order relation. (Formally, the quotient of the relation by the equivalence relation "$\equiv$" is an order relation.)

Theorems: order relations

Refinement ("$\sqsubseteq$") induces an order relation. *⁄Refine_order⁄*

Specialization ("$\subseteq$") induces an order relation *⁄Special_order⁄*

Implementation induces an order relation
(except that it is reflexive for feasible elements only) *⁄Implementation_order⁄*

Comment: For brevity, the separate components of each of the above properties (reflexivity, antisymmetry, transitivity) are not individually included. They figure in the Isabelle/HOL repository with their proofs, under self-explanatory names consistent with the general conventions, such as *⁄Refine_transitive⁄*.

Definition: The following theorems will express when refinement, specialization and implementation are compatible with basic operations. An operation will be called **refinement-safe** if applying it preserves refinement properties. Similarly: **implementation-safe**, **specialization-safe**.

6.4 Refinement and basic operations

As an example:

Theorem: Refinement safety (for restriction)

If $q \sqsubseteq p$, then $C: q \sqsubseteq C: p$ *⁄Restrict_refinesafety⁄*
-- Restriction is refinement-safe.

Comment: On the other hand, many operations are not refinement-safe (although many will be specialization-safe as seen below) because of precondition widening. They include corestriction, composition and choice. In the last case a form of refinement-safety does hold per the following theorem.

Theorem: Refinement safety for choice

If $q \sqsubseteq p$ and q post-strengthens v, then $q \cup v \sqsubseteq p \cup v$ *⁄Choice_refinesafety⁄*

Implementation safety follows directly from refinement safety.

6.5 Specialization and basic operations

Unlike with refinement, all basic operations as well as corestriction are specialization-safe.

Theorems: Specialization safety

If $q \subseteq p$, then the following properties hold:

$C: q \subseteq C: p$ *⁄Restrict_specialsafety⁄*
-- Restriction is specialization-safe.

$C \setminus q \subseteq C \setminus p$ *⁄Corestrict_specialsafety⁄*
-- Coestriction is specialization-safe.

$q \cup v \subseteq p \cup v$

$v \cup q \subseteq v \cup p$ */Choice_ specialsafety/*

-- Choice is specialization-safe.

$q \ ; \ v \subseteq p \ ; \ v$

$v \ ; \ q \subseteq v \ ; \ p$ */Compose_ specialsafety/*

-- Composition is specialization-safe.

6.6 Restriction/corestriction under refinement and specialization

Comment: Even though they are very close, having the same effect on the postcondition, refinement and specialization have distinct and sometimes inverse effects on restriction and specialization.

Notation reminder: To avoid any confusion, note that "$\subseteq$" in properties such as $D \subseteq C$ for subsets of the state set S is the plain "subset" relation. On programs,"$\subseteq$" is specialization and "$\subseteq$" is refinement. As another reminder, "abstraction" was defined in section (6.2) as the inverse of refinement.

Theorems: restriction and corestriction under refinement and specialization

For conditions C and D such that $D \subseteq C$:

$C\!: p \subseteq p$

-- Restriction is a form of specialization. */Restrict_ special/*

$p \sqsubseteq C\!: p$

-- Restriction is a form of abstraction. */Restrict_ refine/*

-- Note reversal of order from */Restrict_ special/*.

$p \setminus C \subseteq p$

-- Corestriction is a form of specialization. */Corestrict_ special/*

$C\!: p \sqsubseteq D\!: p$ */Restrict_ refineorder/*

-- Under refinement, restriction reverses subsetting.

$D\!: p \subseteq C\!: pr$ */Restrict_ specialsubset/*

-- Under specialization, restriction retains subsetting.

$p \setminus D \subseteq p \setminus C$ */Corestrict_ specialsubset/*

-- Under specialization, corestriction retains subsetting.

Comment: The last two properties, governing specialization, have no direct counterparts for refinement.

6.7 Refinement/specialization and basic programs

Theorems: Basic programs under refinement and specialization

$Infeasible \sqsubseteq Skip \sqsubseteq Havoc \sqsubseteq Fail$ */Refine_ special/*

-- Full ordering of basic programs under refinement.

$Fail \subseteq Infeasible \subseteq Skip \subseteq Havoc$ */Special_ special/*

-- Their (different) ordering under specialization.

$p \sqsubseteq \underline{p}\!: \ Havoc$ */Refine_ havoccompose/*

$p \subseteq \underline{p}\!: \ Havoc$ */Special_ havoccompose/*

$p \sqsubseteq Havoc$ if p is total. /Refine_havoc*/

$p \subseteq Havoc$ /Special_havoc*/

$p \sqsubseteq Fail$ /Refine_fail/

$Fail \subseteq p$ /Special_fail/

$Fail \subseteq p$ if and only if $p \equiv Fail$. /Refine_failonly/

$p \subseteq Fail$ if and only if $p \equiv Fail$. /Special_failonly/

Comment: Section 3 noted that the basic programs (not only *Skip*) have variants such as $Fail_C$ limited to specific state subsets. Here are the corresponding properties.

Refinement and specialization properties of state-specific basic programs

For $D \subseteq C$:

$Skip_D \subseteq Skip_C$ $Skip_C \sqsubseteq Skip_D$

$Fail_D \subseteq Fail_C$ $Fail_C \sqsubseteq Fail_D$

$Infeasible_D \subseteq Infeasible_C$ $Infeasible_C \sqsubseteq Infeasible_D$ /Special_nonempty/

$Havoc_D \subseteq Havoc_C$ -- Not true for refinement.

$Fail_{False} = Infeasible_{False} = Skip_{False} = Havoc_{False}$

Comment: These properties again illustrate the difference between refinement and specialization. Refinement may introduce new behavior by weakening the precondition.

7 Conditional instructions

Among the basic operators, composition (";") is directly reflected in programming languages by the sequencing operator, usually written with the same symbol. As noted in section 4.2, the other two, restriction and choice, are fundamental building blocks but not generally used directly for programming. Actual programming languages generally use higher-level constructs: conditionals (this section) and loops (section 8).

7.1 Forms of conditional instruction

Comment: A conditional instruction is a combination of choice and restriction. It applies to an arbitrary number of restricted programs, called **branches**, with special case for one and two branches.

Convention: In the following definitions and theorems:

- C, D and C_1, .. ,C_n are conditions. C' is the complement of C.
- p, q and p_1, .. , p_n are programs.

Notation	Definition
if C_1: p_1, …, C_n: p_n **end**	$\bigcup_{i:=1}^{n} C_i: p_i$ /Conditional_set/
if C **then** p **else** q **end**	**if** C: p, C': q **end** /Conditional_two/
if C **then** p **end**	**if** C **then** p **else** *Skip* **end** /Conditional_one/

Definition: As noted, each C_i: p_i part of the basic conditional form is called a **branch**.

Extension: Not included, but easy to add, are variants with an arbitrary number of **elseif** clauses, with or without a final **else**.

Caveat: All the variants given have the effect of a *Skip* — in other words, no effect — if none of the conditions are satisfied. Dijkstra's version of the "guarded form" **if** C_1: p_1, ..., C_n: p_n **end** fails in that case. This behavior can be obtained by introducing a variant produces *Fail* if $\bigcup C_i$ has value *False*. The versions with **then** are not affected since they cannot fail unless one of their branches does.

7.2 Properties of conditional instructions

Convention: For the purpose of the present discussion, the general conditional (the first form above) may be abbreviated **if** C_i: p_i **end**.

Theorems: properties of conditionals

The conditional instruction **if** C_i: p_i **end** is equal to any other conditional obtained by permutation of its branches. */Conditional_commute/*

If $D_i \subseteq C_i$ for all i then (**if** D_i: p_i **end**) $\subseteq$ (**if** C_i: p_i **end**). */Conditional_subspecial/*

If $q_i \subseteq p_i$ for all i then (**if** C_i: q_i **end**) $\subseteq$ (**if** C_i: p_i **end**). */Conditional_multsubspecial/*
-- No such properties for refinement
-- (same reason as non-safety of choice, see 6.4).

C: p = **if** C: p **end** */Conditional_set_one/*
-- A plain **if** ... **then** ... **end** is just a restriction.

C_i: $p_i \subseteq$ (**if** C_1: p_1, ..., C_n: p_n **end**) (for $1 \leq i \leq n$). */Conditional_elements/*
-- Every branch of a conditional specializes it.

D: (**if** C_i: p_i **end**) $\equiv$ **if** $(D \cap C_i)$: p_i **end** */Conditional_distrib/*
-- Equivalence only because the order of applying
-- restriction and choice matters for dead code.

Justification: The following properties explain the use (from 3.8) of the alternative names "*True*" for S and *False* for $\varnothing$. Equivalence only.

Theorems: properties of special conditions for conditionals

if *True* **then** p_1 **else** p_2 **end** $\equiv p_1$ */If_true/*

if *False* **then** p_1 **else** p_2 **end** $\equiv p_2$ */If_false/*

8 Loops and invariants

Justification: Two key capabilities of computers are: to execute a operation repeatedly (and very fast); and to evaluate a condition — meaning formally, as we have seen, to perform a membership test between a given element (a state) and a given set. As a programming construct, the loop in its various forms captures this idea.

8.1 Loops: definition

Comment: To describe loops it is useful to start with a construct — fixed repetition — that reflects the standard mathematical practice of defining "power" as repeated self-application, and serves as a stepping stone for defining more general loops. (In the following definitions, p is an arbitrary program as usual.)

Name	Notation	Definition
Zero repetition	p^0	$Skip_{\underline{p}}$ /*Zero_rep*/
Fixed repetition (or: "power") (for $n > 0$)	p^n (Alternate notation: **for** i: $1..n$ **loop** p **end**)	$\Sigma_{i=1}^{n} \quad p$ /*Fixed_rep*/
Unbounded repetition	**loop** p **end**	$\bigcup_{i:\ \mathbb{N}} \quad p^i$ /*Arbitrary_rep*/
"From" loop	**from** a **until** C **loop** b **end**	a ; (**loop** $\neg C$: p **end**) $\setminus$ C /*From_loop*/
"Repeat" loop	**repeat** b **until** C **end**	**from** b **until** C **loop** b **end** /*While_loop*/
"While" loop	**while** C **loop** b **end**	**from** *Skip* **until** $\neg C$ **loop** b **end** /*Repeat_loop*/

Justification: A loop repeats a certain program (in practice, a program part) subject to an exit condition, or equivalently a continuation condition (these two forms of stating the condition are simply the negation of each other in the sense of the complement operation "$\neg$"). The **from** and **repeat** forms loop until an exit condition holds; the **while** form loops until a continuation condition does not hold. Both **from** and **while** forms may iterate the body b zero or more times; **repeat**, one or more times (it starts by executing b and only then asks whether to stop). The **from** form includes an often useful initialization stage b. All three forms are defined on the basis of the unbounded repetition construct (**loop** ... **end** which includes the zero-power p^0 among its possibilities. Although not common in programming languages, unbounded repetition exists for example in Ada.

Comment: The value for p^0 (/*Zero_rep*/) is required for consistency; see /*Power_inductive*/ below.

Caveat: While an informal description of the fixed-repetition construct is that it "executes p n times", in reality it executes it *at most* n times since the repetition will stop if after any number i of iterations ($0 \leq i \leq n$) $\underline{p}$ does not hold.

Theorems: loop properties

$p^{m+n} \;=\; p^m \;;\; p^n$ /*Power_inductive*/
-- For any $m, n \geq 0$. For $m = 0$, follows from
-- /*Zero_rep*/, /*Restrict_own*/ and /*Skip_comprestrict*/.

while C **loop** b **end** $\equiv \bigcup_{i\,\in\,\mathbb{N}} \;(C\!: b)^i \;\; \setminus \; \neg\, C$ /*Loop_choice*/

If $p^i \equiv Fail$, then $p^j \equiv Fail$ for all $j \geq i$ /*Repetition_fail*/
-- Program repetition cannot recover after any step has failed.

$\bigcup_{1\leq i}^{n} p^i \equiv Fail$ implies $\bigcup_{1\leq i}^{m} p^i \equiv Fail$ for all m and n. /*Loop_fail*/
If a loop fails, it fails for any number of iterations.

$\overline{\textbf{from}\ a\ \textbf{until}\ C\ \textbf{loop}\ b\ \textbf{end}} \subseteq C$ /*Loop_range*/
-- A terminating loop establishes its exit condition. See also /*Loop_correct*/.

If p and q are feasible, $\overline{p} \cap \underline{q} = \varnothing$ and $\overline{q} \cap \underline{p} = \varnothing$, then:
(**loop** $p \cup q$ **end**) = (**loop** p **end**) $\cup$ (**loop** q **end**) /*Loop_distrib*/
-- Under disjointness conditions, loop distributes over union.

Comment: The last result, /*Loop_distrib*/ can serve as a *parallelization* law in advance of the discussion of concurrency (section 10).

8.2 Invariants

Notation reminder: $r\ (C)$, for a relation r, is the image of a set C under r.

Definition: invariants

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

Justification: 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 $I \cap \underline{p}$, since results of p only matter when p starts in its precondition. The following theorems follow directly from the definition.

Invariant properties

If I is an invariant of p and $p \equiv q$, then I is an invariant of q. */Equiv_inv/*

Any I disjoint from $\underline{p}$ is an invariant of p . */Inv_disjoint/*

$True$ and $False$ are both invariants of p. */Inv_truefalse/*

$\overline{post_p}$ and $\overline{p}$ are both invariants of p. */Inv_subset/*
-- Follow from properties of relations, particularly */Image_restrict/*.

If I and J are invariants of p , then so are $I \cup J$ and $I \cap J$. */Inv_inter/*
-- Follow from */Image_union/* and */Image_inter/*.

If I is an invariant of p and $q \subseteq p$, then I is an invariant of $\underline{p}$: q. */Inv_special/*

If I is an invariant of p and $q \sqsubseteq p$, then I is an invariant of q. */Inv_refines/*

Some of the above results are cases of invariant-preserving operators in the following sense:

Definition and theorem: invariant-preserving operator, general invariant theorem

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

All the program operators defined so far are invariant-preserving. */Gen_inv/*

Comment: The operators covered are all those of section 4.2.

8.3 Loop invariants

The general notion of invariant just introduced is particularly important in the case of loops.

Definition: loop invariant

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

Justification: A loop invariant is ensured by the loop initialization (a) and preserved by the loop body (b), and hence will hold after any execution of the loop as a whole.

Notation reminder: $\overline{p}$ is short for $\overline{post_p\ /\ \underline{p}}$: the set of states that p actually produces (3.3).

Theorem: loop invariant properties

If I is a loop invariant of the loop L = **from** a **until** C **loop** b **end**, then
$\overline{L} \subseteq C \cap I$. */Loop_correct/*

Justification: This theorem is the fundamental property of loop invariants[22, 11]: 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). From this observation follows a program construction method which splits the desired final condition into these two parts and produces the corresponding loop.

A loop invariant of L = **from** a **until** C **loop** b **end** is also an invariant of L, of a, and of **loop** $\neg\ C$: b **end**. /Loop_invinv/

Comment: This theorem (which follows from /Inv_subset/ above) relates the notion of loop invariant to the more general notion of invariant.

9 Contracted programs

As noted at the beginning of section 6, the traditional distinction between specification and implementation is vague; the formal definitions of refinement, implementation and specialization provided the first steps towards establishing it on solid ground. To obtain a complete picture, we must also consider the criterion of *correctness*.

It makes no sense to ask whether a program/specification, by itself is correct. Correct with respect to what? In the traditional world of programming, we do know what correctness means for a program: it performs according to a stated specification. A relative notion.

What truly distinguishes a program from a specification, in the common usage of these terms, is neither just the level of abstraction nor the possibility of execution, but the existence of **two** programs/specifications in the sense of the present theory, such that one of them is a refinement of the other. More precisely a *feasible* refinement, or "implementation" (6.2 and /Implement_feasible/). The following notation reflects this analysis.

Definition and notation: correctness, contracted program

A program q is **correct for** a program p if it is an implementation of p. The following **contracted program** notation expresses this property: /Correct_def/

require $\underline{p}$ **do** q **ensure** $post_p$ **end**

Comment: The property is transitive in the sense of the following theorem.

Theorem: rule of consequence

A program v, correct for q which is correct for p, is correct for p. /Rule_consequence/

Comment: The following concepts provide a different perspective on the preceding definitions. They refer to a program p, a condition C and a relation r in $S \leftrightarrow S$.

Definition: strongest postcondition, weakest precondition,

$post_p \ / \ C$ is the **strongest precondition** of p for C, also written /Strongest_postcondition/

p **sp** C

$\underline{p}\ -\ \underline{(post_p - r)}$ is the **weakest precondition** of p for r, also written /Weakest_precondition/

p **wp** r

Explanation: $post_p - r$ is the set difference of two relations, giving us the set of pairs that belong to the first but not to the second. Its domain, $\underline{post_p - r}$, is the set of states for which p produces at least one result that r could never produce. Subtracting this domain from $\underline{p}$, the precondition of p , gives us the full set of states satisfying the precondition on which p is guaranteed to agree with r.

Convention: In the following properties, p and q are programs, C and D sets of states, r and t relations in $S \leftrightarrow S$. While an **sp** expression takes a set as its second operand, a **wp** expression needs a relation in $S \leftrightarrow S$; $True_{\leftrightarrow}$ will denote the full relation (also written $S \times S$ and $Univ\ [S]$) and $False_{\leftrightarrow}$ the empty relation.

Theorems: Properties of strongest postcondition and weakest precondition

If q implements p, then

q **sp** $\underline{p} \subseteq post_p$ /Post_charac/

and

$\underline{p} \subseteq q$ **wp** $post_p$ /Pre_charac/

-- Also expressible as q **sp** $\underline{p} \Rightarrow post_p$ and $\underline{p} \Rightarrow q$ **wp** $post_p$

p **sp** $False = False$ /Sp_false/

p **wp** $False_{\leftrightarrow} = False$ for feasible p /*Wp_false*/

p **wp** $False_{\leftrightarrow} = \underline{p} - \underline{post_p}$ /*Wp_infeas*/

$Fail$ **wp** $r = False$ /*Wp_fail*/

$Fail$ **sp** $C = False$ /*Sp_fail*/

p **sp** $True = post_p$ /*Sp_true*/

p **wp** $True_{\leftrightarrow} = \underline{p}$ /*Wp_true*/

$Havoc$ **sp** $C = post\ (Havoc)\ /\ C$ /*Sp_havoc*/

p **sp** $(C \cup D) = (p$ **sp** $C) \cup (p$ **sp** $D)$ /*Sp_distrib*/

p **wp** $(r \cup t) = (p$ **wp** $r) \cup (p$ **wp** $t)$ /*Wp_distrib*/

10 Concurrency

One of the most important but also most delicate mechanisms of programming is **concurrent operation**[13]. The reason concurrency is delicate is that it involves arbitrary *interleaving*. As concurrency textbooks usually explain at the outset (see e.g. [1]), when dealing with concurrency we do not need to assume that two events ever take place at the same time, or even ask whether that is actually possible: we only have to contend with not knowing which comes first. So stating that two elementary operations a and b execute concurrently simply means that the execution could be $a\ ;\ b$ or $b\ ;\ a$. This observation suggests a simple concurrency operator, although as we will see it is not general enough.

10.1 Definition: Atomic concurrency

Name	Notation	Definition
Atomic concurrency	$p \mid\mid\mid q$	$(p\ ;\ q) \cup (q\ ;\ p)$ /*Atomic_conc*/

Comment: Atomic concurrency is not associative (as can be seen by noting for example that $p \mid\mid\mid (q \mid\mid\mid r)$ includes among its choices $p\ ;\ r\ ;\ q$, which is not among the choices for $(p \mid\mid\mid q) \mid\mid\mid r)$. It can be generalized into an associative operator taking a list of operands, omitted here since it is not needed for the definition of the more general concurrency operator coming next. The following properties are the most important ones.

Theorems: Properties of atomic concurrency

$p \mid\mid\mid q = q \mid\mid\mid p$
-- Atomic concurrency is commutative. /*Atomic_commute*/

C: $(p \mid\mid\mid q) = (\ C$: $p) \mid\mid\mid (\ C$: $q)$
-- Restriction distributes over atomic concurrency. /*Atomic_restrict*/

$(p \mid\mid\mid q) \setminus C = (p \setminus C) \mid\mid\mid (q \setminus C)$
-- Corestriction distributes over atomic concurrency. /*Atomic_corestrict*/

$p \mid\mid\mid Fail = p \mid\mid\mid Fail = Fail$
-- $Fail$ absorbs atomic concurrency, left and right. /*Atomic_fail*/

[13] Also known as "parallel". There are technical differences between the two terms, which do not matter here.

10.2 Introducing a level of granularity

The "$|||$" operator gives a general idea of what concurrency is about but is not appropriate for the practical uses of concurrency. The problem is that the programs we want to run in parallel, say p and q, are generally themselves composite programs, and a practical notion of concurrency must specify at what level of granularity we allow their *components* to run in parallel.

If $p = (a\ ;\ b)$ and $q = (c\ ;\ d)$, then $p\ |||q$ only includes two executions: $(a\ ;\ b\ ;\ c\ ;\ d)$ and $(c\ ;\ d;\ a\ ;\ b)$. True concurrent execution would also include all interleavings in which all four appear as long as a is before b and c before d, including $(c\ ;\ a\ ;\ b\ ;\ d)$, $(a\ ;\ c\ ;\ b\ ;\ d)$ etc. As a simple example, consider programs using one of the most important resources of Constructor Institute of Technology, shown in Fig. 6 and supporting a variant of Alfred Renyi's maxim[14]: two software engineers are a device for turning coffee into a theory of programming.

Figure 6: **A parallel coffee machine**

The program p could be

$$\mathit{put\_capsule\_left}\ ;\ \mathit{push\_button\_left}\ ;\ \mathit{get\_coffee\_left}$$

and q the same with left replaced by right. Then $p\ |||q$ represents making two coffees one after the other, the only flexibility being afforded by allowing either order. In practice, however, the machine is "more parallel" than that: it allows any interleaving of the operations on both sides, $\mathit{put\_capsule\_x}$, $\mathit{push\_button\_x}$ and $\mathit{get\_coffee\_x}$ where x is left or right, separately preserving the given order for each x. (Practitioners of concurrency know that such a finer level of granularity can lead to problems, known as "data races", if these operations use shared resources, but we are not there yet.) As one of the variants tested as part of the preparation of this article, the following sequence, not covered by $p\ |||q$, was experimentally shown to work[15]:

$$\begin{array}{l}\mathit{put\_capsule\_right}\ ;\ \mathit{put\_capsule\_left}\ ;\ \mathit{push\_button\_left}\ ;\\ \mathit{push\_button\_right}\ ;\ \mathit{get\_coffee\_right}\ ;\ \mathit{get\_coffee\_left}\end{array}$$

These observations indicate that obtaining a general notion of concurrency requires choosing the appropriate level of granularity. We must accordingly express programs in a specific form.

10.3 Civilized programs and Choice Normal Form

Definitions: basic program, basis, civilized program

Given a set B of "**basic programs**" which includes Skip and Fail, a program is **civilized** with **basis** B if it of one of the following forms, where p and q are (recursively) civilized: */Civilized_def/*

- An element of B */Civilized_basic/*
- $p \cup q$ */Civilized_choice/*
- $p\ ;\ q$ */Civilized_sequence/*
- C: p */Civilized_restrict/*

[14] The original is about one mathematician and is often wrongly ascribed to Renyi's friend Paul Erdös.

[15] Order is preserved between the successive composition components of each operand of the $|||$, but not between those of different operands. The property, established for full concurrency as a theorem below */Conc_sound/*, is a general feature of concurrency: one cannot assume the existence of a global clock. In this particular execution trace, while the left button was pushed first, the corresponding coffee came out second. Perhaps left wanted a large coffee and right a small one.

The rest of this discussion assumes that we are dealing with civilized programs, for some basis. This assumption entails no loss of generality, since we can always include all the mechanisms we want — if not covered by sequence, choice or restriction — in the basis. (But we will not then be able to interleave their components.)

Any civilized program can be expressed in the following *normal form*:

Definition and theorem: Choice Normal Form (CNF)

Every civilized program can be expressed in Choice Normal Form as:

$$\bigcup_{i:\ I} \ \sum_{j:\ J_i} \ p_i^j \qquad /Normal\_form/$$

where every p_i^j is a basic program possibly with a restriction.

Explanation: "With a restriction" means of the form C: p for a condition C. The J_i index sets are finite; I can be infinite, but the discussion will limit itself to the finite case.

Justification: Intuitively, the Choice Normal Form represents expressing a program as the set of all its possible execution paths.

Terminology: Each Σ term of the overall $\bigcup$ will be called a **sum**. An empty sum denotes *Fail*. For non-empty sum s, s^1 is the first element and s^- the remaining (possibly empty) sum after removing the first element.

10.4 Fine-grained concurrency

On civilized programs, we can define a meaningful concurrency operator "$||$", which extends the atomic form "$|||$" by taking into account a chosen level of granularity.

Definition: Concurrent operation of two programs

The value of both $p \ || \ q$ and $q \ || \ p$ is defined as follows by structural induction on the definition of civilized programs (*/Civilized_def/*), always applying the first rule that matches:

- If both p and q are basic, possibly with a restriction, then $p \ ||| \ q$.
  -- This case uses atomic concurrency as defined in 10.1: $(p \ ; \ q) \cup (q \ ; \ p)$. */Conc_basic/*
- If p is C: x, then $(p' \ || \ q)$ where p' is p with one restriction removed through */Restrict_inter/*, */Restrict_distrib/* or */Compose_absorbrest/*. */Conc_restrict/*
- If p is $u \cup v$, then $(p \ || \ u) \cup (p \ || \ v)$. */Conc_choice/*
- If both p and q are sums, then
  $(p^1 \ ; \ (p^- \ || \ q)) \ \cup \ (q^1 \ ; \ (p \ || \ q^-))$. */Conc_sum/*

Notation: A frequently encountered notation for $p \ || \ q \ || \ \ldots$ in languages supporting concurrency is **parbegin** $p, q, \ldots$. **end**, or some other keyword instead of **parbegin**. The number of operands can be arbitrary thanks to the associativity of the operator, */Conc_assoc/* below.

Explanation: As noted, the cases have to be applied in order, using the first applicable one, the components then recursively subjected to the same rules. The four cases reflect the notion of interleaving:

- For basic operands */Conc_basic/*, parallel execution means that either of the two possible orders is possible.
- Case */Conc_basic/* can only apply to non-basic p (otherwise the preceding case would capture it). Then p must involve another operator; the rules on restriction will remove one restriction operator, enabling recursive application of the remaining cases (in the end restriction will only apply to basic operators, triggering the previous case).
- Executing p in parallel with either u or v means either executing p in parallel with u or executing p in parallel with v — as captured by */Conc_choice/*.
- As a result of the repeated recursive application of the previous cases we are only left with the case */Conc_sum/* of parallel execution of two Σ sums p and q. (If the operator on either side is any other than ";", one of the other cases kicks in.) Then parallel execution means that we have a choice between

only two possibilities: start with the first step of p and continue with the rest of p in parallel with all of q; and the other way around. Each is of course subject to recursive expansion through the four cases.

Defining the size of an expression as the number of its operators, every case reduces the size of at least one operand of every "$||$" subexpression, ensuring that the recursion will eventually terminate.

Example: Consider $(u \cup v) \;||\; (x \;;\; y)$ (or this expression with its operands of the "$\cup$" switched):

- It is subject to */Conc_choice/*, not */Conc_sum/*. The rule yields $(u \;||\; (x \;;\; y)) \cup (v \;||\; (x \;;\; y))$.
- Each of the two operands of the resulting choice is a parallel application of two sums (with one of the sums having only one element). Assuming all operands are basic, applying */Conc_sum/* twice yields $((u \;;\; x \;;\; y) \cup (x \;;\; u \;;\; y) \cup (x \;;\; y \;;\; u)) \cup ((v \;;\; x \;;\; y) \cup (x \;;\; v \;;\; y) \cup (x \;;\; y \;;\; v))$.
- Removing unneeded parentheses (per associativity of choice */Choice_assoc/*) puts this result in CNF.

Comment: The last example illustrates a fundamental property of the parallel operator expressed by the following theorem, which expresses when in parallel execution we can expect an original order of components to be preserved, and when not. The theorem relies on the notion of "occurs before" and "interleaved".

Terminology:

- A (civilized) program q is "**part of**" another if it is that program or (recursively) part of one of its operands in the corresponding case of the definition of civilized programs (*/Civilized_def/*).
- u "**occurs before** v in p" if some part of p is of the form $u' \;;\; v'$ where u is part of u' and v part of v'.
- p and q are "**interleaved**" in v if both p occurs before q and conversely in v.

For a program in CNF, "occurs before" can be stated non-recursively: u is an element of one of the sums of p, and v an element of that same sum, with a higher index. This definition, however, is more restricted since it only applies to basic programs.

10.5 Concurrency properties

Theorem: Concurrency soundness

If u occurs before v in p and x appears before y in q, then in $p \;||\; q$: */Conc_sound/*

- These two "occurs before" properties also hold. */Conc_soundpreserve/*
- Each of u and v is interleaved with each of x and y. */Conc_soundinterleave/*

Proof: Structural induction on the definition of "$||$". The theorem holds for basic programs (*/Conc_basic/*) and is preserved by each of the other cases if it holds for the operands.

Comment: The theorem is also notable for what it does not guarantee: any order property between individual components of p and q. Concurrency does not assume any global clock. Order of execution is preserved between sequentially ordered parts of each operand of the concurrency (each "concurrent process" in frequently used terminology), but there is no guaranteed order between parts of different operands.

In addition to this fundamental theorem, concurrent operation satisfies a number of important properties:

Theorems: concurrency properties

$p \;||\; q = q \;||\; p$ */Conc_commute/*
-- Concurrency is commutative.

$(p \;||\; q) \;||\; u = p \;||\; (q \;||\; u)$ */Conc_assoc/*
-- Concurrency is associative.

$p \;||\; (q \cup u) = (p \;||\; q\,) \cup (p \;||\; u)$ */Conc_choicedistrib/*
-- Concurrency distributes over choice.

$(p \;||\; q) \;;\; u \quad \subseteq \quad p \;||\; (q \;;\; u)$ */Conc_composeleft/*

$q \;;\; (p \;||\; u) \quad \subseteq \quad p \;||\; (q \;;\; u)$ */Conc_composeleftright/*

$$p \;;\; (q \;||\; u) \;\subseteq\; (p \;;\; q) \;||\; u \qquad /\mathit{Conc\_composeright}/$$

$$(p \;||\; u) \;;\; q \;\subseteq\; (p \;;\; q) \;||\; u \qquad /\mathit{Conc\_composerightleft}/$$

$$(p \;||\; q) \;;\; (u \;||\; v) \;\subseteq\; (p \;;\; u) \;||\; (q \;;\; v) \qquad /\mathit{Conc\_composegeneral}/$$

Comment: These theorems — in particular */Conc_ composegeneral/* — cover the "axiom of exchange" and all the other "laws of exchange" of the "Laws of Programming" work of Hoare and colleagues.

11 Conclusion and further work

The PRISM theory outlined above illustrate the two principal theses of the present work: that programming does not need a complex mathematical apparatus (as in most formal methods work), but is entirely definable as the combination of one relation and one set, on the sole basis of elementary set-theory concepts; and that such a description requires no new axioms whatsoever, all relevant properties being provable (and actually proved through a mechanical prover as part of the production of this article).

The part of the theory that we have presented covers fundamental programming constructs: the classical control structures of "structured programming" (sequence, conditionals and loops) as well as a general framework for concurrency. Work still lies ahead to cover other programming mechanisms, including those of imperative programming languages (such as assignment), functional programming languages, object-oriented programming, as well as specialized mechanisms such as exception handling. The theory can also serve as the basis for numerous programming tools and for providing facilities for trying out, compiling, interpreting and verifying programming ideas and constructs.

Acknowledgments: While section 1 has pinpointed perceived limitations of classic formal-methods approaches, particularly axiomatic ones, this work is obviously based on their insights and indebted to their pioneering authors. A talk on the topic to the IFIP WG2.3 working group elicited important suggestions by Mark Utting, Gary Leavens and other members.

References

[1] Jean-Raymond Abrial. *The B-book*. Cambridge university press, 1996.

[2] Ralph-Johan Back and Joakim Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.

[3] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming, 2nd edition*. Addison-Wesley, 2005.

[4] Ole-Johan Dahl, Edsger Wybe Dijkstra, and C. A. R. Hoare. *Structured programming*. Academic Press Ltd., 1972.

[5] John Derrick and Eerke A. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology (FACIT). Springer, London, 2 edition, 2014.

[6] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[7] Edsger W Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.

[8] Edsger W Dijkstra. Ewd 463: Some questions, 1974.

[9] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[10] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967.

[11] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, pages 277–300, 2010.

[12] Eric C.R. Hehner. *A Practical Theory of Programming.* Springer, 1993.

[13] C. A. R. Hoare. Unified theories of programming. In *Mathematical methods in program development*, pages 313–367. Springer, 1997.

[14] C. A. R. Hoare. Laws of programming with concurrency (slides). In *MSR Concurrency Workshop*, 06 2013.

[15] C. A. R. Hoare, Ian Hayes, He Jifeng, Carroll Morgan, A. Roscoe, Jeff Sanders, Ib Sørensen, J. Spivey, and Bernard Sufrin. Laws of programming. *Communications of the ACM*, 30:672–686, 08 1987.

[16] C. A. R. Hoare, Alexandra Mendes, and João F. Ferreira. Algebra, logic, geometry: at the foundations of computer science. In *Formal Methods Teaching, LNCS 11758*, pages 3–20. Springer, 2018.

[17] C. A. R. Hoare and Stephan van Staden. In praise of algebra. *Formal Aspects of Computing*, 24:423–431, 2012.

[18] C. A. R. Hoare and Stephan van Staden. The laws of programming unify process calculi. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction*, pages 7–22. Springer, 2012.

[19] Gilles Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Science*, 1987.

[20] Shaoying Liu. Extending operation semantics to enhance the applicability of formal refinement. In *Specification, Algebra and Software*, volume LNCS 8373, pages 434–440. Springer, 2014.

[21] Eliot Mendelson. *Introduction to Mathematical Logic (6th edition).* Chapman and Hall, 1976.

[22] Bertrand Meyer. A basis for the constructive approach to programming. In *IFIP Congress*, pages 293–298, 1980.

[23] Bertrand Meyer. *Object-Oriented Software Construction (first edition).* Prentice Hall, 1988.

[24] Bertrand Meyer. Theory of programs. In Bertrand Meyer and Martin Nordio, editors, *Proceedings of LASER summer schools 2007, 2008, 2013, 2014, Lectures on Software Engineering, Lecture Notes in Computer Science*, volume 8987. Springer, 2014.

[25] Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics).* Routledge, 2015.

[26] Carroll Morgan. *Programming from specifications.* Prentice-Hall, Inc., 1990.

[27] Carroll Morgan. *Programming from Specifications, Second edition.* Prentice Hall, 1998.

[28] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer, 2002.

[29] A.W. Roscoe. *Understanding Concurrent Systems: Communicating Sequential Processes.* Springer, London, 2010.

[30] Reto Weber. PRISM proofs repository (operational from 1 March 2025), February 2025. Available at: `https://github.com/CI-CSE/PRISM`.

[31] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[32] Niklaus Wirth. *Systematic Programming: An Introduction.* Prentice Hall PTR, USA, 1973.