Calculational Semantics: Deriving Programming Theories from Equations by Functional Predicate Calculus

RAYMOND T. BOUTE
INTEC, Ghent University

The objects of programming semantics, namely, programs and languages, are inherently formal, but the derivation of semantic theories is all too often informal, deprived of the benefits of formal calculation "guided by the shape of the formulas." Therefore, the main goal of this article is to provide for the study of semantics an approach with the same convenience and power of discovery that calculus has given for many years to applied mathematics, physics, and engineering. The approach uses *functional predicate calculus* and *concrete generic functionals*; in fact, a small part suffices. Application to a semantic theory proceeds by describing program behavior in the simplest possible way, namely by *program equations*, and discovering the axioms of the theory as theorems by calculation. This is shown in outline for a few theories, and in detail for axiomatic semantics, fulfilling a second goal of this article. Indeed, a chafing problem with classical axiomatic semantics is that some axioms are unintuitive at first, and that justifications via denotational semantics are too elaborate to be satisfactory. Derivation provides more transparency. Calculation of formulas for ante- and postconditions is shown in general, and for the major language constructs in particular. A basic problem reported in the literature, whereby relations are inadequate for handling nondeterminacy and termination, is solved here through appropriately defined program equations. Several variants and an example in mathematical analysis are also presented. One conclusion is that formal calculation with quantifiers is one of the most important elements for unifying continuous and discrete mathematics in general, and traditional engineering with computing science, in particular.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages

General Terms: Algorithms, Design, Documentation, Languages, Standardization, Theory, Verification

Additional Key Words and Phrases: Assignment, axiomatic semantics, calculational reasoning,

Author's address: R. T. Boute, INTEC, Universiteit Gent, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: boute@intec.UGent.be.

functional predicate calculus, generic functionals, intuitive semantics, loops, nondeterminacy, strongest postcondition, termination, weakest antecondition, formal semantics, programming theories, functional predicate calculus

1. INTRODUCTION

1.1 Goal: Bringing Formal Calculation into Semantic Theories

1.1.1 *Formal Calculation.* "Formal" means that expressions are handled on the basis of their *form* (syntax) by precise calculation rules, unlike informal handling on the basis of their *interpretation* (semantics) in some application domain.

As noted by Dijkstra [2000], Gries and Schneider [1993], Gries [1996], and many others [Dean and Hinchey 1996], formal calculation rules provide significant guidance in expression manipulation, as nicely captured by the maxim *"Ut faciant opus signa"* (let the symbols do the work) of the Mathematics of Program Construction conferences [Boiten and Möller 2002]. In other domains as well, we found the parallel syntactic intuition derived from regular practice with formal calculation an invaluable complement to the more common semantic intuition, especially when the latter is not yet mature, for example, when exploring new or unfamiliar domains.

Spectacular feats of formality are known in physics [Wigner 1960], Dirac's work being a salient example. More modestly, any student who enjoys classical physics will recall the excitement felt when calculus gave insight where intuition was clueless. In this process, we can identify three (often intertwined and alternating) phases:

(1) Express system behavior by equations that use only the basic laws.
(2) By formal calculation, derive consequences or explore issues that are not clear.
(3) Interpret the (intermediate, final) results of the calculations to feed intuition.

We forgo the custom of stressing obvious caveats, and just mention some further benefits of formal calculation: exploring ramifications, side-issues, refinements and other variants, and confirming or correcting intuitive ideas by "gedanken-experiments."

1.1.2 *Example: Colliding Balls.* We choose this example because its style of modelling systems by the states 'before' and 'after' will appear again in program equations.

Newton's Cradle is a desk adornment with five identical steel balls hanging in a straight row from a frame by pairs of strings in V-shape to inhibit lateral movement (Figure 1). If a number of balls at one end is moved away (keeping the strings taut) and released, after the collision the same number of balls will move away at the other end (and then fall back; the process repeats itself until

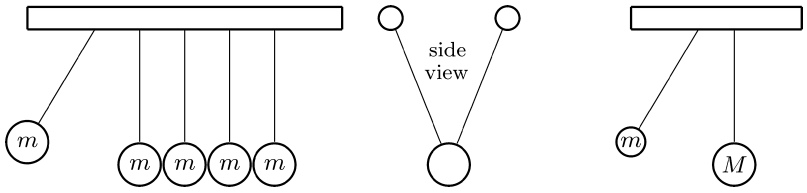

Fig. 1. Newton's Cradle and a variant.

all energy is spent due to inevitable losses). We consider a variant with two balls of mass m and M.

With obvious conventions, the state s is a pair of velocities: $s = v, V$. The states $\backprime s$ just before and s' just after collision are related by conservation of momentum and, neglecting losses, conservation of energy, respectively. Combined into one relation R,

$$\begin{aligned} R\,(\backprime s, s') \;\equiv\;& \quad m \cdot \backprime v + M \cdot \backprime V \;=\; m \cdot v' + M \cdot V' \\ & \wedge \;\; \frac{m \cdot \backprime v^2}{2} + \frac{M \cdot \backprime V^2}{2} = \frac{m \cdot v'^2}{2} + \frac{M \cdot V'^2}{2} \;\; . \end{aligned} \tag{1}$$

Letting $a := M/m$, and assuming $v' \neq \backprime v$ and $V' \neq \backprime V$ to discard the trivial case, we obtain by elementary high school algebra:

$$R\,(\backprime s, s') \;\Leftarrow\; v' - \backprime v = a \cdot (\backprime V - V') \;\wedge\; v' + \backprime v = \backprime V + V' \;\; .$$

Solving the righthand side as an equation with unknown s' for given $\backprime s$ yields

$$R\,(\backprime s, s') \;\Leftarrow\; v' = -\frac{a-1}{a+1} \cdot \backprime v + \frac{2 \cdot a}{a+1} \cdot \backprime V \;\wedge\; V' = \frac{2}{a+1} \cdot \backprime v + \frac{a-1}{a+1} \cdot \backprime V \;\; . \tag{2}$$

We consider two particular cases, assuming $\backprime s = w, 0$ (state just before the collision).

— Case $a = 1$. Then, $v' = 0$ and $V' = w$, so $s' = 0, w$ (the balls "switch roles").
— Case $a = 3$. Then $v' = -w/2$ and $V' = w/2$, hence $s' = -w/2, w/2$. Assuming the collision took place at the lowest position, the balls move to the same height h (with $h = \frac{w^2}{8 \cdot g}$), and return to collide at the same spot with $\backprime s = w/2, -w/2$, for which (2) yields $s' = -w, 0$. Starting with the next collision, the cycle repeats.

The crux is that mathematics is not used as just a "compact language" (a layman's view of mathematics), but that calculation yields insight hard to obtain by intuition.

1.1.3 *A Dichotomy in Applied Mathematics.* In situations illustrated by the example, reasoning is guided by symbolic calculation in an essentially formal way. This is common fare in physics and engineering *as far as concerns algebra or calculus*. It also explains why engineers so readily adopt software tools in these areas.

Yet, the formal calculations in differential and integral calculus sharply differ from the very informal style of the *logical* arguments usually given to support them.

Indeed, Taylor [2000] notes that the usual informal δ-ϵ proofs in analysis texts obscure the logical structure of the arguments, so "examiners fully deserve the garbage they get in return." He attributes the problem to *syncopation*, that is, using mathematical symbols (especially quantifiers) as mere shorthands for natural language, for example, $\forall$ and $\exists$ abbreviating "for all" and "there exists," respectively. This leaves predicate logic unexploited as a *calculation tool* for everyday mathematical practice.

1.1.4 *Logic in Theories of Program Semantics.* Literature samples [Gordon 2003; Loeckx and Sieber 1984; Meyer 1991; Winskel 1993] show that, with very rare exceptions [Dijkstra and Scholten 1990], the same happens in program semantics: predicate logic and quantifiers are used as mere notation, not as a true calculus.

If logic is "assumed known" or outlined at all, it is the traditional variant: not shown in action, and with proofs as informal as in analysis or left to the reader as "too lengthy." The consequences are more severe than in calculus, since the *application* of language semantics to practical problems is formal logic, as well. In such an incomplete conceptual setting, tools cannot be readily adopted. Identifying the cause is equally important.

Logic is suffering from the historical accident of never having had the opportunity to evolve into a proper calculus for humans [Dijkstra 2000; Gries 1996] before attention shifted to automation. In this way, an important, even crucial, evolutionary step is being skipped [Ravaglia et al. 1999].

Without the ability to do logic formally by "hand" calculation guided by the shape of the formulas, and especially when intuition is clueless, the use of mathematics in computing science can never achieve the effectiveness and sense of discovery provided by, for instance, the use of calculus in physics [Wigner 1960].

Hence, we want to make the benefits of formal calculation available to the *theory* of semantics, not just to the *applications* where it is obligatory, or even automated.[1]

1.1.5 *Objectives.* For a (new) formalism[2] to be a valuable intellectual investment, it must provide the following epistemological benefits.

(a) Have wide applicability: covering many different fields. As Bass [2003] notes, "*[relief] [in coping with the monumental growth of usable knowledge] is found in the use of abstraction and generalization [by] simple unifying concepts. This process has sometimes been called 'compression.*' Reducing fragmentation is a crucial issue."

(b) Handle simple things simply: in target areas, the formalization of conceptually simple topics should be equally simple (or perhaps reveal hidden complexities).

(c) Let the symbols do the work: the formal rules should disclose to programming and logic the benefits long appreciated in calculus as used in physics [Wigner 1960].

[1]Incidentally, this dichotomy exists for most automated tools: The object of automation is very formal, but the exposition and derivation of the theory and algorithms are very informal.

[2]By *formalism*, we always mean a language (or notation) together with formal manipulation rules.

1.1.6 *Approach.* The preceding objectives are met correspondingly as follows.

(a) We use a formalism covering both continuous (e.g., analysis) and discrete mathematics (including programs), rather than yet another specialist logic. The main elements are a functional predicate calculus [Boute 2002, 2006] and a small theory of concrete generic functionals [Boute 2003]. In fact, only a minor part of either will suffice for this article.

(b) Program behavior[3] is described in arguably the simplest possible way, namely by *program equations* having the same flavor as, for instance, equations in physics as exemplified by Equation (1), and circuit equations in electronics.

(c) Calculation enables deriving the axioms of postulational theories as theorems *without knowing them in advance* but by *discovering* them, guided by the rules.

We illustrate this for a few theories, with special attention paid to axiomatic semantics because it is the most classical, yet in its usual formulation leaves so many unanswered questions. Our approach will cover, in about a dozen pages with detailed proofs, a terrain requiring tenfold in other formalisms that leave proofs to the reader.

1.1.7 *Related Work.* Clearly, what basically distinguishes our work from that of others is the exploitation of formal calculation, guided by the shape of the formulas. Most similar in this respect is the predicate calculus from Dijkstra and Scholten [1990], but the latter is neither intended nor convenient for continuous mathematics. We mention some semantic theories that most resemble ours in objectives or technical aspects.

Hehner's [1999, 2004] theory is the only formulation of program behavior that does justice to the simplicity of basic language constructs in the same way as program equations. The similarity between the formulations and the fact that they arose independently are strong indications that they provide perhaps the simplest description of program behavior possible. Some necessary technical differences are discussed later. Termination is also handled very differently.

Axiomatic semantics [Hoare 1969; Dijkstra 1976; Dijkstra and Scholten 1990] is the theory chosen to be calculationally derived in detail because its formulation has been passed on nearly unchanged among generations [Dijkstra 1976; Cohen 1990; Meyer 1991], yet the axioms have remained rather more opaque than necessary. Some are un- or even counterintuitive at first, as noted by Gries and Schneider [1993], Gordon [2003], and others. The typical plausibility arguments given in words can equally well lead to other axioms that afterwards would turn out to model programs incorrectly.

Certain treatments make the axioms plausible via denotational semantics at the introductory level, as in Gordon [2003] and Meyer [1991], or the more advanced level [Loeckx and Sieber 1984; Winskel 1993], but at the cost of considerable technical detail in the intermediate phases, "leaving proofs to the reader" and tradeoffs between space and coverage. For instance, Gordon [2003] observes

[3]Strictly speaking, "program behavior" should be seen as a succinct way of saying "behavior of the program's execution on an abstract computer" (model for a reliable computer/compiler combination).

that simple relational formulations cannot handle nondeterminism and termination, and refers to more advanced work using power domains [Plotkin 1980] for a solution. Our approach shows how a separate equation for termination suffices.

1.1.8 *Overview.* Section 2 introduces the mathematical conventions sufficient to make the article self-contained. Section 3 illustrates our very first derivation of the "axioms" for assignment from an equation, unveiling all "mysteries" by formal calculation. Further sections generalize the idea. Section 4 presents a framework that formalizes program behavior in the simplest possible way by program equations. Section 5 shows how to express more abstract theories in this framework. Section 6 is an extended example showing how the axioms of axiomatic semantics are discovered by calculation. Finally, Section 7 further discusses some extensions, relations to other theories, and an example in continuous mathematics. The conclusion (Section 8) discusses the crucial role of formal calculation with quantifiers in unifying discrete and continuous mathematics.

2. CONVENTIONS AND FORMAL CALCULATION RULES

Our formalism, common to continuous and discrete mathematics, consists of two parts: generic functionals [Boute 2002, 2003] and functional predicate calculus [Boute 2002, 2006]. The cited documents explain the formalism in its generality, and show how the wide scope is achieved.

Practical applications in a diversity of domains require the complete formalism. However, for the current purpose, namely the calculational derivation of semantic theories, only a small part will suffice summarized here in a few paragraphs. The legend (f, g functions, X, Y sets, P, Q predicates, etc.) is clear from the context.

2.1 Simple Expressions, Substitution, and Proposition Calculus

2.1.1 *Simple Expressions.* These are either (i) *identifiers* (constants, variables) or (ii) function *applications* (prefix, infix). Examples are: c, $succ\, x$, $x + y$, $x \Rightarrow y$. We adopt the usual conventions, making certain parentheses optional. Infix operator precedence, in decreasing order, is: domain-specific (e.g., arithmetic +), relational (e.g., $\leq$, $=$), and logical; for these: $\wedge$, $\vee$ (equal precedence), then $\Rightarrow$, and lowest $\equiv$.

Substitution is the most basic syntactic operation on expressions [Boute 2002; Gries and Schneider 1993]. Letting d and e be expressions and v a variable, we write $d[^v_e$ for the expression obtained from d by substituting e for every occurrence of v, for instance, $(x + y)[^x_{u \cdot v} = (u \cdot v + y)$. Parallel substitution is similar, as in $(x + y)[^{x,y}_{u \cdot v,z} = (u \cdot v + z)$. An important use is *instantiation*: If p is a theorem (i.e., derived without hypotheses), then $p[^v_e$ is a theorem [Boute 2002; Gries and Schneider 1993].

2.1.2 *The Calculational Style.* Equational calculation, that is, formal calculation with $=$ only, is based on the equivalence properties of $=$ (reflexivity, symmetry, transitivity) and Leibniz's principle (replacing subexpressions by

equal subexpressions). Transitivity justifies chaining equalities in the calculational style:

$$\begin{aligned} \mathit{expression}_0 &= \langle \text{Justification}_0 \rangle\ \mathit{expression}_1 \\ &= \langle \text{Justification}_1 \rangle\ \mathit{expression}_2 \end{aligned}$$

The justification is an instantiated theorem, Leibniz's principle, or both combined. Up to the explicit justification discipline, this style is familiar from high school. Chaining by other relations (with compatible direction) also uses transitivity.

2.1.3 *Proposition Logic.* Logic is likewise made calculational, $\equiv$, $\Rightarrow$, and $\Leftarrow$ taking the roles of $=$, $\leq$, and $\geq$, respectively. As Gries [1996] notes, most logic texts present for formal logic just the axioms, elaborate technicalities (model theory), and then drop it all to handle further topics (e.g., cardinal and ordinal numbers) informally!

Calculational proposition logic (or *proposition calculus*) establishes a rich collection of algebraic laws that makes formal logic practical for routine use, and also by hand. Overviews can be found in Boute [2002] and Gries and Schneider [1993]. As in algebra, it pays off to introduce names for often-used rules (distributivity, isotony, etc.). Most of these are well-known, except perhaps for *shunting*:

$$\begin{aligned} &\text{Shunting } \Rightarrow\text{:} \quad x \Rightarrow y \Rightarrow z \ \equiv\ y \Rightarrow x \Rightarrow z \\ &\text{Shunting } \wedge\text{:} \quad x \Rightarrow y \Rightarrow z \ \equiv\ x \wedge y \Rightarrow z \end{aligned}$$

Note that $\Rightarrow$ is not associative, but by convention, $x \Rightarrow y \Rightarrow z$ stands for $x \Rightarrow (y \Rightarrow z)$. Given the precedences, the second rule is actually $(x \Rightarrow (y \Rightarrow z)) \equiv ((x \wedge y) \Rightarrow z)$.

Since it is advantageous to embed logic in arithmetic [Boute 1993], our logic constants are 0, 1, rather than F, T, but for this article either convention is adequate.

2.2 Sets, Functions, Generic Functionals, and Predicate Calculus

2.2.1 *Sets.* We treat sets formally, with the basic operator $\in$ and calculation rules defined or derived via proposition logic, such as $x \in X \cap Y \ \equiv x \in X \wedge x \in Y$. The empty set $\emptyset$ has the axiom $x \notin \emptyset$; the singleton set injector ι has the axiom $x \in \iota\, y \ \equiv x = y$.

2.2.2 *Bindings.* This is a technicality to cover before proceeding. A *binding* has the general form $i : X \wedge p$ (the $\wedge\, p$ is optional). It denotes no object by itself, but *introduces* or *declares* a (tuple of) identifiers(s) i, at the same time specifying that $i \in X \wedge p$. For instance, $n : \mathbb{Z} \wedge n \geq 0$ is interchangeable with $n : \mathbb{N}$. As explained elsewhere [Boute 2002, 2006], the common practice of overloading the relational operator $\in$ with the role of binding, as in $\{x \in \mathbb{Z} \mid x < y\}$, can lead to ambiguities; therefore, we always use : for binding.

Variables are identifiers bound in an abstraction *binding . expression* (see the following).

2.2.3 *Functions.* A function is *not* defined via its representation as a set of pairs (the function's *graph*), but as basic concept in its own right, characterized by a *domain* ($\mathcal{D}\ f$ for function f) and a *mapping* (image $f\ x$ for x in $\mathcal{D}\ f$).

For functions, the syntax of (i) *identifiers* and (ii) *applications* is augmented by (iii) *abstraction* and (iv) *tupling*, bringing the total of language constructs to four.

Ad (iii): *abstraction*, of the form *binding . expression*, is typed lambda-abstraction without redundant λ. It denotes a function with $d \in \mathcal{D}(v : X \mathrel{\mathbin{:}} p\,.\,e) \equiv d \in X \wedge p[^v_d$ for the domain and $d \in \mathcal{D}(v : X \mathrel{\mathbin{:}} p\,.\,e) \Rightarrow (v : X \mathrel{\mathbin{:}} p\,.\,e)\,d = e[^v_d$ for the mapping. We do not go into technicalities; the substitution rules are as in lambda calculus.

A trivial example is in specifying the *constant function definer* ($\bullet$) by $X \bullet e = v : X\,.\,e$, assuming $v \notin \varphi\, e$ (v not free in e). Particular forms are: the *empty function* ε with $\varepsilon = \emptyset \bullet e$ (any e) and the *one-point function definer* $\mapsto$ with $d \mapsto e = \iota\, d \bullet e$.

Abstractions may contain free variables (bound in outer contexts). Example: with $y : \mathbb{Z}$ as context, $x : \mathbb{Z} \mathrel{\mathbin{:}} x \geq y\,.\,x - y$ has domain $\{x : \mathbb{Z} \mid x \geq y\}$ and range $\mathbb{N}$.

Ad (iv): a *tuple* is any function with domain of the form $\square\, n$, the n first naturals. *Tupling* is the notation e, e', e'' (any number of expressions separated by commas), denoting a function with $\mathcal{D}(e, e', e'') = \square\, 3$ for the domain and $(e, e', e'')\,0 = e$ and $(e, e', e'')\,1 = e'$ and $(e, e', e'')\,2 = e''$ for the mapping. Furthermore, $\tau\, e = 0 \mapsto e$.

2.2.4 *Generic Functionals.* Generic functionals [Boute 2002, 2003] are general-purpose operators on functions. They differ from familiar variants by not restricting the argument functions, but instead, precisely defining the domain of the result. We use them to support point-free expression.

Direct extension ($\widehat{\ }$) extends any (infix) operator $\star$ to an operator $\widehat{\star}$ on functions:

$$f\ \widehat{\star}\ g\ =\ x : \mathcal{D}\ f \cap \mathcal{D}\ g \mathrel{\mathbin{:}} (f\ x, g\ x) \in \mathcal{D}(\star)\,.\,f\ x \star g\ x. \tag{3}$$

Example: if f and g are number-valued, $(f\ \widehat{+}\ g)x = f\ x + g\ x$ for any x in $\mathcal{D}\ f \cap \mathcal{D}\ g$.

Often, *half direct extension* (i.e., only one argument being a function) is wanted:

$$f\ \overleftarrow{\star} x = f\ \widehat{\star}\ (\mathcal{D}\ f \bullet x) \text{ and } x\ \overrightarrow{\star}\ f = (\mathcal{D}\ f \bullet x)\ \widehat{\star}\ f. \tag{4}$$

Composition ($\circ$), with $g \circ f = x : \mathcal{D}\ f \mathrel{\mathbin{:}} f\ x \in \mathcal{D}\ g\,.\,g\,(f\ x)$, covers direct extension ($\overline{\ }$) for single-argument functions: $\overline{g}\ f = g \circ f$. Example: $\overline{\neg}\ P = \neg \circ P$.

For function typing, a generic $\times$ [Boute 2003, 2006] generalizes the Cartesian product. Here, we just note $\times(X \bullet Y) = X \to Y$ and $\times(X, Y) = X \times Y$.

Function filtering ($\downarrow$) supports transition between pointwise and point-free styles; set filtering is similar. Convenient shorthands for $f \downarrow P$ and $X \downarrow P$ are f_P and X_P, respectively.

$$f \downarrow P = x : \mathcal{D}\ f \cap \mathcal{D}\ P \mathrel{\mathbin{:}} P\ x\,.\,f\ x \qquad X \downarrow P = \{x : X \cap \mathcal{D}\ P \mid P\ x\}. \tag{5}$$

This suffices for now; other generic functionals will be mentioned in passing.

Table I. Distributivity Laws for Quantification

Name of the Rule	Point-Free Form	Letting $P := v : X \,.\, p$ with $v \notin \varphi\, q$
Distributivity $\vee/\forall$	$q \vee \forall P \equiv \forall(q \mathbin{\overrightarrow{\vee}} P)$	$q \vee \forall(v : X \,.\, p) \equiv \forall(v : X \,.\, q \vee p)$
L(eft)-distrib. $\Rightarrow/\forall$	$q \Rightarrow \forall P \equiv \forall(q \mathbin{\overrightarrow{\Rightarrow}} P)$	$q \Rightarrow \forall(v : X \,.\, p) \equiv \forall(v : X \,.\, q \Rightarrow p)$
R(ight)-distr. $\Rightarrow/\exists$	$\exists P \Rightarrow q \equiv \forall(P \mathbin{\overleftarrow{\Rightarrow}} q)$	$\exists(v : X \,.\, p) \Rightarrow q \equiv \forall(v : X \,.\, p \Rightarrow q)$
P(seudo)-dist. $\wedge/\forall$	$q \wedge \forall P \equiv \forall(q \mathbin{\overrightarrow{\wedge}} P)$	$q \wedge \forall(v : X \,.\, p) \equiv \forall(v : X \,.\, q \wedge p)$

Table II. Algebraic Laws for Quantification

Name	Point-Free Form	Pointwise Form
Distrib. $\forall/\wedge$	$\forall(P \mathbin{\widehat{\wedge}} Q) \equiv \forall P \wedge \forall Q$	$\forall(x : X \,.\, p \wedge q) \equiv \forall(x : X \,.\, p) \wedge \forall(x : X \,.\, q)$
One-point rule	$\forall P_{=e} \equiv e \in \mathcal{D}\, P \Rightarrow P\, e$	$\forall(v : X \,.\, v = e \Rightarrow p) \equiv e \in X \Rightarrow p[^v_e$
Trading	$\forall P_Q \equiv \forall(Q \mathbin{\widehat{\Rightarrow}} P)$	$\forall(x : X \mathbin{\widehat{\wedge}} q \,.\, p) \equiv \forall x : X \,.\, q \Rightarrow p$
Nesting	$\forall Q \equiv \forall(\forall \circ Q^{\mathsf{C}})$	$\forall(x, y : X \times Y \,.\, p) \equiv \forall(x : X \,.\, \forall y : Y \,.\, p)$
Transp./Swap	$\forall(\forall \circ R) \equiv \forall(\forall \circ R^{\mathsf{T}})$	$\forall(x : X \,.\, \forall y : Y \,.\, p) \equiv \forall(y : Y \,.\, \forall x : X \,.\, p)$

2.2.5 *Functional Predicate Calculus.* *Predicates* are $\mathbb{B}$-valued functions, where $\mathbb{B} := \{0, 1\}$. *Relations* are predicates on tuples (allowing infix notation, as in $x\, R\, y$).

The *quantifiers* $\forall, \exists$ are predicates over predicates: for any predicate P, define

$$\forall P \;\equiv\; P = \mathcal{D}\, P \bullet 1 \qquad \exists P \;\equiv\; P \neq \mathcal{D}\, P \bullet 0 \quad . \tag{6}$$

Letting P be $v : X \,.\, p$ yields familiar expressions of the form $\forall v : X \,.\, p$. The abstraction may contain free variables, for example, $\forall x : \mathbb{Z} \,.\, y < x^2$ in the context of $y : \mathbb{Z}$ is a Boolean expression (type $\mathbb{B}$) that depends on y, and $\forall y : \mathbb{Z} \,.\, y < 0 \equiv \forall x : \mathbb{Z} \,.\, y < x^2$. Note: abstraction parses to the end, so this reads $\forall y : \mathbb{Z} \,.\, (y < 0 \equiv \forall x : \mathbb{Z} \,.\, y < x^2)$.

For every algebraic law, most elegantly stated in point-free form, a corresponding pointwise (familiar looking) form is obtained by substituting abstractions for P.

An example is *duality* or *generalized De Morgan's law*: Recalling that $\overline{\neg}\, P = \neg \circ P$, we have $\overline{\neg} \forall P = \exists(\overline{\neg}\, P)$ or, in point-wise form, $\neg(\forall v : X \,.\, p) \equiv \exists v : X \,.\, \neg\, p$.

An extensive collection of laws and their derivation can be found in Boute [2002, 2005]. All derivations are based on the single definition of *function equality* and on predicates being functions, which is why the calculus is called *functional*. For this article, a remarkably small part of the collection of laws suffices.

Table I lists the main distributivity laws (all have duals, not listed). As written in Table I, by lack of space, pseudodistributivity $\wedge/\forall$ assumes $\mathcal{D}\, P \neq \emptyset$. In general, $(p \wedge \forall P) \vee \mathcal{D}\, P = \emptyset \equiv \forall(p \mathbin{\overrightarrow{\wedge}} P)$. Also, $\varphi\, e$ is the set of free identifiers in e. Table II lists some more laws, which will be used later. As written in Table II, distributivity $\forall/\wedge$ assumes $\mathcal{D}\, P = \mathcal{D}\, Q$. In general, $\forall P \wedge \forall Q \Rightarrow \forall(P \mathbin{\widehat{\wedge}} Q)$; if $\mathcal{D}\, P = \mathcal{D}\, Q$, the converse holds. In the last two lines, $Q : X \times Y \to \mathbb{B}$ and $R : X \to Y \to \mathbb{B}$. Generic transposition ($^{\mathsf{T}}$) satisfies $(x : X \,.\, y : Y \,.\, e)^{\mathsf{T}} = y : Y \,.\, x : X \,.\, e$ and $^{\mathsf{C}}$ is currying.

2.3 Conventions for Use in Programming Theories

2.3.1 *Design Choices.* In most semantic formalisms used in practice, assertions, ante- and postconditions, etc., are expressions with program (and perhaps auxiliary) variables, for example, $\mathrm{x} > 3$ and $\mathrm{x} > 7$ in the Hoare triple $\{\mathrm{x} > 3\}\ \mathrm{x} := \mathrm{x} + 4\ \{\mathrm{x} > 7\}$.

Incidentally, we prefer the prefix "ante" over "pre" for several reasons: Latinist concerns ("ante" and "post," both with the accusative), common usage (A.M. and P.M.), and convenience (short symbols for often-used concepts physicists use m for *m*ass, v for *v*elocity, etc., so we use a and p for ante- and postconditions, respectively).

In the (relatively) standard terminology of logic, said expressions would be called *propositions*. Yet, some semantic theories refer to $\mathsf{wa}[\![\mathrm{x} := \mathrm{x} + 4]\!]$ as a *predicate* transformer. Depending on the case, forms like $\mathrm{x} > 7$ are either just *called* predicates [Hoare and Jifeng 1998], or *made* into predicates by taking all operators (e.g., $>$) as defined on *structures* [Dijkstra and Scholten 1990].

We shall do neither, but, to keep our formalism generally applicable (also outside computing) with little explanation, we respect common nomenclature: As in Section 2.1, *propositions* are Boolean *expressions* (perhaps with quantifiers), and *predicates* are Boolean-valued *functions*. Propositions can be made into predicates by abstraction, for example, $P := v : X \,.\, p$. Applications of predicates are propositions, for example, $P\ w \ \equiv\ p[^{v}_{w}$.

Predicates support a more pure style using application (e.g., $P\ w$), whereas working with propositions requires substitution (e.g., $p[^{v}_{w}$), a metalevel notion. Predicate transformers, for example, $Q : (X \to \mathbb{B}) \to (Y \to \mathbb{B})$, map mathematical objects, but proposition transformers like $\mathsf{wa}[\![\mathrm{x} := \mathrm{x} + 4]\!]$ map syntactic forms, for example, $\mathrm{x} > 7$ to $\mathrm{x} > 3$. For $P : \mathbb{B} \to \mathbb{B}$, clearly $p \Rightarrow P\,(p \wedge q) \ \equiv\ p \Rightarrow P\,q$, but for a transformer $T : B \to B$, on the syntactic category B of Boolean expressions, $p \Rightarrow T\,(p \wedge q) \ \equiv\ p \Rightarrow T\,q$ is clearly wrong (define, e.g., $T\ p = \forall\, x : X \,.\, p$ for all $p : B$), so caution is needed.

Our approach is unifying and hence, supports both styles. In view of direct applicability ($a\,/p$-conditions as propositions), we use the propositional style first. The cost of making calculation as convenient as it is for predicates is the need for the brief explanation to follow. An additional benefit is obviating the often overemphasized but unnecessary distinction in "flavor" between mathematical and program variables.

2.3.2 *Variables.* Variables are just identifiers, and we freely borrow the program variables to use them mathematically. When referring to different states in the same equation, the age-old practice of using markings ensures disambiguation, as illustrated by using $\backslash s$ and s' in the mechanics example of Equation (1).

Thus, the propositional style becomes just a matter of economy by *variable reuse*, doing mathematics (calculational reasoning) with program variables. For program variable v, we write $\backslash v$ for the alias of v before execution of a command, and v' for the alias after execution. This allows freely using both aliases in one equation without ambiguity. Accordingly, we write $\backslash e$ for expression e in

which every program variable v is properly replaced by the corresponding $\backprime v$, and similarly for e'.

In particular, let s be syntactic shorthand for the tuple of all program variables. Example: if $s = \mathrm{x}, \mathrm{y}$, then by the preceding convention $\backprime s = \backprime\mathrm{x}, \backprime\mathrm{y}$ and $s' = \mathrm{x}', \mathrm{y}'$. Observe also that $\backprime e = e[^{s}_{\backprime s}$ and $e' = e[^{s}_{s'}$ as a property of parallel substitution.

Substitutions and variable changes of this kind are familiar practice in engineering mathematics and systems modelling, but here we use them in a systematic way.

We shall see how calculation about semantics is now reduced to general predicate calculus and general-purpose calculation rules, requiring no "special" logics.

3. CALCULATING ASSIGNMENT AXIOMS FROM EQUATIONS

This first calculation, whose smooth development motivated the rest of this work, was meant to eradicate the mysterious looking elements from the assignment axiom.

Indeed, axiomatic semantics for assignment is unintuitive at first and even seems "backward," as noted by Gordon [2003], Gries and Schneider [1993], and others. It is puzzling to students, and intuitive arguments remain vague and unsatisfactory.

The difficulty is aggravated by the fact that certain "forward"-looking special cases are also correct, such as $\{v = d\}\ v{:=}e\ \{v = e[^{v}_{d}\}$, if $v \notin \varphi\, d$. Also puzzling to intuition are the dissimilar formulas for the weakest ante- and the strongest postcondition: Given postcondition p, the weakest antecondition is $p[^{v}_{e}$, and given antecondition a, the strongest postcondition is $\exists (u : V \,.\, v = e^{v}_{u} \ \wedge\ a[^{v}_{u})$.

Calculation from a simple intuitive equation will be shown so as to fully clarify all.

3.1 Deriving *ap*-Semantics from Program Equations

3.1.1 *The Equation for Assignment.* Let v be of type V, as specified by the declarations in the program. The equations for an assignment of the form $v{:=}e$ are

$$w' = \backprime w \ \wedge\ v' = \backprime e, \tag{7}$$

where $w : W$ is a variable not subject to assignment. In calculations, we found that one such w represents all, so let $\mathbf{S} := W \times V$ be the state space and s be w, v. Later, we will write this more generically. Equation (7) reflects intuitive understanding of program behavior in a style familiar from physics and engineering, as in Equation (1).

3.1.2 *Floyd-Hoare Semantics.* The usual axioms, resembling "pulling a rabbit out of a hat," can be replaced by a straightforward definition via program equations.

We shall define a Hoare-triple $\{a\}\ c\ \{p\}$ for a command c by formalizing the usual informal legend, namely, "for every possible execution of c starting in a

state satisfying a, the state after this execution satisfies p," in a direct, intuitive way.

The informal notion "possible execution of c" is made precise as saying that the pair $\backslash s, s'$ satisfies the program equations for c, written formally as $\mathsf{R}_c\,(\backslash s, s')$ with suitable predicate R_c on $\mathbf{S}^2$. Hence, the set of "possible executions of c" is $(\mathbf{S}^2)_{\mathsf{R}_c}$, and the intuitive legend for Hoare-triples can be formalized by transliterating it as $\{a\}\ c\ \{p\} \equiv \forall\,(\backslash s, s') : (\mathbf{S}^2)_{\mathsf{R}_c}\ .\ \backslash a \Rightarrow p'$, or by trading and nesting (see Table II),

Definition 3.1. $\{a\}\ c\ \{p\} \equiv \forall\,\backslash s : \mathbf{S}\,.\,\forall\, s' : \mathbf{S}\,.\,\mathsf{R}_c\,(\backslash s, s') \Rightarrow \backslash a \Rightarrow p'$.

From Equation (7), $\mathsf{R}_{v:=e}\,(\backslash s, s') \equiv w'{=}\backslash w \wedge v'{=}\backslash e$ (for $\mathbf{S} := W \times V$, $s := w, v$). We calculate

$$
\begin{array}{ll}
\{a\}\ v{:=}e\ \{p\} & \\
\quad \equiv \langle \text{Def. 3.1} \rangle & \forall\,(\backslash w, \backslash v) : W \times V\ .\ \forall\,(w', v') : W \times V\ .\ w' = \backslash w \wedge v' = \backslash e \Rightarrow \backslash a \Rightarrow p' \\
\quad \equiv \langle \text{Nest, swap} \rangle & \forall\,\backslash w : W\ .\ \forall\, w' : W\ .\ \forall\,\backslash v : V\ .\ \forall\, v' : V\ .\ w' = \backslash w \wedge v' = \backslash e \Rightarrow \backslash a \Rightarrow p' \\
\quad \equiv \langle \text{Shunt } \wedge \rangle & \forall\,\backslash w : W\ .\ \forall\, w' : W\ .\ \forall\,\backslash v : V\ .\ \forall\, v' : V\ .\ w' = \backslash w \Rightarrow v' = \backslash e \Rightarrow \backslash a \Rightarrow p' \\
\quad \equiv \langle \text{Ldist. } \Rightarrow/\forall \rangle & \forall\,\backslash w : W\ .\ \forall\, w' : W\ .\ w' = \backslash w \Rightarrow \forall\,\backslash v : V\ .\ \forall\, v' : V\ .\ v' = \backslash e \Rightarrow \backslash a \Rightarrow p' \\
\quad \equiv \langle \text{1-pt. rule} \rangle & \forall\,\backslash w : W\ .\ \forall\,\backslash v : V\ .\ \forall\, v' : V\ .\ v' = \backslash e \Rightarrow \backslash a \Rightarrow p'[^{w'}_{\backslash w}] \\
\quad \equiv \langle \text{Change var.} \rangle & \forall\, w : W\ .\ \forall\,\backslash v : V\ .\ \forall\, v' : V\ .\ v' = e[^{v}_{\backslash v}] \Rightarrow a[^{v}_{\backslash v}] \Rightarrow p[^{v}_{v'}]
\end{array}
$$

Hence we found the following theorem.

THEOREM 3.2. $\{a\}\ v{:=}e\ \{p\} \equiv \forall\, w : W\ .\ \forall\,\backslash v : V\ .\ \forall\, v' : V\ .\ v'{=}e[^{v}_{\backslash v}] \Rightarrow a[^{v}_{\backslash v}] \Rightarrow p[^{v}_{v'}]$

3.1.3 *Calculating the Weakest Antecondition.* We calculate a into an antecedent:

$$
\begin{array}{lll}
\{a\}\ v{:=}e\ \{p\} & \equiv \langle \text{Thm. 3.2} \rangle & \forall\, w : W\ .\ \forall\,\backslash v : V\ .\ \forall\, v' : V\ .\ v'{=}e[^{v}_{\backslash v}] \Rightarrow a[^{v}_{\backslash v}] \Rightarrow p[^{v}_{v'}] \\
& \equiv \langle \text{Shunting } \Rightarrow \rangle & \forall\, w : W\ .\ \forall\,\backslash v : V\ .\ \forall\, v' : V\ .\ a[^{v}_{\backslash v}] \Rightarrow v'{=}e[^{v}_{\backslash v}] \Rightarrow p[^{v}_{v'}] \\
& \equiv \langle \text{Ldist. } \Rightarrow/\forall \rangle & \forall\, w : W\ .\ \forall\,\backslash v : V\ .\ a[^{v}_{\backslash v}] \Rightarrow \forall\, v' : V\ .\ v'{=}e[^{v}_{\backslash v}] \Rightarrow p[^{v}_{v'}] \\
& \equiv \langle \text{One-pt. rule} \rangle & \forall\, w : W\ .\ \forall\,\backslash v : V\ .\ a[^{v}_{\backslash v}] \Rightarrow e[^{v}_{\backslash v} \in V \Rightarrow p[^{v}_{e[^{v}_{\backslash v}]}] \\
& \equiv \langle \text{Hypothesis} \rangle & \forall\, w : W\ .\ \forall\,\backslash v : V\ .\ a[^{v}_{\backslash v}] \Rightarrow p[^{v}_{e[^{v}_{\backslash v}]}] \\
& \equiv \langle \text{Change var.} \rangle & \forall\, w : W\ .\ \forall\, v : V\ .\ a \Rightarrow p[^{v}_{e}].
\end{array}
$$

The hypothesis is that $v := e$ is type correct, namely, $\forall\,\backslash v : V\ .\ e[^{v}_{\backslash v} \in V$, to be verified by static type checking; otherwise, the result is $a \Rightarrow e \in V \Rightarrow p[^{v}_{e}]$. This proves

THEOREM 3.3. $\{a\}\ v{:=}e\ \{p\} \equiv \forall\, s : \mathbf{S}\,.\,a \Rightarrow p[^{v}_{e}]$.

So $p[^{v}_{e}]$ is, at most, as strong as any antecondition a. Also, $p[^{v}_{e}]$ is an antecondition since $\{p[^{v}_{e}]\}\ v{:=}e\ \{p\} \equiv \forall\, s : \mathbf{S}\,.\,p[^{v}_{e}] \Rightarrow p[^{v}_{e}]$. This justifies defining operator wa by

Definition 3.4. $\mathsf{wa}\,[\![v{:=}e]\!]\,p \equiv p[^{v}_{e}]$.

3.1.4 *Calculating the Strongest Postcondition.* Here, we make p a consequent:

$$\begin{array}{ll}
\{a\}\ v{:=}e\ \{p\} & \equiv \langle \text{Thm. 3.2}\rangle \quad \forall w:W\,.\,\forall \,\backprime v:V\,.\,\forall v':V\,.\,v'=e[^{v}_{\backprime v}\Rightarrow a[^{v}_{\backprime v}\Rightarrow p[^{v}_{v'}\\
& \equiv \langle \text{Shunting } \wedge\rangle \quad \forall w:W\,.\,\forall \,\backprime v:V\,.\,\forall v':V\,.\,v'=e[^{v}_{\backprime v}\wedge a[^{v}_{\backprime v}\Rightarrow p[^{v}_{v'}\\
& \equiv \langle \text{Swap } \forall/\forall\rangle \quad \forall w:W\,.\,\forall v':V\,.\,\forall \,\backprime v:V\,.\,v'=e[^{v}_{\backprime v}\wedge a[^{v}_{\backprime v}\Rightarrow p[^{v}_{v'}\\
& \equiv \langle \text{Rdist. } \Rightarrow/\exists\rangle \quad \forall w:W\,.\,\forall v':V\,.\,\exists(\backprime v:V\,.\,v'=e[^{v}_{\backprime v}\wedge a[^{v}_{\backprime v})\Rightarrow p[^{v}_{v'}\\
& \equiv \langle \text{Change var.}\rangle \forall w:W\,.\,\forall v:V\,.\,\exists(\backprime v:V\,.\,v=e[^{v}_{\backprime v}\wedge a[^{v}_{\backprime v})\Rightarrow p.
\end{array}$$

The variable $\backprime v$ cannot be eliminated. Considerations, as before, justify

Definition 3.5. $\mathsf{sp}\,[\![v{:=}e]\!]\,a \;\equiv\; \exists\,\backprime v:V\,.\,v=e[^{v}_{\backprime v}\wedge a[^{v}_{\backprime v}$.

3.1.5 *Interesting Excursions.* More examples illustrate how formal calculation effortlessly yields answers where informal reasoning or intuition is plodding. The first justifies the "forward" rule $\{v=d\}\ v:=e\ \{v=e[^{v}_{d}\}$, provided $v\notin\varphi\,d$.

$$\begin{array}{ll}
\{v=d\}\ v{:=}e\ \left\{v=e[^{v}_{d}\right\} & \\
\quad\equiv \langle \text{Thm. 3.2}\rangle \quad \forall w:W\,.\,\forall \,\backprime v:V\,.\,\forall v':V\,.\,v'=e[^{v}_{\backprime v}\Rightarrow \backprime v=d[^{v}_{\backprime v}\Rightarrow v'=e[^{v}_{d[^{v}_{\backprime v}}\\
\quad\equiv \langle v\notin\varphi\,d\rangle \quad \forall w:W\,.\,\forall \,\backprime v:V\,.\,\forall v':V\,.\,v'=e[^{v}_{\backprime v}\Rightarrow \backprime v=d\Rightarrow v'=e[^{v}_{d}\\
\quad\equiv \langle \text{Leibniz, bis}\rangle \forall w:W\,.\,\forall \,\backprime v:V\,.\,\forall v':V\,.\,v'=e[^{v}_{\backprime v}\Rightarrow \backprime v=d\Rightarrow e[^{v}_{\backprime v}=e[^{v}_{\backprime v}\\
\quad\equiv \langle p\Rightarrow 1\;\equiv\;1\rangle \quad 1.
\end{array}$$

The second excursion investigates to what extent "bouncing" ante- and postconditions yields the original assertion, that is, does $\mathsf{sp}\,c\,(\mathsf{wa}\,c\,p)\;=\;p$ and $\mathsf{wa}\,c\,(\mathsf{sp}\,c\,a)=a$, or similar? Here, we assume c is $v{:=}e$. Calculation yields the answer:

$$\begin{array}{ll}
\mathsf{sp}\,c\,(\mathsf{wa}\,c\,p) & \equiv \langle \text{Def. wa (3.4)}\rangle\ \mathsf{sp}\,c\,\left(p[^{v}_{e}\right)\\
& \equiv \langle \text{Def. sp (3.5)}\rangle\ \exists\,\backprime v:V\,.\,p[^{v}_{e[^{v}_{\backprime v}}\wedge v=e[^{v}_{\backprime v}\\
& \equiv \langle \text{Substitut.}\rangle \quad \exists\,\backprime v:V\,.\,p[^{v}_{e[^{v}_{\backprime v}}\wedge v=e[^{v}_{\backprime v}\\
& \equiv \langle \text{Leibniz}\rangle \quad \exists\,\backprime v:V\,.\,p[^{v}_{v}\wedge v=e[^{v}_{\backprime v}\\
& \equiv \langle \text{Dist. } \wedge/\exists\rangle \quad p\wedge\exists\,\backprime v:V\,.\,v=e[^{v}_{\backprime v}\\
\mathsf{wa}\,c\,(\mathsf{sp}\,c\,a) & \equiv \langle \text{Def. sp (3.5)}\rangle\ \mathsf{wa}\,c\,\left(\exists\,\backprime v:V\,.\,a[^{v}_{\backprime v}\wedge v=e[^{v}_{\backprime v}\right)\\
& \equiv \langle \text{Def. wa (3.4)}\rangle\ \left(\exists\,\backprime v:V\,.\,a[^{v}_{\backprime v}\wedge v=e[^{v}_{\backprime v}\right)[^{v}_{e}\\
& \equiv \langle \text{Substitut.}\rangle \quad \exists\,\backprime v:V\,.\,a[^{v}_{\backprime v}\wedge e=e[^{v}_{\backprime v}.
\end{array}$$

Hence, we have proved the following, where c is the assignment $v{:=}e$.

THEOREM 3.6. (a) $\mathsf{sp}\,c\,(\mathsf{wa}\,c\,p)\;\equiv\;p\wedge\exists\,\backprime v:V\,.\,v=e[^{v}_{\backprime v}$;
(b) $\mathsf{wa}\,c\,(\mathsf{sp}\,c\,a)\;\equiv\;\exists\,\backprime v:V\,.\,a[^{v}_{\backprime v}\wedge e=e[^{v}_{\backprime v}$.

Of course, we have the weaker forms $\mathsf{sp}\,c\,(\mathsf{wa}\,c\,p)\Rightarrow p$ and $a\Rightarrow\mathsf{wa}\,c\,(\mathsf{sp}\,c\,a)$.

Example: with declaration $\mathrm{y}:\mathtt{int}$, let $c\;:=\;[\![\mathrm{y}:=\mathrm{y}^2+7]\!]$ and $p\;:=\;[\![\mathrm{y}>11]\!]$ and $q\;:=\;[\![\mathrm{y}<7]\!]$ and $a\;:=\;[\![\mathrm{y}>2]\!]$. Then the following calculations are

instructive.

$$
\begin{array}{lll}
\mathsf{sp}\,c\,(\mathsf{wa}\,c\,p) & \equiv \langle \text{Thm. 3.6.a} \rangle & p \wedge \exists\, v : V \,.\, v = e[^{v}_{v} \\
& \equiv \langle \text{Def. } p, V, e \rangle & \mathrm{y} > 11 \wedge \exists\, x : \mathbb{Z} \,.\, \mathrm{y} = x^2 + 7 \ \text{(stronger than } p\text{!)} \\
\mathsf{sp}\,c\,(\mathsf{wa}\,c\,q) & \equiv \langle \text{Thm. 3.6.a} \rangle & q \wedge \exists\, v : V \,.\, v = e[^{v}_{v} \\
& \equiv \langle \text{Def. } q, V, e \rangle & \mathrm{y} < 7 \wedge \exists\, x : \mathbb{Z} \,.\, \mathrm{y} = x^2 + 7 \\
& \equiv \langle \text{Dist. } \wedge/\exists \rangle & \exists\, x : \mathbb{Z} \,.\, \mathrm{y} < 7 \wedge \mathrm{y} = x^2 + 7 \\
& \equiv \langle \text{Leibniz} \rangle & \exists\, x : \mathbb{Z} \,.\, x^2 + 7 < 7 \wedge \mathrm{y} = x^2 + 7 \\
& \equiv \langle \text{Arithmetic} \rangle & \exists\, x : \mathbb{Z} \,.\, x^2 < 0 \wedge \mathrm{y} = x^2 + 7 \\
& \equiv \langle \forall\, x : \mathbb{Z} \,.\, x^2 \geq 0 \rangle & \exists\, x : \mathbb{Z} \,.\, 0 \wedge \mathrm{y} = x^2 + 7 \\
& \equiv \langle 0 \wedge p \equiv 0 \rangle & \exists\, x : \mathbb{Z} \,.\, 0 \\
& \equiv \langle \exists\, (X \bullet 0) \equiv 0 \rangle & 0 \ \text{(strongest of all propositions)} \\
\mathsf{wa}\,c\,(\mathsf{sp}\,c\,a) & \equiv \langle \text{Thm. 3.6.b} \rangle & \exists\, v : V \,.\, a[^{v}_{v} \wedge e = e[^{v}_{v} \\
& \equiv \langle \text{Def. } a \text{ and } e \rangle & \exists\, x : \mathbb{Z} \,.\, x > 2 \wedge \mathrm{y}^2 + 7 = x^2 + 7 \\
& \equiv \langle \text{Arithmetic} \rangle & \exists\, x : \mathbb{Z} \,.\, x > 2 \wedge (x = \mathrm{y} \vee x = -\mathrm{y}) \\
& \equiv \langle \text{Distr. } \wedge/\vee \rangle & \exists\, x : \mathbb{Z} \,.\, (x = \mathrm{y} \wedge x > 2) \vee (x = -\mathrm{y} \wedge x > 2) \\
& \equiv \langle \text{Distr. } \exists/\vee \rangle & \exists\, (x : \mathbb{Z} \,.\, x = \mathrm{y} \wedge x > 2) \vee \exists\, (x : \mathbb{Z} \,.\, x = -\mathrm{y} \wedge x > 2) \\
& \equiv \langle \text{One-pt. rule} \rangle & (\mathrm{y} \in \mathbb{Z} \wedge \mathrm{y} > 2) \vee (-\mathrm{y} \in \mathbb{Z} \wedge -\mathrm{y} > 2) \\
& \equiv \langle \text{Arithmetic} \rangle & (\mathrm{y} \in \mathbb{Z} \wedge \mathrm{y} > 2) \vee (\mathrm{y} \in \mathbb{Z} \wedge \mathrm{y} < -2) \\
& \equiv \langle \text{Distr. } \exists/\vee \rangle & \mathrm{y} \in \mathbb{Z} \wedge (\mathrm{y} > 2 \vee \mathrm{y} < -2) \ \text{(compare with } a\text{)}
\end{array}
$$

We emphasized logic (illustrating the algebraic style of the calculation) over arithmetic (where algebraic calculation is commonplace). When calculational logic has become familiar, steps can be skipped in the same way as in algebra and analysis.

4. DESCRIBING BEHAVIOR BY PROGRAM EQUATIONS

We formalize program behavior in the simplest possible way that allows expressing (Section 5) and deriving (Section 6) other, more "abstract" theories.

4.1 Program Equations for State Changes

4.1.1 *Principle.* The goal is formulating program behavior in the simplest way possible. The result is compact, clear, without extraneous elements, and its similarity to an independently developed formulation [Hehner 1999, 2004] indicates that this has been achieved.

Many of our design decisions are based on the following analogy with circuits.

Aspect modelled	Circuits	Programs
Behavior of components	Device equations	Equations for assignment
Behavior of interconnections	Kirchhoff's laws	Equations for composition, etc.

The style is both denotational and operational, obviating the traditional distinctions. The axiomatic style [Dijkstra 1976] for describing and deriving *programs* is covered by calculation.

Table III. Intuitive Understanding of Program Behavior

Name	Syntax (Command c)	Intuitive Behavior
Assignment	$v := e$	Assign to variable v the value e.
No change	`skip`	Leave all variables as they are.
Composition	$c' ; c''$	Do c' and then do c''.
Choice	`if` $[]\, i : I \,.\, b_i$ `->` c'_i `fi`	For one of the b_i that holds, do c'_i.
Repetition	`do` b `->` c' `od`	If b holds, do c' and start again, else skip.

Table IV. Formalization by Program Equations

	Name	Syntax (Command c)	$\mathsf{R}_c\,(\backslash s, s')$ or Equivalent Program
(a)	Assignment	$v := e$	$s' = \backslash s[^{\backslash v}_{\backslash e}$
(b)	No change	`skip`	$s' = \backslash s$
(c)	Composition	$c' ; c''$	$\exists s \bullet \mathsf{R}_{c'}\,(\backslash s, s) \wedge \mathsf{R}_{c''}\,(s, s')$
(d)	Choice	`if` $[]\, i : I \,.\, b_i$ `->` c'_i `fi`	$\exists i : I \,.\, \backslash b_i \wedge \mathsf{R}_{c'_i}\,(\backslash s, s')$
(e)	Repetition	`do` b `->` c' `od`	`if` b `->` $(c' ; c)$ $[]$ $\neg b$ `->` `skip` `fi`

4.1.2 *States.* In the denotational view [Gordon 2003; Loeckx and Sieber 1984; Winskel 1993], a state is a function $s : \mathit{Variable} \to \mathit{Value}$.

The following more directly reflects the program view. We let the state space $\mathbf{S}$ be the Cartesian product induced by the variable declarations in a way that is most elegantly expressed by generic functionals [Boute 2003, 2006]. Here, a self-evident example suffices:

Given:	declaration	`var x : int, b : bool`
Derived:	state space	$\mathbf{S} = \mathbb{Z} \times \mathbb{B}$
	state tuple	$s = \mathtt{x}, \mathtt{b}$

We shall use s as the shorthand for the tuple of all program (and auxiliary) variables, v as the shorthand for a subtuple, and S_v for its type, in particular, $\mathbf{S} = S_s$.

Often we abbreviate $s : \mathbf{S} \,.\, e$ as $s \bullet e$. Clearly, $(s \bullet e) r \;=\; e[^s_r$ for any r in $\mathbf{S}$, and $(s \bullet e) s \;=\; e$; the latter choice of variables is sometimes exploited in the sequel.

4.1.3 *Program Behavior.* The "logical" behavior of a program (command) c is formalized by a function $\mathsf{R}\_ : C \to \mathbf{S}^2 \to \mathbb{B}$, with $\mathsf{R}_c\,(\backslash s, s')$ specifying the possible executions of c starting in $\backslash s$ and terminating in s'. Other program aspects can be expressed similarly [Boute 1988].

We choose Dijkstra's [1976] guarded command language as a running example for its rich characteristics, including nondeterminacy, with few constructs.

The syntax and intuitive behaviors of these constructs are shown in Table III. Clearly, in the name c' for (families of) commands, the accent is not a state marking. The header "intuitive behavior" reflects the programmer's understanding, but the entries can also be seen as language or machine (compiler + computer) specifications.

Formalization by equations is shown in Table IV, using only simple relations, no special logics or semantic theories. Note how directly the equations reflect intuitive understanding. This is an optimal common ground for comparing language designs.

We briefly point out a technicality. In specifying R_c, quantification is understood: $\forall\, \backslash s \bullet \forall s' \bullet \mathsf{R}_c\,(\backslash s, s') \equiv$ (defining expression). In (b) and (c), and in more abstract theories, $\backslash s$, s, s' may be just dummies. In (a) and (d), which involve $\backslash v$, $\backslash e$, $\backslash b_i$, the symbol s is the tuple of program variables, which is important when instantiating. This is illustrated in the first of the following remarks on (a), (d), and (e).

ad (a) Assignment is assumed generalized to *multiple assignment*, with v a tuple of variables and e a tuple of expressions (matched in length and type). The first equation in Table IV takes this in its stride. Substitutions are illustrated by

$$\begin{aligned} \mathsf{R}_{\mathtt{i,j\ :=\ i+j,j+k}}\,((\backslash \mathtt{i}, \backslash \mathtt{j}, \backslash \mathtt{k}, \backslash \mathtt{l}), (\mathtt{i}', \mathtt{j}', \mathtt{k}', \mathtt{l}')) \\ \equiv\ \ \mathtt{i}', \mathtt{j}', \mathtt{k}', \mathtt{l}' = \backslash \mathtt{i} + \backslash \mathtt{j}, \backslash \mathtt{j} + \backslash \mathtt{k}, \backslash \mathtt{k}, \backslash \mathtt{l}, \end{aligned}$$

where `k` and `l` represent variables not appearing to the left of the `:=` sign.

ad (d) In the choice construct, $[]$ is the prefix operator for choice, b a family of propositions (on program variables), and c' a family of commands. The domain I is an indexing set, say, $\Box\, n$ for n alternatives. As in Boute [2002, 2006], when appropriate, a prefix operator F has a variadic infix version f defined by $e\ f\ e'\ f\ e'' = F\,(e, e', e'')$. In particular, b_0 `->` c'_0 $[]$ b_1 `->` c'_1 $[]$ b_2 `->` $c'_2 = [] \ i : \Box\, 3 \bullet b_i$ `->` c'_i, etc.

A variant is defined by `if` b `then` c' `else` c'' `fi` = `if` b `->` c' $[]$ $\neg b$ `->` c'' `fi`; here, b is just a proposition. Similarly, `if` b `then` c `fi` = `if` b `->` c $[]$ $\neg\ b$ `->` `skip` `fi`.

ad (e) In the repetition construct, b is a proposition. For $c := [\![\mathtt{do}\ b\ \mathtt{->}\ c'\ \mathtt{od}]\!]$, $c =$ `if` b `, then` c'`;`c `fi`. The recursive form raises interesting issues discussed later.

4.1.4 *Brief Note on Algebraic Issues.* Equivalence of programs (commands), with respect to a behavioral model $M : C \to \mathbf{S}^k \to \mathbb{B}$, is defined by $c \sim_M c' \equiv M\,c = M\,c'$.

Given the basic status of R, we define $c \approx c' \equiv c \sim_{\mathsf{R}} c'$. Finally, we write $c \simeq c'$ if c and c' are fully interchangeable (in Leibniz's sense) in the considered theory.

Nearly all (imperative) languages have a command like `skip`, either explicit or via any assignment $v := v$. It is both a left and right identity of composition: `skip`$\,;c \approx c$ and $c\,;$ `skip` $\approx c$, assuming the composition equation from Table IV.

On algebraic grounds, it is also useful to have (at least conceptually) a command acting as a left and right zero. To this effect, one can introduce `abort` with equation $\mathsf{R}_{\mathtt{abort}}\,(s, s') \equiv 0$, indeed resulting in `abort`$\,;c \approx$ `abort` and $c\,;$`abort` $\approx$ `abort`. The name reflects that the set $(\mathbf{S}^2)_{\mathsf{R}_{\mathtt{abort}}}$ of possible executions is empty.

Establishing or preserving useful algebraic properties provides guidance in design.

Table V. Program Equations for State Change Augmented by Termination

Syntax	Behavior (Program Equations or Equivalent Program)	
Command c	State Change $\mathsf{R}_c\,(\backprime s, s')$	Termination $\mathsf{T}_c\,\backprime s$
$v := e$	$s' = \backprime s[{}^{v}_{e}]$	1
`skip`	$s' = \backprime s$	1
`abort`	0	0
$c'\,;c''$	$\exists\, s \bullet \mathsf{R}_{c'}\,(\backprime s, s) \wedge \mathsf{R}_{c''}\,(s, s')$	$\mathsf{T}_{c'}\,\backprime s \wedge \forall\, s \bullet \mathsf{R}_{c'}\,(\backprime s, s) \Rightarrow \mathsf{T}_{c''}\,s$
`if` $[\!]\, i : I \,.\, b_i$ `->` c'_i `fi`	$\exists\, i : I \,.\, \backprime b_i \wedge \mathsf{R}_{c'_i}\,(\backprime s, s')$	$\exists (i : I \,.\, \backprime b_i) \wedge \forall\, i : I \,.\, \backprime b_i \Rightarrow \mathsf{T}_{c'_i}\,\backprime s$
`do` b `->` c' `od`	`if` $\neg b$ `->` `skip` $[\!]$ b `->` $(c'\,;c)$ `fi`	

4.2 Program Equations for Theories with Termination

4.2.1 *Termination Equations.* Such equations for the guarded command language are shown in the rightmost column of Table V, expressed via $\mathsf{T}\_ : C \to \mathbf{S} \to \mathbb{B}$. For maximal practical relevance, we made the design decision that termination should be *guaranteed*, not just possible. This is reflected in an intuitively clear fashion by the equation for composition: $\mathsf{T}_{c';c''}\,\backprime s \equiv \mathsf{T}_{c'}\,\backprime s \wedge \forall\, s \bullet \mathsf{R}_{c'}\,(\backprime s, s) \Rightarrow \mathsf{T}_{c''}\,s$.

We never attempt to express termination via the state change relation. This avoids problems, as we discovered later by reading Gordon [2003], who observes that termination for nondeterminacy cannot be adequately handled just by a state change relation. As we shall see, in our approach, introducing T suffices.

4.2.2 *Algebraic Issues.* Assume $\simeq$ defined by $c \simeq c' \;\equiv\; c \sim_{\mathsf{R}} c' \wedge c \sim_{\mathsf{T}} c'$. We consider the consequences of requiring `abort` to be a left and right zero of ; w.r.t. $\simeq$. If we specify $\mathsf{T}_{\mathtt{abort}}\,\backprime s \;\equiv\; 0$, the left zero property $\mathtt{abort}\,;c \simeq \mathtt{abort}$ is immediate. The right zero property $c\,;\mathtt{abort} \simeq \mathtt{abort}$ is immediate for the contribution of R, but, because $\mathsf{T}[\![c\,;\mathtt{abort}]\!]\,\backprime s \;\equiv\; \mathsf{T}_c\,\backprime s \wedge \forall\, s \bullet \mathsf{R}_c\,(\backprime s, s) \Rightarrow \mathsf{T}_{\mathtt{abort}}\,s$, this also requires

$$\mathsf{T}_c\,\backprime s \Rightarrow \exists\, s \bullet \mathsf{R}_c\,(\backprime s, s). \tag{8}$$

As we shall see, this is equivalent to the law of the excluded miracle [Dijkstra 1976], or LEM. In Dijkstra [1976], although not in Dijkstra and Scholten [1990], LEM is required for all commands. For instance, for the choice construct it is ensured by $\exists\, i : I \,.\, \backprime b_i$ in the image definition of T.

5. EXPRESSING THEORIES VIA PROGRAM EQUATIONS

Here, we show how other, sometimes more "abstract," theories are expressed via program equations. Calculational derivation of properties is discussed in Section 6.

5.1 Hehner's Practical Theory of Programming

Confluence in viewpoints justifies special attention to the approach in Hehner [1999, 2004], called to our attention by B. Möller after the first ideas reported in this article were elaborated upon.

Table VI. Main Constructs for *A Practical Theory of Programming*

Name	Syntax and Semantics
Assignment	$v := e \ \equiv \ s' = s[^{v}_{e}$
No action	$ok \ \equiv \ s' = s$
Composition	$p\,;q \ \equiv \ \exists t \bullet p[^{s'}_{t} \wedge q[^{s}_{t}$
Choice	$\texttt{if}\ b\ \texttt{then}\ p \ \equiv \ (b \Rightarrow p) \wedge (\neg b \Rightarrow s' = s)$

The level of abstraction is the same as for program equations. As mentioned, the similarity suggests that the simplest possible formulation has been obtained.

There are differences, however. In Hehner's approach, constructs stand for their own semantics,[4] namely, propositions about the state before and after execution, using VDM conventions. The relation with our formulation could be expressed by $c \ \equiv \ \mathsf{R}_c\,(s, s')$. The main language constructs are defined in Table VI, translated into our conventions for the sake of uniformity. For instance, in Hehner [1999, 2004], assignment is defined by $x := e \ \equiv \ x' = e \wedge y' = y \wedge \ldots \wedge z' = z$ and "no action," written *ok* in the language, by $ok \ \equiv \ x' = x \wedge y' = y \wedge \ldots \wedge z' = z$.

As we shall see later, the beauty of this language is that it carries its own theory, requiring only proposition and predicate calculus. One calculates directly with program constructs, a feature previously found only in functional programming.

5.2 Expressing *ap*-Conditions in terms of Program Equations

5.2.1 *Formalizing Intuitive Understanding of Floyd-Hoare Semantics.* We write B for the set of propositions, and $a : B$ and $p : B$ for ante- and postconditions, respectively.

For *partial correctness*, that is, ignoring termination issues, Definition 3.1 formalizes the informal legend of Hoare triples, and is recalled here for easy reference.

Definition 5.1. $\{a\}\ c\ \{p\} \ \equiv \ \forall\, \backslash s \bullet \forall\, s' \bullet \mathsf{R}_c\,(\backslash s, s') \Rightarrow \backslash a \Rightarrow p'$.

Note that $\backslash a \Rightarrow p'$ gives anteconditions the flavor of sufficient conditions, or alternatively, reflects goal-directed thinking about programs. More will be said regarding this later.

For *total correctness*, we require termination, expressed in an intuitively clear way by Term in Definition 5.2, and use this in Definition 5.3 for Hoare triples.

Definition 5.2. $\mathsf{Term}_c\, a \ \equiv \ \forall\, \backslash s \bullet \backslash a \Rightarrow \mathsf{T}_c\, \backslash s$.

Definition 5.3. $[a]\ c\ [p] \ \equiv \ \{a\}\ c\ \{p\} \wedge \mathsf{Term}_c\, a$.

5.2.2 *Defining Weakest Ante- and Strongest Postconditions.* We define $\preceq : B^2 \to \mathbb{B}$ with $q \preceq p \ \equiv \ \forall s \bullet q \Rightarrow p$. This is read "$q$ is *at least as strong as* p."

[4]Some readers may cringe at this, but if it is done properly, as is the case here, it is harmless.

Informally, a *weakest antecondition* for a given command c and postcondition p is an antecondition wa that is at least as weak as any antecondition. Formally, writing $\langle\ \rangle$ generically for either $[\ ]$ or $\{\}$, $\langle wa\rangle\ c\ \langle p\rangle \wedge \forall\, a : B\,.\,\langle a\rangle\ c\ \langle p\rangle \Rightarrow a \preceq wa$. We make metalevel quantification over B explicit so that, by predicate calculus, we can prove

(a) uniqueness, namely, if wa and wa' are weakest anteconditions, then $wa \equiv wa'$;

(b) the equivalent form $\forall\, a : B\,.\,\langle a\rangle\ c\ \langle p\rangle \equiv a \preceq wa$ (both left as exercises).

Similarly, the *strongest postcondition* for given antecondition a and command c is a proposition sp satisfying $\forall\, p : B\,.\,\langle a\rangle\ c\ \langle p\rangle \equiv sp \preceq p$.

Each has a *liberal* variant, where $\langle\ \rangle$ is $\{\}$, and a *strict* variant, where $\langle\ \rangle$ is $[\ ]$.

Summarizing in terms of relations issp_c and issp_c, both of type $B^2 \to \mathbb{B}$:

$$wa\ \ \mathsf{iswa}_c\ p \ \equiv\ \forall\, a : B\,.\,\langle a\rangle\ c\ \langle p\rangle \equiv a \preceq wa\ ; \tag{9}$$

$$sp\ \ \mathsf{issp}_c\ a \ \equiv\ \forall\, p : B\,.\,\langle a\rangle\ c\ \langle p\rangle \equiv sp \preceq p. \tag{10}$$

6. CALCULATING THEOREMS IN A PROGRAMMING THEORY

Here, we show how algebraic properties of a programming theory are calculationally derived from the theory's formulation by program equations. The examples illustrate how the calculations indeed directly lead to the *discovery* of theorems without knowing them in advance. We emphasize this by stating the theorems after calculating them.

The discovered algebraic properties constitute the basic rules for using the theory in the derivation or verification of programs in actual practice. They can be cast in the same form and thereby offer the same level of abstraction as their common formulation, but calculating them as theorems from a more elementary basis yields a vast improvement in understanding.

In an article we can give the complete detailed calculations for one theory only. For the reasons stated earlier, we chose weakest ante- and strongest postconditions. The calculations start with language-independent properties, and then turn to individual constructs of Dijkstra's guarded command language.

6.1 Calculating a General Theory for *ap*-Conditions

6.1.1 *Calculating Weakest Antecondition Operators.* To match the shape of Equation (9), namely, $\langle a\rangle\ c\ \langle p\rangle \equiv \forall\, s\,.\,a \Rightarrow wa$, we must make a the antecedent in Definition 5.1.

$$\begin{aligned}
\{a\}\ c\ \{p\} \ &\equiv \langle \text{Def. 5.1}\rangle && \forall\, \text{\textbackslash}s\,.\,\forall\, s'\,.\,\mathsf{R}_c\,(\text{\textbackslash}s, s') \Rightarrow \text{\textbackslash}a \Rightarrow p' \\
&\equiv \langle \text{Shunting } \Rightarrow\rangle && \forall\, \text{\textbackslash}s\,.\,\forall\, s'\,.\,\text{\textbackslash}a \Rightarrow \mathsf{R}_c\,(\text{\textbackslash}s, s') \Rightarrow p' \\
&\equiv \langle \text{Ldistr. } \Rightarrow/\forall\rangle && \forall\, \text{\textbackslash}s\,.\,\text{\textbackslash}a \Rightarrow \forall\, s'\,.\,\mathsf{R}_c\,(\text{\textbackslash}s, s') \Rightarrow p' \\
&\equiv \langle \text{Change var.}\rangle && \forall\, s\,.\,a \Rightarrow \forall\, s'\,.\,\mathsf{R}_c\,(s, s') \Rightarrow p'
\end{aligned}$$

The consequent $\forall\, s'\,.\,\mathsf{R}_c\,(s, s') \Rightarrow p'$ suggests defining $\mathsf{wla} : C \to B \to B$ by

Definition 6.1. $\quad \mathsf{wla}\,c\ p \ \equiv\ \forall\, s'\,.\,\mathsf{R}_c\,(s, s') \Rightarrow p'$.

So, by the calculation, $\mathsf{wla}\,c\ p$ is the weakest liberal antecondition for p, recorded as

THEOREM 6.2. $\{a\}\ c\ \{p\} \ \equiv\ a \preceq \mathsf{wla}\, c\ p.$

For the strict variant we follow the same strategy, also reusing previous results.

$$\begin{array}{lll}
[a]\ c\ [p] & \equiv \langle \text{Dif. 5.3} \rangle & \{a\}\ c\ \{p\} \wedge \mathsf{Term}_c\, a \\
& \equiv \langle \text{Thm. 6.2, Def. 5.2} \rangle & \forall\, (s \centerdot a \Rightarrow \mathsf{wla}\, cp) \wedge \forall\, (\backslash s \centerdot \backslash a \Rightarrow \mathsf{T}_c\, \backslash s) \\
& \equiv \langle \text{Change var.} \rangle & \forall\, (s \centerdot a \Rightarrow \mathsf{wla}\, cp) \wedge \forall\, (s \centerdot a \Rightarrow \mathsf{T}_c\, s) \\
& \equiv \langle \text{Distrib. } \forall/\wedge \rangle & \forall\, s \centerdot (a \Rightarrow \mathsf{wla}\, cp) \wedge (a \Rightarrow \mathsf{T}_c\, s) \\
& \equiv \langle \text{Ldistr. } \Rightarrow/\wedge \rangle & \forall\, s \centerdot a \Rightarrow \mathsf{wla}\, cp \wedge \mathsf{T}_c\, s
\end{array}$$

The consequent $\mathsf{wla}\, cp \wedge \mathsf{T}_c\, s$ suggests defining $\mathsf{wa} : C \to B \to B$ by

Definition 6.3. $\mathsf{wa}\, cp \ \equiv\ \mathsf{wla}\, cp \wedge \mathsf{T}_c\, s.$

So, by the calculation, $\mathsf{wa}\, cp$ is the weakest antecondition for p, recorded as

THEOREM 6.4. $[a]\ c\ [p] \ \equiv\ a \preceq \mathsf{wa}\, cp.$

6.1.2 *Calculating Strongest Postcondition Operators.* To match the shape of Equation (10), namely, $\langle a \rangle\ c\ \langle p \rangle \ \equiv\ \forall\, s \centerdot sp \Rightarrow p$, we must make p the consequent in Definition 5.1.

$$\begin{array}{lll}
\{a\}\ c\ \{p\} & \equiv \langle \text{Def. 5.1} \rangle & \forall\, \backslash s \centerdot \forall\, s' \centerdot \mathsf{R}_c\, (\backslash s, s') \Rightarrow \backslash a \Rightarrow p' \\
& \equiv \langle \text{Shunting } \wedge \rangle & \forall\, \backslash s \centerdot \forall\, s' \centerdot \mathsf{R}_c\, (\backslash s, s') \wedge \backslash a \Rightarrow p' \\
& \equiv \langle \text{Swapping } \forall \rangle & \forall\, s' \centerdot \forall\, \backslash s \centerdot \mathsf{R}_c\, (\backslash s, s') \wedge \backslash a \Rightarrow p' \\
& \equiv \langle \text{Rdist. } \Rightarrow/\exists \rangle & \forall\, s' \centerdot \exists\, (\backslash s \centerdot \mathsf{R}_c\, (\backslash s, s') \wedge \backslash a) \Rightarrow p' \\
& \equiv \langle \text{Change var.} \rangle & \forall\, s \centerdot \exists\, (\backslash s \centerdot (\mathsf{R}_c\, (\backslash s, s) \wedge \backslash a) \Rightarrow p
\end{array}$$

The antecedent $\exists\, \backslash s \centerdot \mathsf{R}_c\, (\backslash s, s) \wedge \backslash a$ suggests defining $\mathsf{slp} : C \to B \to B$ by

Definition 6.5. $\mathsf{slp}\, c\, a \ \equiv\ \exists\, \backslash s \centerdot \mathsf{R}_c\, (\backslash s, s) \wedge \backslash a \quad ,$

So, by the calculation, $\mathsf{slp}\, c\, a$ is the strongest liberal postcondition for a, recorded as

THEOREM 6.6. $\{a\}\ c\ \{p\} \ \equiv\ \mathsf{slp}\, c\, a \preceq p.$

For the strict variant, we try the same strategy, but the shape of the formulas does not allow going beyond $[a]\ c\ [p] \ \equiv\ \mathsf{Term}_c\, a \wedge \forall\, s \centerdot \mathsf{slp}\, c\, a \Rightarrow p$, justifying only

Definition 6.7. sp is defined conditionally by $\mathsf{Term}_c\, a \Rightarrow (\mathsf{sp}\, c\, a \ \equiv\ \mathsf{slp}\, c\, a)$.

6.1.3 *Algebraic Properties.* We just provide a few samples from a rich theory.

(1) Theorems 6.2 and 6.6 yield, by transitivity of $\equiv$,

THEOREM 6.8. $\mathsf{slp}\, c\, a \preceq p \ \equiv\ a \preceq \mathsf{wla}\, c\ p.$

If relations $\preceq : X^2 \to \mathbb{B}$ and $\sqsubseteq : Y^2 \to \mathbb{B}$ and functions $F : X \to Y$ and $G : Y \to X$ satisfy $F\, x \sqsubseteq y \ \equiv\ x \preceq G\, y$, this property is called a *Galois connection*; F is the lower adjoint and G is the upper adjoint [Backhouse 2002]. Theorem 6.8 shows that $\mathsf{slp}\, c$ is the lower and $\mathsf{wla}\, c$ the upper adjoint in a Galois connection. Theorem 6.4 has no counterpart for $\mathsf{sp}\, c$, hence $\mathsf{wa}\, c$ has no lower adjoint.

(2) Definition 6.1 implies $\mathsf{wla}\,c\,1 \equiv 1$ and, using Definition 6.3, $\mathsf{wa}\,c\,1 \equiv \mathsf{T}_c\,s$ and $\mathsf{wa}\,c\,p \equiv \mathsf{wa}\,c\,1 \wedge \mathsf{wla}\,c\,p$. This cleanly links our predicate T_c to the theory in Dijkstra and Scholten [1990], yielding $\mathsf{wa}\,c\,p \equiv \mathsf{wa}\,c\,1 \wedge \mathsf{wla}\,c\,p$ in another way.

(3) Definition 6.3 shows that requiring $\mathsf{wa}\,c\,0 \equiv 0$, Dijkstra's law of the excluded miracle (LEM) amounts to $\mathsf{T}_c\,\backslash s \Rightarrow \exists\, s' \bullet \mathsf{R}_c\,(\backslash s, s')$, as announced earlier. It has been noted, for example, in Nelson [1989], that requiring LEM may be overly restrictive.

(4) Defining the *conjugate* f^* of a proposition transformer f by $f^*(\neg\, p) \equiv \neg\,(f\ p)$, clearly $(\mathsf{wla}\,c)^*\,p \equiv \exists\, s' \bullet \mathsf{R}_c\,(s, s') \wedge p'$, and, assuming LEM, $\mathsf{wa}\,c\,p \Rightarrow (\mathsf{wla}\,c)^*\,p$.

Conversely, c is called *deterministic* iff $\forall\, p : B \,.\, \forall\, s : \mathbf{S} \,.\, (\mathsf{wla}\,c)^*\,p \Rightarrow \mathsf{wa}\,c\,p$ [Dijkstra and Scholten 1990]. This naming is justified by proving that, if c is deterministic and satisfies LEM, then $\exists\, f : \mathbf{S}_{\mathsf{T}_c} \to \mathbf{S} \,.\, \forall\, \backslash s : \mathbf{S}_{\mathsf{T}_c} \,.\, \forall\, s' : \mathbf{S} \,.\, \mathsf{R}_c\,(\backslash s, s') \equiv s' = f\,\backslash s$. One way to show this is by taking $p := [\![s = t]\!]$ and invoking function comprehension [Boute 2002, 2006].

(5) Some other algebraic properties, derived by easy calculation from the operator definitions, are the following; w(l)a is a generic name for both wla and wa.

$$\mathsf{w(l)a}\ \ \mathtt{skip}\ p \ \equiv\ p \tag{11}$$

$$p \preceq q \ \Rightarrow\ \mathsf{w(l)a}\,c\,p \preceq \mathsf{w(l)a}\,c\,q \tag{12}$$

$$\mathsf{w(l)a}\,c\,(p \vee q) \ \Leftarrow\ \mathsf{w(l)a}\,c\,p \vee \mathsf{w(l)a}\,c\,q \tag{13}$$

$$\mathsf{w(l)a}\,c\,(p \wedge q) \ \equiv\ \mathsf{w(l)a}\,c\,p \wedge \mathsf{w(l)a}\,c\,q \tag{14}$$

$$\mathsf{wa}\,c\,(p \wedge q) \ \equiv\ \mathsf{wla}\,c\,p \wedge \mathsf{wa}\,c\,q \quad \text{(“Borrowing”)} \tag{15}$$

If c is deterministic and LEM holds, then $\mathsf{w(l)a}\,c\,(p \vee q) \equiv \mathsf{w(l)a}\,c\,p \vee \mathsf{w(l)a}\,c\,q$.

Similarly, for the strongest postcondition (considering slp only),

$$\mathsf{slp}\ \ \mathtt{skip}\,a \ \equiv\ a \tag{16}$$

$$a \preceq b \ \Rightarrow\ \mathsf{slp}\,c\,a \preceq \mathsf{slp}\,c\,b \tag{17}$$

$$\mathsf{slp}\,c\,(a \wedge b) \ \Rightarrow\ \mathsf{slp}\,c\,a \wedge \mathsf{slp}\,c\,b \tag{18}$$

$$\mathsf{slp}\,c\,(a \vee b) \ \equiv\ \mathsf{slp}\,c\,a \vee \mathsf{slp}\,c\,b \tag{19}$$

6.1.4 *On Dummies and Shorthands.* The shape of the formulas shows that Hoare-triples are most elegantly expressed by the symmetric dummies $\backslash s, s'$, but that variable changes are reduced by choosing s, s' for w(l)a-related formulas and $\backslash s, s$ for s(l)p-related ones. We shall often use the latter choices in the sequel. For w(l)a-related formulas, we also introduce $\mathsf{r} : C \to B$ and $\mathsf{t} : C \to B$ to obtain shorthands via

Definition 6.9. $\quad \mathsf{r}\,c \equiv \mathsf{R}_c\,(s, s')$ and $\mathsf{t}\,c \equiv \mathsf{T}_c\,s$.

The burden of these extra operators is rewarded by convenience and style issues (elaborated on later), and is hopefully mitigated by having chosen matching letters.

6.2 Application to Assignment (Embedding in the General Case)

Assignment can be multiple, that is, v in $v := e$ can be a tuple of variables and e a matching tuple of expressions. As one of the reviewers observed, a parallel operator improves readability, as it allows writing $x, y := d, e$ as $x := d \parallel y := e$.

Table V yields $\mathsf{R}_{v:=e}\,(\backslash s, s') \equiv s' = \backslash s[^{\backslash v}_{e}$, from which wla and slp are calculated using Definitions 6.1 and 6.5 by the method shown in Section 3, with similar results:

$$\mathsf{wla}\,[\![v{:=}e]\!]\,p \;\equiv\; p[^{v}_{e} \;\text{ and }\; \mathsf{slp}\,[\![v{:=}e]\!]\,a \;\equiv\; \exists\backslash v : \mathbf{S}_v\,.\,a[^{v}_{\backslash v} \wedge v = e[^{v}_{\backslash v}. \qquad (20)$$

Since $\mathsf{T}_{v:=e}\,s \;\equiv\; 1$, these are also strict. Also, $\mathsf{wa}\,[\![\texttt{skip}]\!]\,p \;\equiv\; p$ and $\mathsf{sp}\,[\![\texttt{skip}]\!]\;p \;\equiv\; p$.

6.3 Calculations for Composition

The relevant program equations from Table V are recalled for easy reference as

Definition 6.10. (a) $\mathsf{R}_{c';c''}\,(\backslash s, s') \equiv \exists s \,.\, \mathsf{R}_{c'}\,(\backslash s, s) \wedge \mathsf{R}_{c''}\,(s, s')$;
(b) $\mathsf{T}_{c';c''}\,\backslash s \;\equiv\; \mathsf{T}_{c'}\,\backslash s \wedge \forall s \,.\, \mathsf{R}_{c'}\,(\backslash s, s) \Rightarrow \mathsf{T}_{c''}\,s.$

We tacitly change $\backslash s, s'$ to s, s' in antecalculations and to $\backslash s, s$ in postcalculations. A remaining dummy can be reused as an auxiliary dummy, for example, to write Definition (6.10.a) as $\mathsf{R}_{c';c''}\,(s, s') \equiv \exists\backslash s \,.\, \mathsf{R}_{c'}\,(s, \backslash s) \wedge \mathsf{R}_{c''}\,(\backslash s, s')$ or $\mathsf{R}_{c';c''}\,(\backslash s, s) \equiv \exists s' \,.\, \mathsf{R}_{c'}\,(\backslash s, s') \wedge \mathsf{R}_{c''}\,(s', s)$.

6.3.1 *Weakest Antecondition.* We start by calculating $\mathsf{wla}\,[\![c'; c'']\!]\,p$.

$$\begin{array}{lll}
\mathsf{wla}\,[\![c'; c'']\!]\,p & \equiv \langle \text{Def. wla (6.1)}\rangle & \forall s' \,.\, \mathsf{R}_{c';c''}\,(s, s') \Rightarrow p' \\
& \equiv \langle \text{Def. R (6.10.a)}\rangle & \forall s' \,.\, \exists\,(\backslash s \,.\, \mathsf{R}_{c'}\,(s, \backslash s) \wedge \mathsf{R}_{c''}\,(\backslash s, s')) \Rightarrow p' \\
& \equiv \langle \text{R-distrib. } \Rightarrow/\exists\rangle & \forall s' \,.\, \forall\backslash s \,.\, \mathsf{R}_{c'}\,(s, \backslash s) \wedge \mathsf{R}_{c''}\,(\backslash s, s') \Rightarrow p' \\
& \equiv \langle \text{Swap } \forall, \text{ shunt } \wedge\rangle & \forall\backslash s \,.\, \forall s' \,.\, \mathsf{R}_{c'}\,(s, \backslash s) \Rightarrow \mathsf{R}_{c''}\,(\backslash s, s') \Rightarrow p' \\
& \equiv \langle \text{L-distrib. } \Rightarrow/\forall\rangle & \forall\backslash s \,.\, \mathsf{R}_{c'}\,(s, \backslash s) \Rightarrow \forall s' \,.\, \mathsf{R}_{c''}\,(\backslash s, s') \Rightarrow p' \\
& \equiv \langle \text{Def. wla (6.1)}\rangle & \forall\backslash s \,.\, \mathsf{R}_{c'}\,(s, \backslash s) \Rightarrow (\mathsf{wla}\,c''\;p)[^{s}_{\backslash s} \\
& \equiv \langle \text{Def. wla (6.1)}\rangle & \mathsf{wla}\,c'\,(\mathsf{wla}\,c''\;p)
\end{array}$$

Hence we have derived the following.

THEOREM 6.11. $\mathsf{wla}\,[\![c'; c'']\!]\,p \;\equiv\; \mathsf{wla}\,c'\,(\mathsf{wla}\,c''\;p)$.

Next, we calculate $\mathsf{wa}\,[\![c'; c'']\!]\,p$, recalling that $\mathsf{t}\,c$ is shorthand for $\mathsf{T}_c\,s$.

$$\begin{array}{lll}
\mathsf{wa}\,[\![c'; c'']\!]\,p & \equiv \langle \text{Def. wa (6.3)}\rangle & \mathsf{wla}\,[\![c'; c'']\!]\,p \wedge \mathsf{t}\,[\![c'; c'']\!] \\
& \equiv \langle \text{Thm. 6.11}\rangle & \mathsf{wla}\,c'\,(\mathsf{wla}\,c''\;p) \wedge \mathsf{t}\,[\![c'; c'']\!] \\
& \equiv \langle \text{Lemma 6.13}\rangle & \mathsf{wla}\,c'\,(\mathsf{wla}\,c''\;p) \wedge \mathsf{wa}\,c'\,(\mathsf{t}\,c'') \\
& \equiv \langle \text{Borrow. (15)}\rangle & \mathsf{wa}\,c'\,(\mathsf{wla}\,c''\;p \wedge \mathsf{t}\,c'') \\
& \equiv \langle \text{Def. wa (6.3)}\rangle & \mathsf{wa}\,c'\,(\mathsf{wa}\,c''\;p)
\end{array}$$

Hence we have derived

THEOREM 6.12. $\mathsf{wa}\,[\![c'; c'']\!]\,p \;\equiv\; \mathsf{wa}\,c'\,(\mathsf{wa}\,c''\;p)$.

The invoked lemma (stated next) amounts to recognizing the shape of Definitions 6.1 (for wla) and 6.3 (for wa) in the righthand side of Definition 6.10.b.

LEMMA 6.13. $\mathsf{t}[\![c'; c'']\!] \equiv \mathsf{wa}\, c'\, (\mathsf{t}\, c'')$.

6.3.2 *On Theorems and Lemmata.* In most mathematics texts, theorems are presented as finished products and lemmata are given in advance. We typically *calculate* theorems, not even assuming the "result" to be known, and record proof obligations arising along the way as lemmata, proved afterwards for theorems worth keeping.

6.3.3 *Strongest Postcondition.* We calculate $\mathsf{slp}\,[\![c'; c'']\!]\, a$.

$$
\begin{array}{lll}
\mathsf{slp}\,[\![c'; c'']\!]\, a & \equiv \langle \text{Def. slp (6.5)} \rangle & \exists\, \backslash s \bullet \backslash a \wedge \mathsf{R}_{c';c''}\, (\backslash s, s) \\
& \equiv \langle \text{Def. R (6.10.a)} \rangle & \exists\, \backslash s \bullet \backslash a \wedge \exists\, s' \bullet \mathsf{R}_{c'}\, (\backslash s, s') \wedge \mathsf{R}_{c''}\, (s', s) \\
& \equiv \langle \text{Distr. } \wedge/\exists \rangle & \exists\, \backslash s \bullet \exists\, s' \bullet \backslash a \wedge \mathsf{R}_{c'}\, (\backslash s, s') \wedge \mathsf{R}_{c''}\, (s', s) \\
& \equiv \langle \text{Swapping } \exists \rangle & \exists\, s' \bullet \exists\, \backslash s \bullet \backslash a \wedge \mathsf{R}_{c'}\, (\backslash s, s') \wedge \mathsf{R}_{c''}\, (s', s) \\
& \equiv \langle \text{Distr. } \wedge/\exists \rangle & \exists\, s' \bullet \exists\, (\backslash s \bullet \backslash a \wedge \mathsf{R}_{c'}\, (\backslash s, s')) \wedge \mathsf{R}_{c''}\, (t, s') \\
& \equiv \langle \text{Def. slp (6.5)} \rangle & \exists\, s' \bullet (\mathsf{slp}\, c'\, a)[^{s}_{s'} \wedge \mathsf{R}_{c''}\, (s', s) \\
& \equiv \langle \text{Def. slp (6.5)} \rangle & \mathsf{slp}\, c''\, (\mathsf{slp}\, c'\, a)
\end{array}
$$

Hence, we have derived the following

THEOREM 6.14. $\mathsf{slp}\,[\![c'; c'']\!]\, a \equiv \mathsf{slp}\, c''\, (\mathsf{slp}\, c'\, a)$.

Deriving the strict variant $\mathsf{Term}_{c';c''}\, a \Rightarrow (\mathsf{sp}\,[\![c'; c'']\!]\, a \equiv \mathsf{sp}\, c''\, (\mathsf{sp}\, c'\, a))$ reuses the last two steps from the calculation by invoking $\mathsf{Term}_{c'}\, a \Rightarrow (\mathsf{sp}\, c'\, a \equiv \mathsf{slp}\, c'\, a)$ and $\mathsf{Term}_{c''}\, (\mathsf{sp}\, c'\, a) \Rightarrow (\mathsf{sp}\, c''\, (\mathsf{sp}\, c'\, a) \equiv \mathsf{slp}\, c''\, (\mathsf{sp}\, c'\, a))$. Checking the antecedents:

$$
\begin{array}{lll}
\mathsf{Term}_{c';c''}\, a & \equiv \langle \text{Def. Term (5.2)} \rangle & a \preceq \mathsf{t}[\![c'; c'']\!] \\
& \equiv \langle \text{Lem. 6.13; def. 6.3} \rangle & a \preceq (\mathsf{t}\, c' \wedge \mathsf{wla}\, c'\, (\mathsf{t}\, c'')) \\
& \equiv \langle \text{L-distribut. } \preceq/\wedge \rangle & (a \preceq \mathsf{t}\, c') \wedge (a \preceq \mathsf{wla}\, c'\, (\mathsf{t}\, c'')) \\
& \equiv \langle \text{Galois conn. (6.8)} \rangle & (a \preceq \mathsf{t}\, c') \wedge (\mathsf{slp}\, c'\, a \preceq \mathsf{t}\, c'') \\
& \equiv \langle \text{Def. Term (5.2)} \rangle & \mathsf{Term}_{c'}\, a \wedge (\mathsf{slp}\, c'\, a \preceq \mathsf{t}\, c'') \\
& \equiv \langle \text{Def. sp (6.7)} \rangle & \mathsf{Term}_{c'}\, a \wedge (\mathsf{sp}\, c'\, a \preceq \mathsf{t}\, c'') \\
& \equiv \langle \text{Def. Term (5.2)} \rangle & \mathsf{Term}_{c'}\, a \wedge \mathsf{Term}_{c''}\, (\mathsf{sp}\, c'\, a).
\end{array}
$$

6.4 Calculations for Choice

Let $c := [\![\texttt{if}\ \ [\!]\ i : I \,.\, b_i \ \texttt{->}\ c'_i \ \texttt{fi}]\!]$ throughout this subsection. Table V yields

Definition 6.15. (a) $\mathsf{R}_c\, (\backslash s, s') \equiv \exists\, i : I \,.\, \backslash b_i \wedge \mathsf{R}_{c'_i}\, (\backslash s, s')$;
(b) $\mathsf{T}_c\, \backslash s \equiv \exists\, (i : I \,.\, \backslash b_i) \wedge \forall\, i : I \,.\, \backslash b_i \Rightarrow \mathsf{T}_{c'_i}\, \backslash s$.

6.4.1 *Weakest Antecondition.* As usual, we simply calculate. For $\mathsf{wla}\, c\, p$,

$$
\begin{array}{lll}
\mathsf{wla}\, c\, p & \equiv \langle \text{Def. wla (6.1)} \rangle & \forall\, s' \bullet \mathsf{R}_c\, (s, s') \Rightarrow p' \\
& \equiv \langle \text{Def. R (6.15.a)} \rangle & \forall\, s' \bullet \exists\, (i : I \,.\, b_i \wedge \mathsf{R}_{c'_i}\, (s, s')) \Rightarrow p' \\
& \equiv \langle \text{Rdist. } \Rightarrow/\exists \rangle & \forall\, s' \bullet \forall\, i : I \,.\, b_i \wedge \mathsf{R}_{c'_i}\, (s, s') \Rightarrow p' \\
& \equiv \langle \text{Shunting } \wedge \rangle & \forall\, s' \bullet \forall\, i : I \,.\, b_i \Rightarrow \mathsf{R}_{c'_i}\, (s, s') \Rightarrow p' \\
& \equiv \langle \text{Swapping } \forall \rangle & \forall\, i : I \,.\, \forall\, s' \bullet b_i \Rightarrow \mathsf{R}_{c'_i}\, (s, s') \Rightarrow p' \\
& \equiv \langle \text{Ldistr. } \Rightarrow/\forall \rangle & \forall\, i : I \,.\, b_i \Rightarrow \forall\, s' \bullet \mathsf{R}_{c'_i}\, (s, s') \Rightarrow p' \\
& \equiv \langle \text{Def. wla (6.1)} \rangle & \forall\, i : I \,.\, b_i \Rightarrow \mathsf{wa}\, c'_i\, p.
\end{array}
$$

This calculation yields

THEOREM 6.16. $\mathsf{wla}\,[\![\texttt{if } [\!]\, i:I\,.\,b_i \texttt{ -> } c_i'\texttt{fi}]\!]\,p \;\equiv\; \forall i:I\,.\,b_i \Rightarrow \mathsf{wla}\,c_i'\,p.$

The calculations for $\mathsf{wa}\,c\,p$ are equally straightforward:

$$\begin{array}{lll}
\mathsf{wa}\,c\,p & \equiv \langle \text{Def. wa (6.3)}\rangle & \mathsf{t}\,c \wedge \mathsf{wla}\,c\,p \\
& \equiv \langle \text{Def. 6.15.b, Thm. 6.16}\rangle & \exists b \wedge \forall (i:I\,.\,b_i \Rightarrow \mathsf{t}\,c_i') \wedge \forall i:I\,.\,b_i \Rightarrow \mathsf{wla}\,c_i'\,p \\
& \equiv \langle \text{Distrib. laws}\rangle & \exists b \wedge \forall i:I\,.\,b_i \Rightarrow \mathsf{t}\,c_i' \wedge \mathsf{wla}\,c_i'\,p \\
& \equiv \langle \text{Def. wa (6.3)}\rangle & \exists b \wedge \forall i:I\,.\,b_i \Rightarrow \mathsf{wa}\,c_i'\,p
\end{array}$$

Clarification: $b = i:I\,.\,b_i$, so $\exists\, b \;\equiv\; \exists i:I\,.\,b_i$. The preceding calculation yields

THEOREM 6.17. $\mathsf{wa}\,[\![\texttt{if } [\!]\, i:I\,.\,b_i \texttt{ -> } c_i'\texttt{fi}]\!]\,p \;\equiv\; \exists b \wedge \forall i:I\,.\,b_i \Rightarrow \mathsf{wa}\,c_i'\,p.$

COROLLARY 6.18. $\mathsf{wa}\,[\![\texttt{if } b \texttt{ then } c' \texttt{ else } c'' \texttt{ fi}]\!]\,p \;\equiv\; (b \Rightarrow \mathsf{wa}\,c'\,p) \wedge (\neg b \Rightarrow \mathsf{wa}\,c''\,p)$.

Observing that $(q \Rightarrow r) \wedge (\neg q \Rightarrow r') \;\equiv\; (q \wedge r) \vee (\neg q \wedge r')$ yields an equivalent form.

6.4.2 *Strongest Postcondition.* Calculating $\mathsf{slp}\,c\,a$

$$\begin{array}{lll}
\mathsf{slp}\,c\,a & \equiv \langle \text{Def. slp (6.5)}\rangle & \exists\, \backslash s\,.\,\backslash a \wedge \mathsf{R}_c\,(\backslash s, s) \\
& \equiv \langle \text{Def. R (6.15.a)}\rangle & \exists\, \backslash s\,.\,\backslash a \wedge \exists i:I\,.\,\backslash b_i \wedge \mathsf{R}_{c_i'}\,(\backslash s, s) \\
& \equiv \langle \text{Distrib. } \wedge/\exists\rangle & \exists\, \backslash s\,.\,\exists i:I\,.\,\backslash a \wedge \backslash b_i \wedge \mathsf{R}_{c_i'}\,(\backslash s, s) \\
& \equiv \langle \text{Swapping } \exists\rangle & \exists i:I\,.\,\exists\, \backslash s\,.\,\backslash a \wedge \backslash b_i \wedge \mathsf{R}_{c_i'}\,(\backslash s, s) \\
& \equiv \langle \text{Def. slp (6.5)}\rangle & \exists i:I\,.\,\mathsf{slp}\,c_i'\,(a \wedge b_i)
\end{array}$$

yields

THEOREM 6.19. $\mathsf{slp}\,[\![\texttt{if } [\!]\, i:I\,.\,b_i \texttt{ -> } c_i'\texttt{fi}]\!]\,a \;\equiv\; \exists i:I\,.\,\mathsf{slp}\,c_i'\,(a \wedge b_i)$.

COROLLARY 6.20. $\mathsf{slp}\,[\![\texttt{if } b \texttt{ then } c' \texttt{ else } c'' \texttt{ fi}]\!]\,a \;\equiv\; \mathsf{slp}\,c'\,(a \wedge b) \vee \mathsf{slp}\,c''\,(a \wedge \neg b)$.

The antecedent for justifying the last step in a similar calculation of the strict variant is covered by $\mathsf{Term}_c\,a \;\equiv\; \forall (s\,.\,a \Rightarrow \exists\, b) \wedge \forall i:I\,.\,\mathsf{Term}_{c_i'}\,(a \wedge b_i)$ (exercise).

6.5 Calculations for Repetition

Letting $c := [\![\texttt{do } b \texttt{ -> } c' \texttt{ od}]\!]$ for this subsection, we calculate termination first.

$$\begin{array}{lll}
\mathsf{t}\,c & \equiv \langle \text{Def. } c\rangle & \mathsf{t}\,[\![\texttt{do } b \texttt{ -> } c' \texttt{ od}]\!] \\
& \equiv \langle \text{Table V, rep.}\rangle & \mathsf{t}\,[\![\texttt{if } \neg b \texttt{ -> skip } [\!]\, b \texttt{ -> } c' \texttt{ ; } c \texttt{ fi}]\!] \\
& \equiv \langle \text{Def. t (6.15.b)}\rangle & (\neg b \Rightarrow \mathsf{t}\,[\![\texttt{skip}]\!]) \wedge (b \Rightarrow \mathsf{t}\,[\![c' \texttt{ ; } c]\!]) \\
& \equiv \langle \text{Def. t skip}\rangle & (\neg b \Rightarrow 1) \wedge (b \Rightarrow \mathsf{t}\,[\![c' \texttt{ ; } c]\!]) \\
& \equiv \langle \text{Prop. calcul.}\rangle & b \Rightarrow \mathsf{t}\,[\![c' \texttt{ ; } c]\!] \\
& \equiv \langle \text{Lemma 6.13}\rangle & b \Rightarrow \mathsf{wa}\,c'\,(\mathsf{t}\,c)
\end{array}$$

The result of this calculation is recorded as

THEOREM 6.21. $\mathsf{t}\,c \;\equiv\; b \Rightarrow \mathsf{wa}\,c'\,(\mathsf{t}\,c)$.

By a similar calculation, using $\langle$Def. $c\rangle$, $\langle$Table V$\rangle$, $\langle$Def. 6.15.a$\rangle$, $\langle \mathsf{R}\ \texttt{skip}\rangle$, $\langle$Def. 6.10.a$\rangle$:

THEOREM 6.22. $\mathsf{r}\,c \;\equiv\; (\neg b \Rightarrow s = s') \wedge (b \Rightarrow \exists\, \backslash s\,.\,\mathsf{R}_{c'}\,(s, \backslash s) \wedge \mathsf{R}_c\,(\backslash s, s'))$,

and, using the theorems for choice (6.16, 6.17), skip, composition (6.11, 6.12):

THEOREM 6.23. $\mathsf{wla}\,cp \;\equiv\; (\neg b \Rightarrow p) \wedge (b \Rightarrow \mathsf{wla}\,c'\,(\mathsf{wla}\,cp))$;

THEOREM 6.24. $\mathsf{wa}\,cp \;\equiv\; (\neg b \Rightarrow p) \wedge (b \Rightarrow \mathsf{wa}\,c'\,(\mathsf{wa}\,cp))$.

6.5.1 *Calculating the Weakest Antecondition.* Solutions of recursion equations such as (6.21), (6.23), and (6.24) are not always unique. The choice depends on operational considerations that can be intricate [Dijkstra and Scholten 1990] and will be given for wa only. For this purpose, we use the termination equation and an interesting "absorption" technique using the borrowing Equation (15).

We fix the choice of solution by bounding the number of iterations. Operationally, let $g_n \equiv$ "c started in state s terminates in at most, n iterations." The easiest formalization is by recursion: $g_0 \equiv \neg b$ (obvious) and $g_{n+1} \equiv$ "either c terminates immediately (case $\neg b$) or c' executes a first time and, from there on, c terminates in, at most, n steps", that is, $g_{n+1} \equiv \neg b \vee (b \wedge \mathsf{t}c' \wedge \forall s' \bullet \mathsf{r}c' \Rightarrow g_n[^{s}_{s'}])$. Simplified,

$$g_0 \;\equiv\; \neg b \quad \text{and} \quad g_{n+1} \;\equiv\; b \Rightarrow \mathsf{wa}\,c'\,g_n. \tag{21}$$

A simple inductive proof[5] using (6.21) yields $\forall n : \mathbb{N} \,.\, g_n \Rightarrow \mathsf{t}c$, so $\exists (n : \mathbb{N} \,.\, g_n) \Rightarrow \mathsf{t}c$. By construction, $\exists n : \mathbb{N} \,.\, g_n$ operationally means that the number of iterations is bounded. We make this a requirement, strengthening $\exists (n : \mathbb{N} \,.\, g_n) \Rightarrow \mathsf{t}c$ to

$$\mathsf{t}c \;\equiv\; \exists n : \mathbb{N} \,.\, g_n. \tag{22}$$

Later on, we shall get rid of this requirement. As for now, let us calculate $\mathsf{wa}\,cp$.

$$\begin{array}{lll}
\mathsf{wa}\,cp & \equiv \langle \text{Def. wa (6.3)} \rangle & \mathsf{wla}\,cp \wedge \mathsf{t}c \\
& \equiv \langle \text{Eqn. t (22)} \rangle & \mathsf{wla}\,cp \wedge \exists n : \mathbb{N} \,.\, g_n \\
& \equiv \langle \text{Distrib. } \wedge/\exists \rangle & \exists n : \mathbb{N} \,.\, \mathsf{wla}\,cp \wedge g_n \\
& \equiv \langle \text{Intro. } h \rangle & \exists n : \mathbb{N} \,.\, h_n
\end{array}$$

introducing h with $h_n \;\equiv\; \mathsf{wla}\,cp \wedge g_n$. So, $h_0 \;\equiv\; \mathsf{wla}\,cp \wedge \neg b$ or, by (6.23), $h_0 \;\equiv\; \neg b \wedge p$.

$$\begin{array}{lll}
h_{n+1} \equiv \langle \text{Def. } h \rangle & \mathsf{wla}\,cp \wedge g_{n+1} \\
\equiv \langle \text{Thm. 6.23} \rangle & (\neg b \Rightarrow p) \wedge (b \Rightarrow \mathsf{wla}\,c'\,(\mathsf{wla}\,cp)) \wedge g_{n+1} \\
\equiv \langle \text{Def. } g \text{ (21)} \rangle & (\neg b \Rightarrow p) \wedge (b \Rightarrow \mathsf{wla}\,c'\,(\mathsf{wla}\,cp)) \wedge (b \Rightarrow \mathsf{wa}\,c'\,g_n) \\
\equiv \langle \text{Ldistr. } \Rightarrow/\wedge \rangle & (\neg b \Rightarrow p) \wedge (b \Rightarrow \mathsf{wla}\,c'\,(\mathsf{wla}\,cp) \wedge \mathsf{wa}\,c'\,g_n) \\
\equiv \langle \text{Borrow. (15)} \rangle & (\neg b \Rightarrow p) \wedge (b \Rightarrow \mathsf{wa}\,c'\,(\mathsf{wla}\,cp \wedge g_n)) \\
\equiv \langle \text{Def. } h \rangle & (\neg b \Rightarrow p) \wedge (b \Rightarrow \mathsf{wa}\,c'\,h_n) \\
\equiv \langle \text{Shannon cnv.} \rangle & (\neg b \wedge p) \vee (b \wedge \mathsf{wa}\,c'\,h_n) \\
\equiv \langle \text{Intro. w (6.25)} \rangle & \mathsf{w}\,h_n
\end{array}$$

introducing w by

Definition 6.25. $\mathsf{w}\,q \;\equiv\; (\neg b \wedge p) \vee (b \wedge \mathsf{wa}\,c'\,q)$.

Conversion from one Shannon form into the other helps to cover h_0 in the result $h_n \;\equiv\; \mathsf{w}^n\,(\neg b \wedge p)$. Considering $\mathsf{wa}\,cp \;\equiv\; \exists n : \mathbb{N} \,.\, h_n$, we have derived

[5]In this context, the proof of the inductive case typically uses isotony of $\mathsf{wa}\,c$, of $\wedge$ and $\vee$, and right isotony of $\Rightarrow$, all with respect to $\Rightarrow$. A more interesting example of induction is given later.

THEOREM 6.26. $\mathsf{wa}\,cp \;\equiv\; \exists\, n:\mathbb{N}\,.\,\mathsf{w}^n\,(\neg\, b \wedge p)$.

Note how the desired solution of the recursion equation of Theorem 6.24 for $\mathsf{wa}\,cp$ was calculated by absorbing (22) into Theorem 6.23, bypassing Theorem 6.24.

If LEM is assumed, $\mathsf{w}\,0 \;\equiv\; \neg\, b \wedge p$, hence $h_n \;\equiv\; \mathsf{w}^{n+1}0$, yielding the following theorem, which is essentially Equation (9.45) in Dijkstra and Scholten [1990].

THEOREM 6.27. $\mathsf{wa}\,cp \;\equiv\; \exists\, n:\mathbb{N}\,.\,\mathsf{w}^n\,0$.

6.5.2 *Outline for* wla. Taking Theorem 6.23, unfolding suggests introducing $r_n \;\equiv\; wl^n\,(\neg\, b \Rightarrow p)$, where $wl\,q \;\equiv\; (\neg\, b \Rightarrow p) \wedge (b \Rightarrow \mathsf{wla}\,c'\,q)$, and so on. Instead, as a convenient abbreviation [Dijkstra and Scholten 1990] we let $d := [\![\texttt{if}\;\; b \texttt{ -> } c' \texttt{fi}]\!]$ and write Theorem 6.23 as $\mathsf{wla}\,cp \;\equiv\; (b \vee p) \wedge \mathsf{wla}\,d\,(\mathsf{wla}\,cp)$.

Hence, $\mathsf{wla}\,c$ is a solution of $f\;p \;\equiv\; (b \vee p) \wedge \mathsf{wla}\,d\,(f\;p)$ with unknown f. Prompted by unfolding, we introduce $r_n \;\equiv\; (\mathsf{wla}\,d)^n\,(b \vee p)$. By induction, we prove $f\;p \Rightarrow r_n$ for any solution f. Mere substitution in the equation and predicate calculus shows that the particular f' defined by $f'\;p \;\equiv\; \forall\, n:\mathbb{N}\,.\,r_n$ is itself a solution, and since $f\;p \Rightarrow f'\;p$ for any solution f, it is the weakest. By the operational arguments in Dijkstra and Scholten [1990], f' is the desired $\mathsf{wla}\,c$. Hence, we have proved the following equivalent of Equation (9.41) in Dijkstra and Scholten [1990, p. 185].

THEOREM 6.28. $\mathsf{wla}\,cp \;\equiv\; \forall\, n:\mathbb{N}\,.\,(\mathsf{wla}\,d)^n\,(b \vee p)$.

6.5.3 *Calculating* s(l)p. Recall that sp and slp differ only by the condition with Term. The equation $\mathsf{Term}_c\,a \;\equiv\; \mathsf{Term}'_c\,(b \wedge a) \wedge \mathsf{Term}_c\,(\mathsf{sp}\,c'\,(b \wedge a))$ is easily derived. For slp, one could use unfolding as for wla to obtain $\mathsf{slp}\,c\,a \;\equiv\; \neg\, b \wedge \exists\, n:\mathbb{N}\,.\,(\mathsf{slp}\,d)^n\,a$, which is Equation (9.13) in Dijkstra and Scholten [1990, p. 213], noting that our slp is named sp in Dijkstra and Scholten [1990]. Yet, for the sake of diversity, and to obviate separate operational arguments, we proceed differently.

6.5.4 *Using Galois Connections.* We recall $\mathsf{slp}\,c\,a \preceq p \;\equiv\; a \preceq \mathsf{wla}\,cp$ from Theorem 6.8, which uniquely defines for given $\mathsf{wla}\,c$ the matching $\mathsf{slp}\,c$, and vice versa.

For $c := [\![\texttt{do}\;\; b \texttt{ -> } c' \;\texttt{od}]\!]$ and $\mathsf{wla}\,c$ given by Theorem 6.28, we derive a proposition transformer f that satisfies $f\,a \preceq p \;\equiv\; a \preceq \mathsf{wla}\,cp$ and hence, must be $\mathsf{slp}\,c$. With shorthands $h := \mathsf{slp}\,d$ and $k := \mathsf{wla}\,d$, we calculate for arbitrary a and p,

$$\begin{array}{lll}
a \preceq \mathsf{wla}\,cp & \equiv \langle \text{Thm. 6.28, def. } k\rangle & a \preceq \forall\, n:\mathbb{N}\,.\,k^n\,(b \vee p) \\
& \equiv \langle \text{Remark to follow}\rangle & \forall\, n:\mathbb{N}\,.\,a \preceq k^n\,(b \vee p) \\
& \equiv \langle \text{Lem. 6.30 to follow}\rangle & \forall\, n:\mathbb{N}\,.\,(\neg\, b \wedge h^n\,a) \preceq p \\
& \equiv \langle \text{Remark to follow}\rangle & \exists\,(n:\mathbb{N}\,.\,\neg\, b \wedge h^n\,a) \preceq p \\
& \equiv \langle \text{Distribut. } \wedge/\exists\rangle & \neg\, b \wedge \exists\,(n:\mathbb{N}\,.\,h^n\,a) \preceq p
\end{array}$$

Remark: the justifications are properties of $\preceq$ that were calculated "in line" in our first version of this proof, but are now factored out as $\forall\,(i:I\,.\,p \preceq q_i) \;\equiv\; p \preceq \forall\, i:I\,.\,q_i$ and $\forall\,(i:I\,.\,q_i \preceq p) \;\equiv\; \exists\,(i:I\,.\,q_i) \preceq p$ for $p:B$ and $q:I \to B$

(easy exercise). So, f defined by $f\,a \equiv \neg b \wedge \exists (n:\mathbb{N}\,.\,h^n\,a)$ satisfies $f\,a \preceq p \equiv a \preceq \mathsf{wla}\,cp$, yielding

THEOREM 6.29. $\mathsf{slp}\,ca \equiv \neg b \wedge \exists n:\mathbb{N}\,.\,(\mathsf{slp}\,d)^n\,a.$

The lemma invoked (originally conjectured in the spirit of Section 6.3.2) is

LEMMA 6.30. $a \preceq k^n\,(b \vee p) \equiv (\neg b \wedge h^n\,a) \preceq p.$

This "rabbit" of *conjecturing* 6.30 can be made digestible by observing that (i) the r.h.s. of $\preceq$ must be p; (ii) for $n = 0$ it is obvious what to do; and (iii) $\neg b$ is clearly part of any postcondition, which is why we opt for $\neg b \wedge h^n\,a$ rather than $h^n\,(\neg b \wedge a)$.

We still must *prove* 6.30, given $h\,a \preceq p \equiv a \preceq k\,p$ (Theorem 6.8). As shorthands, we define families f and g of proposition transformers with $g_n\,p \equiv k^n\,(b \vee p)$ and $f_n\,a \equiv \neg b \wedge h^n\,a$. For the proof, we introduce the induction predicate $P:\mathbb{N}\to\mathbb{B}$ with $P\,n \equiv \forall (a,p):B^2\,.\,a \preceq g_n\,p \equiv f_n\,a \preceq p$ and use the *weak induction* principle [Gries and Schneider 1993]: $\forall P \equiv P\,0 \wedge \forall n:\mathbb{N}\,.\,P\,n \Rightarrow P\,(n+1)$. Base case:

$$\begin{array}{lll}
P\,0 & \equiv & \langle \text{Def. of } P\rangle \quad \forall (a,p):B^2\,.\,\forall (s\,.\,a \Rightarrow g_0\,p) \equiv \forall (s\,.\,f_0\,a \Rightarrow p)\\
& \equiv & \langle \text{Def. } f,\,g\rangle \quad \forall (a,p):B^2\,.\,\forall (s\,.\,a \Rightarrow k^0\,(b \vee p)) \equiv \forall (s\,.\,\neg b \wedge h^0\,a \Rightarrow p)\\
& \equiv & \langle \mathit{function}^0\,x = x\rangle\ \forall (a,p):B^2\,.\,\forall (s\,.\,a \Rightarrow b \vee p) \equiv \forall (s\,.\,\neg b \wedge a \Rightarrow p)\\
& \equiv & \langle \text{Propos. calcul.}\rangle\ \forall (a,p):B^2\,.\,\forall (s\,.\,\neg b \Rightarrow a \Rightarrow p) \equiv \forall (s\,.\,\neg b \Rightarrow a \Rightarrow p)\\
& \equiv & \langle \text{Leibniz, refl. } \equiv\rangle\ 1
\end{array}$$

Inductive case: given n, assume $P\,n$ and prove $P\,(n+1)$ by calculating

$$\begin{array}{lll}
a \preceq g_{n+1}\,p & \equiv \langle \text{Def. } g\rangle & a \preceq k^{n+1}\,(b \vee p)\\
& \equiv \langle \text{Def. } \mathit{func}^n\rangle & a \preceq k\,(k^n\,(b \vee p))\\
& \equiv \langle \text{Galois } h,k\rangle & h\,a \preceq k^n\,(b \vee p)\\
& \equiv \langle \text{Def. } g\rangle & h\,a \preceq g_n\,p\\
& \equiv \langle \text{Hypoth. } P\,n\rangle & f_n\,(h\,a) \preceq p\\
& \equiv \langle \text{Def. } f\rangle & (\neg b \wedge h^n\,(h\,a)) \preceq p\\
& \equiv \langle \text{Prop. } \mathit{func}^n\rangle & (\neg b \wedge h^{n+1}\,a) \preceq p\\
& \equiv \langle \text{Def. } f\rangle & f_{n+1}\,a \preceq p.
\end{array}$$

6.6 Intermezzo: From Galois Connections to Converses

The beauty of Galois connections is most salient at the algebraic level in a point-free style [Backhouse 2002]. Support for this style in our formalism is given by Boute [2003], but since developing it fully is beyond the scope of this article, we feel free to do some calculations pointwise. Also, assuming matching types as in Section 6.1.3, we write $\langle\, F \mid G\,\rangle$ for $\forall x:\mathcal{D}\,F\,.\,\forall y:\mathcal{D}\,G\,.\,F\,x \sqsubseteq y \equiv x \preceq G\,y$.

A Galois connection in general is not symmetric, and this is also the case for a/p-conditions. Yet, note that $\forall (s\,.\,f\,a \Rightarrow p) \equiv \forall (s\,.\,a \Rightarrow g\,p) \equiv \forall (s\,.\,\neg (f\,a) \vee p) \equiv \forall (s\,.\,\neg a \vee g\,p)$, so letting $q := \neg a$ yields $\langle\, f \mid g\,\rangle \equiv \forall (p,q):B^2\,.\,\forall (s\,.\,f^*\,q \vee p) \equiv \forall (s\,.\,q \vee g\,p)$.

This suggests the following: A proposition transformer g is called the *converse* [Dijkstra and Scholten 1990] of a proposition transformer f iff they satisfy

$f \leftharpoonup g$ according to the definition

Definition 6.31. $f \leftharpoonup g \;\equiv\; \forall (p,q) : B^2 \,.\, \forall (s \,.\, f\, q \vee p) \equiv \forall (s \,.\, q \vee g\, p)$.

Observe that $g \leftharpoonup f \;\equiv\; f \rightharpoonup g$. In contexts more general than a/p-conditions, existence is not guaranteed, but uniqueness is: $f \rightharpoonup g \wedge f \rightharpoonup g' \Rightarrow g = g'$.

As shown, $\langle\, f \mid g \,\rangle \;\equiv\; f^* \rightharpoonup g$ and, in particular, $(\mathsf{slp}\, c)^* \rightharpoonup \mathsf{wla}\, c$, the equivalent of Theorem (12.2) in Dijkstra and Scholten [1990, p. 211]. Some equally simple and useful properties are:

(1) $(f \circ g)^* = f^* \circ g^*$ and hence, by induction, $(f^*)^n = (f^n)^*$.

(2) $f \rightharpoonup g \wedge f' \rightharpoonup g' \Rightarrow f \circ f' \rightharpoonup g' \circ g$; hence, by induction, $f \rightharpoonup g \Rightarrow f^n \rightharpoonup g^n$.

Property (2) is essentially Theorem (11.11) in Dijkstra and Scholten [1990, p. 206].

Finally, if h and k are families of proposition transformers having common the index set I and satisfying $\forall i : I \,.\, h_i \;\rightharpoonup\; k_i$, then the proposition transformers f and g, defined by $f\, q \;\equiv\; \forall i : I \,.\, h_i\, q$ and $g\, p \;\equiv\; \forall i : I \,.\, k_i\, p$, respectively, satisfy $f \;\rightharpoonup\; g$. This is essentially Theorem (11.14) in Dijkstra and Scholten [1990, p. 208]. The proof is similar to the calculation leading to Theorem 6.29.

Arguments based on Galois connections can be rewritten in terms of converses.

6.7 Practical Calculation Rules for Repetition

Again let $c := [\![\, \texttt{do}\ b\ \texttt{->}\ c'\ \texttt{od} \,]\!]$. The formulas in Theorems 6.26 and 6.28 are difficult to calculate in concrete situations. The alternatives that follow are easier in practice.

6.7.1 *Invariants.* We call i a *loop invariant* for c iff $\{i \wedge b\}\ c'\ \{i\}$. Intuitively, if c' has to be executed, the invariant i is preserved. Hence, a loop that starts with i satisfied and terminates is expected to establish $\neg\, b$ and i. Formally:

THEOREM 6.32. *If* $\{i \wedge b\}\ c'\ \{i\}$ *then* $\{i\}\ \texttt{do}\ b\ \texttt{->}\ c'\ \texttt{od}\ \{i \wedge \neg\, b\}$.

Proving Theorem 6.32 via (6.28) is routine: $\forall s \,.\, i \Rightarrow \mathsf{wla}\, c\, (i \wedge \neg\, b)$ is transformed into the equivalent $\forall n : \mathbb{N} \,.\, \forall s \,.\, i \Rightarrow (\mathsf{wla}\, d)^n\, (b \vee i)$, which is proved by induction using $\{i \wedge b\}\ c'\ \{i\} \;\equiv\; \{i\}\ d\ \{i\}$. The details are omitted, since Theorem 6.32 will be obviated by Theorem 6.34, which is more practical and independent of (6.28).

6.7.2 *Bound Expressions.* Let D be a set with order $<$ and let $W : \mathcal{P}\, D$ be a well-founded subset under $<$. Then, an expression e of type D is a *bound expression* for c iff (i) $\forall s \,.\, b \Rightarrow e \in W$; (ii) $\forall w : W \,.\, [b \wedge w = e]\ c'\ [e < w]$. Hence the following

Definition 6.33. Proposition i and expression e are an *invariant/bound pair* for the loop c iff (i) $\forall s \,.\, i \wedge b \Rightarrow e \in W$ and (ii)$\forall w : W \,.\, [w = e \wedge i \wedge b]\ c'\ [e < w \wedge i]$.

The next theorem is quite general, and unlike (6.26), does not assume a bounded number of iterations. This allows more interesting forms of nondeterminacy.

THEOREM 6.34. *If i, e is an invariant/bound pair for c then* $[i]\, c\, [i \wedge \neg b]$.

PROOF. Let i, e be an invariant/bound pair. The added value being termination, we first calculate $\mathsf{Term}_c\, i$, that is, $\forall s \bullet i \Rightarrow \mathsf{t}c$. Initial steps are factored out for reuse.

$$\begin{array}{lll} i \Rightarrow \mathsf{t}c & \equiv \langle \text{Thm. 6.21} \rangle & i \Rightarrow b \Rightarrow \mathsf{wa}\, c'\, (\mathsf{t}c) \\ & \equiv \langle \text{Shunting } \wedge \rangle & i \wedge b \Rightarrow \mathsf{wa}\, c'\, (\mathsf{t}c) \\ & \equiv \langle \text{Def. 6.33.i} \rangle & e \in W \wedge i \wedge b \Rightarrow \mathsf{wa}\, c'\, (\mathsf{t}c) \\ & \equiv \langle \text{Shunting } \wedge \rangle & e \in W \Rightarrow i \Rightarrow b \Rightarrow \mathsf{wa}\, c'\, (\mathsf{t}c) \\ & \equiv \langle \text{Thm. 6.21} \rangle & e \in W \Rightarrow i \Rightarrow \mathsf{t}c. \end{array}$$

This yields a LEMMA: $i \Rightarrow \mathsf{t}c \;\equiv\; e \in W \Rightarrow i \Rightarrow \mathsf{t}c$, used first in calculating $\mathsf{Term}_c\, i$.

$$\begin{array}{lll} \mathsf{Term}_c\, i & \equiv \langle \text{Def. Term (5.2)} \rangle & \forall s \bullet i \Rightarrow \mathsf{t}c \\ & \equiv \langle \text{Lem. preceeding} \rangle & \forall s \bullet e \in W \Rightarrow i \Rightarrow \mathsf{t}c \\ & \equiv \langle \text{One-p. rule} \rangle & \forall s \bullet \forall w : W \,.\, w = e \Rightarrow i \Rightarrow \mathsf{t}c \\ & \equiv \langle \text{Swapping } \forall/\forall \rangle & \forall w : W \,.\, \forall s \bullet w = e \Rightarrow i \Rightarrow \mathsf{t}c \\ & \equiv \langle \text{Intro. } P \rangle & \forall w : W \,.\, P\, w \\ & \equiv \langle \text{Well-founded } W \rangle & \forall w : W \,.\, \forall (v : W \,.\, v < w \Rightarrow P\, v) \Rightarrow P\, w. \end{array}$$

Here, $P : D \to \mathbb{B}$ with $P\, v \;\equiv\; \forall s \bullet v = e \Rightarrow i \Rightarrow \mathsf{t}c$. The last step is the equivalence between well-foundedness and supporting induction [Gries and Schneider 1993]. Finally, we prove the last line by calculating for arbitrary $w : W$

$$\begin{array}{lll} \forall v : W \,.\, v < w \Rightarrow P\, v & \equiv \langle \text{Def. } P \rangle & \forall v : W \,.\, v < w \Rightarrow \forall s \bullet v = e \Rightarrow i \Rightarrow \mathsf{t}c \\ & \equiv \langle \text{Rearrange } \forall \rangle & \forall s \bullet \forall v : W \,.\, v = e \Rightarrow v < w \Rightarrow i \Rightarrow \mathsf{t}c \\ & \equiv \langle \text{One-pt. rule} \rangle & \forall s \bullet e \in W \Rightarrow e < w \Rightarrow i \Rightarrow \mathsf{t}c \\ & \equiv \langle \text{Shunt, lem.} \rangle & \forall s \bullet e < w \Rightarrow i \Rightarrow \mathsf{t}c \\ & \Rightarrow \langle \text{Shunt, (12)} \rangle & \forall s \bullet \mathsf{wa}\, c'\, (e < w \wedge i) \Rightarrow \mathsf{wa}\, c'\, (\mathsf{t}c) \\ & \Rightarrow \langle \text{Def. 6.33.ii)} \rangle & \forall s \bullet w = e \wedge i \wedge b \Rightarrow \mathsf{wa}\, c'\, (\mathsf{t}c) \\ & \equiv \langle \text{Shunt, (6.21)} \rangle & \forall s \bullet w = e \Rightarrow i \Rightarrow \mathsf{t}c \\ & \equiv \langle \text{Def. } P \rangle & P\, w \;. \end{array}$$

In our original proof for (6.34), we just joined $\mathsf{Term}_c\, i$ to (6.32), yielding $[i]\, c\, [i \wedge \neg b]$.

However, our proof for (6.32) depends on (6.28) and hence, on the underlying operational arguments from Dijkstra and Scholten [1990], which are rather intricate. To obtain a simpler and self-contained treatment, we calculate directly $[i]\, c\, [i \wedge \neg b]$, or equivalently, $\forall s \bullet i \Rightarrow \mathsf{wa}\, c\, (i \wedge \neg b)$.

$$\begin{array}{lll} i \Rightarrow \mathsf{wa}\, c\, (i \wedge \neg b) & \equiv \langle \text{Thm. 6.24} \rangle & i \Rightarrow (\neg b \Rightarrow i \wedge \neg b) \wedge (b \Rightarrow \mathsf{wa}\, c'\, (\mathsf{wa}\, c\, (i \wedge \neg b))) \\ & \equiv \langle \text{Prop. calc.} \rangle & i \Rightarrow b \Rightarrow \mathsf{wa}\, c'\, (\mathsf{wa}\, c\, (i \wedge \neg b). \end{array}$$

The formal similarity to $i \Rightarrow \mathsf{t}c \;\equiv\; \langle \text{Thm. 6.21} \rangle\; i \Rightarrow b \Rightarrow \mathsf{wa}\, c'\, (\mathsf{t}c)$ in the calculation for $\mathsf{Term}_c\, i$ is striking. Moreover, this calculation uses no intrinsic properties of $\mathsf{t}c$ apart from Theorem 6.21, and is entirely an exercise in predicate calculus. Hence, as the reader can verify, just replacing $\mathsf{t}c$ by $\mathsf{wa}\, c\, (i \wedge \neg b)$ in this calculation completes the proof for $\forall s \bullet i \Rightarrow \mathsf{wa}\, c\, (i \wedge \neg b)$ and for Theorem 6.34. □

As promised, nondeterminacy need not be bounded.

Using (6.34) in practice is best done via a *checklist* [Gries and Schneider 1993]: To show $[a]$ `do` b `->` c' `od` $[p]$, find suitable i, e and prove

(i) i satisfies $[i \wedge b]\ c'\ [i]$;
(ii) i satisfies $a \Rightarrow i$;
(iii) i satisfies $i \wedge \neg b \Rightarrow p$;
(iv) e satisfies $i \wedge b \Rightarrow e \in W$; and
(v) e satisfies $[w = e \wedge i \wedge b]\ c'\ [e < w]$ for arbitrary $w : W$.

Heuristics for finding i are (a) writing p as a conjunct of two propositions and taking one of these as i, the negation of the other as b; and (b) making a constant parameter of the problem into a variable. More can be found in Gries and Schneider [1993].

7. RAMIFICATIONS AND LINKS WITH OTHER THEORIES

Background for the following exploration is the deeper motive for our approach:

—Pushing the limits of what can be done with simple and basic theories.
—Tightening the links with systems modelling methods in classical engineering.

The first motive is in the spirit of some interesting observations in Lamport [2004]; the second is shared by a program initiated in the U.S. [Lee and Messerschmitt 1998; Lee and Varaiya 2003] aimed at evolving towards a unified discipline of electrical and computer engineering (ECE).

Fully exploring how the approach can be applied to all other theories is beyond the scope of any article. Hence, only some samples are presented, and even so, rather unevenly: in some detail if the application yields immediate and interesting rewards, by a few remarks and references if deeper investigation is clearly necessary.

We start with some general observations and choices of conventions for later use.

7.1 On Formulations, Abstractions, and Concretizations

7.1.1 *Rationale in Algebraic Systems Description.* Two views can be distinguished.

—The *prescriptive* view: The central topic is the algebra; it is used to prescribe abstract properties, whereas the instances ("models" in the nomenclature of logic) just describe realizations (systems) satisfying them in a given interpretation.
—The *descriptive* view: The central topic is system behavior, and the algebra is primarily a means for calculational reasoning at an abstract level (without irrelevant details) in a compact (e.g., point-free) style.

Some authors [Dijkstra 1976; Dijkstra and Scholten 1990; Nelson 1989] strongly advocate the prescriptive view. Others (less explicitly) seem to consider the correspondence between the abstract formalism and program execution (or some other kind of system behavior) also important [Gries and Schneider 1993; Hehner 2004; Hoare 1969; Hoare and Jifeng 1998], and sometimes provide a careful formal treatment of this correspondence [Back 1983; Dijkstra 1994, 1998].

In our view, abstract algebras are very valuable for capturing common aspects of different useful instances (models), for example, as in group, ring, lattice, ... theory. However, we also observe that in engineering the reverse is more common: A single model (specific to a class of systems) may require different theories pertaining to different aspects. Moreover, most programming theories (perhaps due to their specificity) appear to have only one useful model.

Hence, we have no compunctions against the descriptive view. Yet, we achieve abstraction by deriving theorems expressible in point-free style (i.e., not referring to points in the interpretation domain) and then, as a discipline, using only those theorems, which thereby play the same role as axioms in an abstract algebra. This yields the best of both worlds and avoids the so-called "formalization gap" [Dijkstra 1998].

Generic functionals [Boute 2003] play a crucial role in deriving such theorems, but for theorems with counterparts in other theories, we shall often replace applications of generic functionals (like $\widehat{\wedge}$ and $\widehat{\Rightarrow}$) by single abstract symbols (like $\sqcap$ and $\dashv\!\exists$) for emphasis.

7.1.2 *Conventions in Propositional Formulations.* As a concession to direct compatibility with practical application, where it is most common to reason about programs via *propositions* (like $\mathrm{x} > 3$ and $\mathrm{x} > 7$ in $\mathsf{wa}[\![\mathrm{x} := \mathrm{x}+4]\!](\mathrm{x} > 7) = (\mathrm{x} > 3)$), most derivations thus far are given in the propositional style, also to show that it can be done properly and fluently with little overhead. The price is some care with the use of variables and implicit bindings, requiring one or two calculations for familiarization.

We must also briefly comment on a ubiquitous convention in mathematics that is quite useful provided we remain aware of it, namely, implicit quantification.

As observed by the author [Boute 2002] and in Lamport [2002], if we write $x < y$, this is usually interpreted as a Boolean value that depends on x and y. However, if we write $x + y = y + x$, this is usually meant as a theorem where quantification is left implicit yet is understood (making it explicit is done by *generalization*).

Specifically, whenever we write an expression like $\mathsf{T}_c\, s$ or $\mathsf{r}\, c$ in isolation, it is just a Boolean expression, but formulas like $\mathsf{t}[\![c';c'']\!] \equiv \mathsf{t}\, c' \wedge \exists s' \bullet \mathsf{r}\, c' \Rightarrow (\mathsf{t}\, c'')[^{s}_{s'}$ appearing as statements of theorems are implicitly quantified (over meta- and state variables). For instance, a theorem like $\mathsf{wa}[\![c';c'']\!]\, p \equiv \mathsf{wa}\, c'\, (\mathsf{wa}\, c''\, p)$ is implicitly quantified over c', c'', p and s, reading $\forall (c', c'') : C^2 \,.\, \forall\, p : B \,.\, \forall s : \mathbf{S} \,.\, \mathsf{wa}[\![c';c'']\!]\, p \equiv \mathsf{wa}\, c'\, (\mathsf{wa}\, c''\, p)$. This is why its equivalent in Dijkstra and Scholten [1990] is written $[wp.\text{"S0;S1"}.X \equiv wp.\text{S0}.(wp.\text{S1}.X)]$ for all X.

Arguably, quantification is best made explicit to ensure clarity. On the other hand, the notational clutter of many quantifications reduces clarity. Experience indicates that implicit quantification is quite "safe", except in inductive proofs, where explicit quantification helps to avoid errors (illicit instantiations) in the inductive step.

Borrowing Dijkstra's [ ] by writing $[p]$ for the universal closure of p over all state variables (as in Hoare and Jifeng [1998]) is a good compromise in specific

contexts (such as programming theories) if it cannot cause confusion with other uses of [ ], but even then, quantification remains implicit for commands (e.g., S0, S1) and verbal for propositions (e.g., X).

A few tacit conventions further smooth the propositional style. Readers may have noticed that, as a mnemonic aid, we write $\mathsf{wa}\,cp$, $\mathsf{r}c$, and $\mathsf{t}c$ for propositional-style expressions (dummies hidden), but subscript, as in R_c and T_c, for predicative-style expressions. Other conventions minimize parentheses and $[\![\ ]\!]$-quotes, for example, around subscripts. There is more, but we do not elaborate since all this is quite secondary.

Indeed, more important are alternative formulations that are not subject to restrictions imposed by immediate practicality, but are aimed at a broader scope, especially unification with theories in other areas of ECE.

In such a wider context, predicates are definitely superior, as explained next.

7.1.3 *Predicates and Engineering Formalisms.* Consider, for instance, Hoare-triples as (parameterized) predicates on predicates, for example, of type $\mathsf{pred}_{\mathbf{S}} \times C \times \mathsf{pred}_{\mathbf{S}} \to \mathbb{B}$, and $\mathsf{Term} : C \to \mathsf{pred}_{\mathbf{S}} \to \mathbb{B}$, with image definitions given by

Definition 7.1.
$$\begin{array}{lll} \text{(a)} & \{A\}\ c\ \{P\} & \equiv\ \forall\, \backslash s \mathbf{.} \forall\, s' \mathbf{.} \mathsf{R}_c\,(\backslash s, s') \Rightarrow A\,\backslash s \Rightarrow P\,s'\ ; \\ \text{(b)} & [A]\ c\ [P] & \equiv\ \{A\}\ c\ \{P\} \wedge \mathsf{Term}_c\,A\ ; \\ \text{(c)} & \mathsf{Term}_c\,A & \equiv\ \forall\, \backslash s : \mathbf{S}\,.\, A\,\backslash s \Rightarrow \mathsf{T}_c\,\backslash s\ . \end{array}$$

Here and in the sequel, $\mathsf{pred}_X = X \to \mathbb{B}$ for the "predicate space" over a set X.

The "pure mathematical" style (i.e., without syntactic mappings) of the calculations is illustrated by reformulating the first calculation of Section 6.1 predicatively:

$$\begin{array}{lll} \{A\}\ c\ \{P\} & \equiv \langle \text{Def. } \{A\}\ c\ \{P\} \rangle & \forall\, \backslash s \mathbf{.} \forall\, s' \mathbf{.} \mathsf{R}_c\,(\backslash s, s') \Rightarrow A\,\backslash s \Rightarrow P\,s' \\ & \equiv \langle \text{Shunting } \Rightarrow \rangle & \forall\, \backslash s \mathbf{.} \forall\, s' \mathbf{.} A\,\backslash s \Rightarrow \mathsf{R}_c\,(\backslash s, s') \Rightarrow P\,s' \\ & \equiv \langle \text{Ldist. } \Rightarrow/\wedge \rangle & \forall\, \backslash s \mathbf{.} A\,\backslash s \Rightarrow \forall\, s' \mathbf{.} \mathsf{R}_c\,(\backslash s, s') \Rightarrow P\,s'\ . \end{array} \tag{23}$$

A predicative style raises the abstraction level, for example, with an abstract state space (generically written as $\mathbb{S}$) and the choice of dummies not depending on concrete program variables. This allows creating a space of abstractions (for specifications) extending beyond behavioral descriptions (of realizations).

This view supports a strong analogy with systems modelling in electronics: The abstract space [Lee and Varaiya 2003] is that of signals and systems as signal transformers, the concrete space is that of electric phenomena (voltages, currents) and circuits. The device level is comparable to assignment in programs (involving reference to program variables).

This similarity is not just vague speculation, but extends to the formal level. Observe first that operators like wla are replaced by genuine predicate transformers (noting that, even for abstract spaces, $\backslash s$, s' remains good mnemonics):

$$\mathbf{def}\ \mathsf{Wla\_} : C \to \mathsf{pred}_{\mathbb{S}} \to \mathsf{pred}_{\mathbb{S}}\ \mathbf{with}\ \mathsf{Wla}_c\,P\,s \equiv \forall\, s' : \mathbb{S}\,.\, \mathsf{R}_c\,(s, s') \Rightarrow P\,s'\ ;$$
$$\mathbf{def}\ \mathsf{Wa\_} : C \to \mathsf{pred}_{\mathbb{S}} \to \mathsf{pred}_{\mathbb{S}}\ \mathbf{with}\ \mathsf{Wa}_c\,P = \mathsf{Wla}_c\,P \mathbin{\widehat{\wedge}} \mathsf{T}_c.$$

We obtain a direct formal similarity between the equations in the following list.

$$\begin{aligned}
\mathsf{Wla}_c\ P\ s &\equiv \forall\, s' : \mathbb{S} \,.\, \mathsf{R}_c\,(s, s') \Rightarrow P\ s' \\
\mathsf{Slp}_c\ A\ s &\equiv \exists\, \backprime s : \mathbb{S} \,.\, \mathsf{R}_c\,(\backprime s, s) \wedge A\ \backprime s \\
\mathsf{Rsp}_d\ f\ x &= \mathtt{I}\, \backprime x : \mathbb{I} \,.\, \mathsf{G}_d\,(\backprime x, x) \cdot f\ \backprime x \qquad \text{(linear } d\text{)} \\
\mathsf{Rsp}_d\ f\ t &= \mathtt{I}\, \backprime t : \mathbb{R} \,.\, \mathsf{h}_d\,(t - \backprime t) \cdot f\ \backprime t \qquad \text{(for LTI } d\text{)} \\
\mathcal{F}\ f\ \omega &= \mathtt{I}\, t : \mathbb{R} \,.\, \exp(-j \cdot \omega \cdot t) \cdot f\ t
\end{aligned}$$

The first two refer to our current topic (and can be generalized). The third expresses for a linear system d with Green's function G_d the "response" $\mathsf{Rsp}_d\ f$ to "input" f. By convention, $\mathtt{I}\ f = \int_a^b f\ x \cdot \mathrm{d}\, x$ if $\mathcal{D}\ f$ is the interval from a to b.

A mechanics example: If d is a beam and f is a continuous load distribution, then in the usual first order approximation, $\mathsf{Rsp}_d\ f\ x$ is the shearing force, bending moment, slope, or deflection at position x if we take $\mathsf{G}_d\,(\backprime x, x)$ to be the shearing force, bending moment, slope, or deflection (respectively) at position x for unit point load in $\backprime x$.

Closer to home: with signals as functions over time domain $\mathbb{R}$ (particularizing the interval of interest $\mathbb{I}$) and systems as signal transformers of type[6] $\mathsf{sig}_{\mathbb{C}} \to \mathsf{sig}_{\mathbb{C}}$, the response of linear system d to input signal f is $\mathsf{Rsp}_d\ f$. Assuming time independence yields the fourth equation, where h_d is the impulse response of d.

The last equation obviously expresses the Fourier transform.

7.1.4 *Conventions Regarding Predicates and Relations.* For any set S, in this context called the *base set*, $\mathsf{pred}_S = S \to \mathbb{B}$ and $\mathsf{rel}_S = \mathsf{pred}_{S^2}$. For the reasons given earlier, we often use $\sqcup$, $\dashv\!\exists$, $\sqcap$, $\bigsqcup$, $\bot$, $\top$ instead of $\overline{\neg}$, $\widehat{\Rightarrow}$, $\widehat{\wedge}$, $\widehat{\vee}$, $S \bullet 0$, $S \bullet 1$, respectively.

For predicates P and Q with common base set,[7] we define $Q \sqsubseteq P \ \equiv\ \forall (Q \dashv\!\exists P)$, read "*$Q$ is at least as strong as P*" or "*Q refines P*". Clearly, $Q \sqsubseteq P \ \equiv\ Q = P \sqcap Q$.

In saying "*Q refines P*," we borrow terminology from the *refinement calculus*, for which an overview is given in Hancock [2004] and some of the schools of thought are represented by Back and von Wright [1992], Hehner [1999, 2004], Morgan [1994], and Morris [1987].

Still, some subtle differences due to independent developments must be considered, for example, in the refinement literature some would write $P \sqsubseteq Q$ ("P is refined by Q") instead. Since $\Rightarrow$ coincides with $\leq$ on $\mathbb{B}$, we prefer $Q \sqsubseteq P \ \equiv\ \forall\, s : S \,.\, Q\ s \leq P\ s$, or with sets $Q \sqsubseteq P \ \equiv\ \{s : S \mid Q\ s\} \subseteq \{s : S \mid P\ s\}$, which aligns the various ordering symbols $\sqsubseteq$, $\leq$, $\subseteq$, $\preceq$ (as is convenient in a wider context).

The *setgraph* of a predicate P is defined by $\mathsf{G}_P = \{x : \mathcal{D}\ P \mid P\ x\}$. Clearly (assuming common base set), $Q \sqsubseteq P \ \equiv\ \mathsf{G}_Q \subseteq \mathsf{G}_P$ and similar isomorphism properties hold. When we use this concept, the predicates are usually relations, for example, with base set $X \times Y$.

7.1.5 *Observations, Specifications, Refinements and Realizability.* In $Q \sqsubseteq P$ as a refinement relation, P may be a *specification* (abstract, minimal

[6] In general, $\mathsf{sig}_X = \mathbb{T} \to X$ for the signal space over time domain $\mathbb{T}$, here $\mathbb{T} = \mathbb{R}$.

[7] This restriction can be lifted with proper generalizations [Boute 2002] with not needed here.

Table VII. Characterizing a One-Bit Machine

x, x'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0, 0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0, 1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1, 0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1, 1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

(a) The 16 relations of type $\mathbb{B}^2 \to \mathbb{B}$

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	u	0	1	d	u	0	1	d	u	0	1	d	u	0	1	d
1	u	u	u	u	0	0	0	0	1	1	1	1	d	d	d	d

(b) The 16 functions of type $\mathbb{B} \to \{u, 0, 1, d\}$.

detail) and Q a *realization* (concrete implementation), usually with intermediate levels in between.

Often the fact that $\bot \sqsubseteq P$ is interpreted as saying that $\bot$ implements anything (in a trivial but unintended way), and similarly that $\top$ can be implemented by anything. Theory design decisions such as choosing fixpoints are based on such interpretations.

As argued next, a concrete view of the predicate space suggests that these interpretations are not always appropriate, in the sense that $\bot$ often does not describe any implementation. This view is based on analogies with classical physics and systems theory, and with the use of "don't care terms" in specifying digital circuits [Gries and Schneider 1993; McCluskey 1965].

The principle is that a predicate describing an *implementation* must precisely characterize all *observable* phenomena. It is always (implicitly) assumed that the system under observation is of the considered class and that the observation is done properly. The predicate $\bot$ then simply indicates that the assumption is not satisfied.

For instance, the possible observations of signals on an ideal linear amplifier with gain G are captured by $R_G(x, y) \equiv \forall t : \mathbb{T} . y\, t = G \cdot x\, t$ for any input signal x and output signal y. Clearly, $R_G \neq \bot$, since for any input signal x there is a signal y expressing the matching output, otherwise the device does not implement any amplifier.

Here are two examples that can be elaborated more completely in a few lines.

—The *One-Bit Machine*. The simplest possible nontrivial system has 1 value and 1 variable, but more enlightening is considering 2 values (say, in $\mathbb{B}$) and 2 variables (say, x and x'). All possible observations on such a system can be described by a relation of type $\mathbb{B}^2 \to \mathbb{B}$. There are 16 relations of this type, listed in Table VII(a) as a family $R\_ : \square 16 \to \mathbb{B}^2 \to \mathbb{B}$ with $R_i(x, x') \equiv \rho_4\, i\, (2 \cdot x + x')$, where $\rho_n\, i\, j$ is the j-th bit in the n-bit binary representation of i (column i, row j).

Not each of these relations describes a system; which ones do depends on the class of system considered. The *one-bit machine* (OBM) is the class where x is input and x' is output. For every x in $\mathbb{B}$, some x' in $\mathbb{B}$ is observable, unless the OBM is defective. Hence, R_i describes an OBM only

if $\forall x : \mathbb{B} . \exists x' : \mathbb{B} . R_i(x, x')$, that is, 7 relations (columns 0, 1, 2, 3, 4, 8, 12; $R_0 = \bot$) do *not* describe OBMs.

For OBMs, more synoptic descriptions than relations are I/O functions of type $\mathbb{B} \to \{u, 0, 1, d\}$, with u for *undefined* and d for *don't care*, as in Table VII(b). The 4 functions of type $\mathbb{B} \to \mathbb{B}$ describe OBMs (columns 5, 6, 9, 10). The 5 remaining functions with ds express nondeterminism (as a realization) or design freedom (as a specification). Hence, as a relation, $\top$ (column 15) is *not* an absurdity.

There is a close correspondence to Gries and Schneider [1993], specifying circuits by propositions with I/O variables. For instance, specification $x' \wedge x \equiv x$ is equivalent to $x \Rightarrow x'$ and corresponds to R_{11}. Although $R_i \sqsubseteq R_{11}$ for any i in $\{0, 1, 2, 3, 8, 9, 10, 11\}$, only 9 and 10 describe circuits. Specification $x' \equiv \neg x'$ is $\bot$, and not realizable.

In this context, realizations may involve extra variables for internal connections, for instance, the data selector specification $z = (x, y)s$ is realized by the AND/OR circuit $(u \equiv \neg s \wedge x) \wedge (v \equiv s \wedge y) \wedge (z \equiv u \vee v)$, or to avoid hazards [McCluskey 1965], $(u \equiv \neg s \wedge x) \wedge (v \equiv s \wedge y) \wedge (w \equiv x \wedge y) \wedge (z \equiv u \vee v \vee w)$.

—The *Square Machine*. With state space $\mathbf{S} := \mathbb{N}$, consider the family of relations

$$\textbf{def } R\_ : \mathbb{N}' \to \mathbf{S}^2 \to \mathbb{B} \textbf{ with } R_i(n, n') \equiv n < i \Rightarrow n' = n^2.$$

Observe that $j \leq i \Rightarrow R_i \sqsubseteq R_j$ for all $(i, j) : \mathbb{N}'^2$. Proof: for any n and n' in $\mathbf{S}$,

$$\begin{aligned} j \leq i \;&\Rightarrow\; \langle \text{Trans. } \leq \rangle \; n < j \Rightarrow n < i \\ &\Rightarrow\; \langle \text{Trans. } \Rightarrow \rangle \; (n < i \Rightarrow n' = n^2) \Rightarrow (n < j \Rightarrow n' = n^2). \end{aligned}$$

Observe that $R_\infty(n, n') \equiv n' = n^2$, describing the "complete" square machine, whereas $R_0(n, n') \equiv 1$ or $R_0 = \top$ is the "empty" one, with arbitrary behavior over $\mathbf{S}$ (R_i guarantees squaring only for natural numbers smaller than i) .

7.1.6 *Structure and Pragmatics of Specifications.* We place the preceding examples in a wider context, starting with function specifications.

In set-theoretic frameworks, relations are defined as sets of pairs with functions as a subcase. In other frameworks, including ours (where functions are objects in their own right and relations are $\mathbb{B}$-valued functions), such sets of pairs are called *graphs*.

The graph $\mathcal{G}_f$ of a function f is defined by $\mathcal{G}_f = \{x, f\,x \mid x : \mathcal{D}\,f\}$. We also define $f \subseteq g \equiv f = g \restriction \mathcal{D}\,f$, read "$f$ is a *subfunction* of g." Note that $f \subseteq g \equiv \mathcal{G}_f \subseteq \mathcal{G}_g$.

For relations, we saw that $R \sqsubseteq S \equiv \mathsf{G}_R \subseteq \mathsf{G}_S$, given a common base set. Still, it would be misleading to use the set-theoretic definitions of functions (as relations) as specification relations; this yields inclusion in the wrong direction.

A better design decision regarding relational specification for functions in $X \nrightarrow Y$ ($\nrightarrow$ being defined by $f \in X \nrightarrow Y \equiv \mathcal{D}\,f \subseteq X \wedge \mathcal{R}\,f \subseteq Y$) is the following.

$$\textbf{def } \mathsf{R}\_ : (X \nrightarrow Y) \to (X \times Y \to \mathbb{B}) \textbf{ with } \mathsf{R}_f(x, y) \equiv x \in \mathcal{D}\,f \Rightarrow y = f\,x. \quad (24)$$

The difference with graphs is highlighted by $x, y \in \mathcal{G}_f \equiv x \in \mathcal{D}\, f \wedge y = f\, x$ (exercise). The idea is ensuring $\forall (x, y) : X \times Y \,.\, x \notin \mathcal{D}\, f \Rightarrow \mathsf{R}_f\,(x, y)$, making images of arguments outside the domain of "don't care" values. Calculating $\mathsf{R}_f \sqsubseteq \mathsf{R}_g$ yields

THEOREM 7.2. $\mathsf{R}_f \sqsubseteq \mathsf{R}_g \equiv (|Y| > 1 \Rightarrow \mathcal{D}\, g \subseteq \mathcal{D}\, f) \wedge \forall x : \mathcal{D}\, f \cap \mathcal{D}\, g \,.\, f\, x = g\, x$.

The calculation starts by expanding $\mathsf{R}_f \sqsubseteq \mathsf{R}_g$, shunting $\Rightarrow$, then using the propositional rule $(a \Rightarrow b) \Rightarrow c \equiv (\neg a \Rightarrow c) \wedge (a \Rightarrow b \Rightarrow c)$. Further, by routine predicate calculus $\forall (x : \mathcal{D}\, g \,.\, \exists (y : Y \,.\, y \neq g\, x) \Rightarrow x \in \mathcal{D}\, f) \wedge \forall (x : \mathcal{D}\, g \,.\, x \in \mathcal{D}\, f \Rightarrow f\, x = g\, x)$, noting that $\forall (x : \mathcal{D}\, g \,.\, \exists (y : Y \,.\, y \neq g\, x) \equiv |Y| > 1)$ and cleaning up.

Important is the corollary for $|Y| > 1$ (as in all nontrivial situations):

COROLLARY 7.3. *If* $|Y| > 1$ *then* $\mathsf{R}_f \sqsubseteq \mathsf{R}_g \equiv g \subseteq f$.

Let us explore some lessons of applying the preceding examples to programs, and generalize the observations for function specifications to general specifications. By the circuit analogy, we say that an ante/postspecification R is *realizable* iff $\forall \backslash s \,.\, \exists s' \,.\, R\,(\backslash s, s')$.

Consider now Dijkstra's guarded command language. According to the language specification in Dijkstra [1976], the following commands are equivalent to `abort`.

$$c' := [\![\texttt{if [] 0 -> skip fi}]\!] \text{ and } c'' := [\![\texttt{do [] 1 -> skip od}]\!]$$

Indeed, c' has no alternative with a satisfied guard, so the body cannot even start, and no s'-value can be observed. This is reflected in the program equations from Definition 6.15: $\mathsf{R}_{c'}\,(\backslash s, s') \equiv 0$ (so $\mathsf{R}_{c'} = \bot$) and $\mathsf{T}_{c'}\,\backslash s \equiv 0$. This does not mean that c' realizes $\bot$: it rather means that c' is "not a program" (nop) but a program error, as stated in Dijkstra [1976].

Command c'' does not terminate according to Equation (22), hence, no s'-value can be observed, which one might express as $\mathsf{R}_{c''} = \bot$. Yet, this would be just a convention since it does not follow from the program equations. However, a weaker requirement than realizability is the LEM, requiring for any command c that $\forall \backslash s \,.\, \mathsf{T}_c\,\backslash s \Rightarrow \exists s' \,.\, \mathsf{R}_c\,(\backslash s, s')$, implying $\mathsf{R}_{c''} = \bot$. Even weaker than LEM is $\forall \backslash s \,.\, \mathsf{T}_c\,\backslash s \Rightarrow \forall s' \,.\, \mathsf{R}_c\,(\backslash s, s') \equiv Q_c\,(\backslash s, s')$, where Q_c is a specification as desired, possibly assigning a poststate even in case of nontermination.

Comparing $\mathsf{R}_f\,(x, y) \equiv x \in \mathcal{D}\, f \Rightarrow y = f\, x$ from (24) with LEM and weaker forms, the recurring pattern is the appearance of an antecedent. This matches the observation that, in practice, most specifications are not absolute but conditional, leaving design freedom ("don't care" situations) if the condition is not satisfied.

Assume all specifications S are conditional propositions of the form $C \dashv P$. Then $R \sqsubseteq S \equiv C \dashv R \sqsubseteq C \dashv P$ (shunting, distributivity of $\Rightarrow$), so the condition C of the specification may be exploited by all realizations (and intermediate design levels). For predicates over some set Z, calculations similar to those for

Theorem 7.2 yield

$$C' \multimap\!\exists P' \sqsubseteq C \multimap\!\exists P \;\equiv\; \forall(z : Z_{\neg P} \,.\, C\,z \Rightarrow C'\,z) \wedge \forall(z : Z_{C \sqcap C'} \,.\, P'\,z \Rightarrow P\,z). \quad (25)$$

For the important subcase when $Z = X \times Y$ and C refers to X only, the r.h.s. becomes

$$\forall(x : X_C \,.\, \overline{\neg}\forall(y : Y \,.\, P\,(x, y)) \Rightarrow C'\,x) \wedge \forall(x : X_{C \sqcap C'} \,.\, \forall\, y : Y \,.\, P'(x, y) \Rightarrow P\,(x, y)),$$

showing how (25) generalizes (7.2). In particular, $(C \sqsubseteq C') \Rightarrow (C' \multimap\!\exists P \sqsubseteq C \multimap\!\exists P)$ captures the antimonotonicity $j \leq i \Rightarrow R_i \sqsubseteq R_j$ from the square machine example.

7.2 Extending the Collection of Language Constructs

7.2.1 *Simple I/O.* One might consider adding an input command $\mathtt{inp}\,v$ and an output command $\mathtt{out}\,e$. One way to express the program equations is extending R to include sequences of inputs and outputs, for instance, $\mathsf{R}'\_ : C \to (\mathbf{S} \times \mathbf{X} \times \mathbf{Y})^2 \to \mathbb{B}$ with

(a) for the commands thus far, $\mathsf{R}'_c\,((\backslash s, \backslash x, \backslash y), (s', x', y')) \equiv \mathsf{R}_c\,(\backslash s, s') \wedge x', y' = \backslash x, \backslash y$

(b) for an input command, $\mathsf{R}'_{\mathtt{inp}\,v}\,((\backslash s, \backslash x, \backslash y), (s', x', y')) \equiv s', x', y' = \backslash s[^{\backslash v}_{\backslash x\,0}, \sigma\,\backslash x, \backslash y$

(c) for an output command, $\mathsf{R}'_{\mathtt{out}\,e}\,((\backslash s, \backslash x, \backslash y), (s', x', y')) \equiv s', x', y' = \backslash s, \backslash x, (\backslash e \mathrel{>\!\!-} \backslash y)$.

7.2.2 *Guards and Assertions.* In Leavens [1995], we find a brief account of Hesselink's [1992] language constructs based on guards and assertions. Typical axioms are:

Guard $?b$	Assertion $!b$
$\mathsf{wla}\,[\![?b]\!]\,p \equiv b \Rightarrow p$	$\mathsf{wla}\,[\![!b]\!]\,p \equiv b \Rightarrow p$
$\mathsf{wa}\,[\![?b]\!]\,p \equiv b \Rightarrow p$	$\mathsf{wa}\,[\![!b]\!]\,p \equiv b \wedge p$.

Simple reverse engineering by Definitions 6.1 and 6.3 yields the program equations:

Guard $?b$	Assertion $!b$
$\mathsf{R}_{?b}\,(s, s') \equiv b \wedge s' = s$	$\mathsf{R}_{!b}\,(s, s') \equiv b \wedge s' = s$
$\mathsf{T}_{?b}\,s \equiv 1$	$\mathsf{T}_{!b}\,s \equiv b.$

This is left as an exercise, as are the proofs for the program equivalences $\mathtt{skip} = ?1 = !1$ as well as $\mathtt{miracle} = ?0$ (not obeying LEM) and $\mathtt{abort} = !0$. Just for completeness: The program equations for *havoc* [Dijkstra and Scholten 1990] are $\mathsf{r}[\![\mathtt{havoc}]\!] \equiv 1$ and $\mathsf{t}[\![\mathtt{havoc}]\!] \equiv 1$.

These two examples present only one view. Variants of guards and assertions have been used under various names in Back and von Wright [1998], De Bakker [1980], Morgan [1994], Morris [1987], Nelson [1989], and other sources. Also, terminology about `abort`, `miracle`, etc. is not uniform throughout the literature; for instance, in Hoare and Jifeng [1998] "abort" (or "chaos") corresponds to

$\mathsf{R}_c = \top$, and "miracle" to $\mathsf{R}_c = \bot$, without reference to a termination equation such as T_c.

7.2.3 *Choice and Nondeterminism.* Hesselink defines a choice operator $[]$ by the axioms

$$\mathsf{wla}\,[\![c' \,[]\, c'']\!]\,p \equiv \mathsf{wla}\,c'\,p \wedge \mathsf{wla}\,c''\,p \text{ and } \mathsf{wa}\,[\![c' \,[]\, c'']\!]\,p \equiv \mathsf{wa}\,c'\,p \wedge \mathsf{wa}\,c''\,p,$$

whence the program equations $\mathsf{r}\,[\![c' \,[]\, c'']\!] \equiv \mathsf{r}\,c \vee \mathsf{r}\,c'$ and $\mathsf{t}\,[\![c' \,[]\, c'']\!] \equiv \mathsf{t}\,c \wedge \mathsf{t}\,c'$. Generalization to $[]\, i : I \,.\, c_i$ assuming nonempty I is immediate. Together with guards, this synthesizes the familiar conditional: `If` b `then` c' `else` c'' `fi` $= (b?\,;\,c')\,[]\,(\neg b?\,;\,c'')$.

The nondeterminism of $[]$ is known as *demonic*. Let us write Δ for $[]$ as a matching symbol for the *angelic* counterpart ∇ defined by $\mathsf{wa}\,[\![c' \nabla c'']\!]\,p \equiv \mathsf{wa}\,c'\,p \vee \mathsf{wa}\,c''\,p$.

Expanding $\mathsf{wla}\,cp \equiv \mathsf{wla}\,c'\,p \vee \mathsf{wla}\,c''\,p$ by (6.1) does not yield an expression for $\mathsf{r}\,c$, mainly because $\forall\,P \vee \forall\,Q \Rightarrow \forall\,(P \mathbin{\widehat{\vee}} Q)$. The closest is $\mathsf{r}\,c \equiv \mathsf{r}\,c' \wedge \mathsf{r}\,c''$, yielding $\mathsf{wla}\,c'\,p \vee \mathsf{wla}\,c''\,p \preceq \mathsf{wla}\,cp$. Similarly, in expanding $\mathsf{wa}\,cp \equiv \mathsf{wa}\,c'\,p \vee \mathsf{wa}\,c''\,p$ by (6.3) the distributivity of $\vee/\wedge$ generates undesirable "cross modulation products." Taking $\mathsf{t}\,c \equiv \mathsf{t}\,c' \vee \mathsf{t}\,c''$ gets only $\mathsf{wa}\,c'\,p \vee \mathsf{wa}\,c''\,p \preceq \mathsf{wa}\,cp$.

Conversely, the pair of equations $\mathsf{r}\,c \equiv \mathsf{r}\,c' \wedge \mathsf{r}\,c''$ and $\mathsf{t}\,c \equiv \mathsf{t}\,c' \vee \mathsf{t}\,c''$ cannot be expressed via proposition or predicate transformers.

The fact that angelic nondeterminism has no counterpart as program equations does not prevent using it in specifications in the sense that $\mathsf{wa}\,c'\,p \preceq \mathsf{wa}\,[\![c' \nabla c'']\!]\,p$ and $\mathsf{wa}\,c''\,p \preceq \mathsf{wa}\,[\![c' \nabla c'']\!]\,p$, viewing $\preceq$ as propositional refinement.

Thorough treatments of combining specifications are given from a lattice-theoretic perspective in Back and von Wright [1992] and in Leino and Manohar [1999].

7.3 Calculational Semantics in Relation to Other Theories

7.3.1 *Hehner's Practical Theory of Programming.* As shown in Table VI, commands in Hehner's [2004] theory stand for propositions that can be used directly in calculation without any special semantic theory. Here are some notes on the style of use.

The style is clearly propositional. The full language expresses *specifications*: p and q in Table VI may be any propositions. Flexibility is illustrated by $p \vee q$ expressing nondeterministic choice. A *program* is a specification where all parts are executable.

Program derivation or verification uses *refinement*: q is *refined* by p iff $q \Leftarrow p$. For instance, $\mathtt{x}' > \mathtt{x}$ is refined by `x := x + 1`, which stands for $\mathtt{x}' = \mathtt{x}+1 \wedge r' = r$, where r represents the other variables. Repetition is covered by taking

$$q \Leftarrow \texttt{while}\ b\ \texttt{do}\ p \quad \text{to stand for} \quad q \Leftarrow \texttt{if}\ b\ \texttt{then}\ (p\,;q).$$

A drawback of language constructs directly standing for propositions is that no other semantics can be attached, for example, termination. However, Hehner [1999] deems total correctness inappropriate and analyzes termination by inserting a time variable, as in

$$q \wedge t' < \infty \Leftarrow \texttt{if}\ b\ \texttt{then}\ (p\,;t := t+1\,;q).$$

Table VIII. Anteconditions in the Dijkstra Scholten Formalism and by Program Equations

Dijkstra Scholten Predicate Calculus	Formulation with Program Equations
Axioms for wla and wa	Definition: $\mathsf{wla}\,cp \equiv \forall s' \bullet \mathsf{R}_c\,(s, s') \Rightarrow p'$
Property: $\mathsf{wa}\,cp \equiv \mathsf{wla}\,cp \wedge \mathsf{wa}\,c\,1$	Definition: $\mathsf{wa}\,cp \equiv \mathsf{wla}\,cp \wedge \mathsf{T}_c\,s$
Definition: $\mathsf{LEM}_c \equiv [\mathsf{wa}\,c\,0 \equiv 0]$	Definition: $\mathsf{LEM}_c \equiv \forall \backslash s \bullet \mathsf{T}_c\,\backslash s \Rightarrow \exists s' \bullet R_c\,(\backslash s, s')$

This brief explanation captures the essence of the language in its full simplicity and generality. In many ways, the language is superior to any other (imperative) language, and arguably should be the basis for any introduction to imperative programming.

The many derivations in Section 6 show that calculational semantics is eminently suitable for reasoning about specifications and programs in Hehner's language.

7.3.2 *Dijkstra and Scholten's Predicate Calculus.* Section 6 shows how calculational semantics (re)discovers the crucial results from Dijkstra and Scholten [1990], in a few pages, with all nontrivial proofs. Some readers may even find this number of example calculations excessive, but as in Dijkstra and Scholten [1990], each derivation tells something extra beyond the mere result.

The epistemological importance of theory compression is discussed in Bass [2003]. Still, Section 6 focuses on program semantics whereas Dijkstra and Scholten [1990] additionally provides a theory on predicate transformers (Chapter 6) and extremal solutions to equations (Chapter 8). Together with the very readable style in which the book is written, this makes it required reading in any serious computing curriculum.

A basic difference with our approach is in the rationale. The prescriptive view (see Section 7.1) is very explicit in its aim in Dijkstra and Scholten [1990] to "relegate what used to be considered the subject matter to the secondary rôle of (ignorable) model." Yet, this rôle is not altogether ignorable, given the many operational elements covered by words in Dijkstra and Scholten [1990] when defining and reasoning about ap-conditions. This, as well as other considerations from Section 7.1, explains why we prefer the descriptive view, and to *derive* rather than *postulate*.

Resulting similarities and differences in the formulations are shown in Table VIII. Derived properties and ramifications have already been discussed in Section 6.1. Also important is that calculational semantics makes many arguments that are presented verbally in Dijkstra and Scholten [1990] amenable to formal calculation. When operational interpretations of formulas are desired, these are simple and direct via program equations.

7.3.3 *Hoare and Jifeng's Unifying Theories of Programming.* The objective of Hoare and Jifeng [1998] is similar to ours, namely the design of unifying theories. Here we point out some technical issues, using our notational conventions to bypass nonessential differences.

As in Hehner's theory, commands are propositions with the drawback previously mentioned. In Hoare and Jifeng [1998], Hoare-triples are defined by $\{a\}\ c\ \{p\} \equiv [c \Rightarrow \backslash a \Rightarrow p']$ (partial correctness), and "weakest preconditions" are

defined via composition as $\mathsf{wp}\, cp \equiv \neg (c\,;\neg p)$. The property $\mathsf{wp}\, c\, 1 \equiv 1$ is noted in Hoare and Jifeng [1998] as a discrepancy with Dijkstra [1976], but is easily explained in calculational semantics by observing that this wp corresponds to wla, not wa.

Repetition is studied by fixpoints. With the weakest fixpoint, a nonterminating command evaluates to 1. This is unwanted since, if 1 is nonterminating, then so are $1\,;c$ and $c\,;1$ (any c), hence $1\,;c \equiv 1$ and $c\,;1 \equiv 1$, which not all c satisfies.

With the strongest fixpoint, nontermination evaluates to 0, which is more acceptable, since it is a 2-sided zero for composition. Still, Hoare and Jifeng [1998] express concerns about $0 \sqsubseteq S$ which we hope are alleviated, perhaps eliminated, by the discussion in Section 7.1.

We conclude by noting that Hoare and Jifeng [1998] addresses this concern differently, namely, by characterizing commands c such that $1\,;c \equiv 1$ and $c\,;1 \equiv 1$ (hence clearly $c \not\equiv 0$). This is achieved by so-called *designs*. Let ok be a distinguished variable meant to indicate that a program has started, and ok' indicating that it has finished. A *design* is a (command) proposition of the form $c \vdash b$, standing for $ok \wedge c \Rightarrow ok' \wedge b$. By redefining assignments as designs, and observing that applying program combinators (choice, conditional, composition) to designs yields designs, programs become designs. So-called *healthiness conditions* ensure the required algebraic properties.

7.3.4 *R. Dijkstra's Computation Calculus.* In the spirit of reducing the formalization gap, R. Dijkstra proposed *computation calculus* [Dijkstra 1998], henceforth called CC.

We prefer CC over an earlier variant [Dijkstra 1994] because it needs no "infinity" state, but instead elegantly incorporates infinite computations into the definition of composition.

Here, we elaborate it as an example (a) to show how to apply calculational semantics to a theory whose formulas look quite different from those derived thus far; and (b) as a bonus, to calculate very appealing system equations for computations (in the CC sense).

As in Boute [1988, 2003], we define $\square n = \{m : \mathbb{N} \mid m < n\}$ for any $n : \mathbb{N}'$, where $\mathbb{N}' := \mathbb{N} \cup \iota\infty$. Further, $S \uparrow n$ (with shorthand S^n) is the set of sequences of length n over set S (formally: $S \uparrow n = \square n \to S$) for any $n : \mathbb{N}'$. Obviously, $S^0 = \iota\varepsilon$ and $S^\infty = \mathbb{N} \to S$. As expected, we also define $S^* = \bigcup n : \mathbb{N} \,.\, S^n$ and $S^+ = \bigcup n : \mathbb{N}_{>0} \,.\, S^n$ and $S^\omega = \bigcup n : \mathbb{N}' \,.\, S^n$.

An operator on sequences is *shift* (σ) with $\sigma\ x = n : \square(\#x - 1) \,.\, x\,(n+1)$ for nonempty sequence x. Redundant, but for the sake of symmetry we define *take* ($\rfloor$) and *drop* ($\lfloor$) by $f \rfloor n = x_{<n}$ and $f \lfloor n = \sigma^n x$ for any $x : S^\omega$ and $n : \square(\#x + 1)$. Useful properties are $\#(f \rfloor n) = n$ and $\#(f \lfloor n) = \#f - n$. Recall also that $\tau\, s = 0 \mapsto s$.

With these preliminaries, we present CC in computational semantics. Auxiliary results are just mentioned for later use; for the main result the calculation is shown.

Given state space $\mathbb{S}$, we define *computations* as elements of $\mathcal{C} := \mathbb{S}^+ \cup \mathbb{S}^\infty$. Specifications and behaviors are expressed by *computation predicates* of type $\mathsf{CP} := \mathcal{C} \to \mathbb{B}$. For later use, let $\mathtt{T} := \mathcal{C} \bullet 1$ en $\mathtt{F} := \mathcal{C} \bullet 0$. The central operator for this theory is *composition* $\_;\_ : \mathsf{CP}^2 \to \mathbb{B}$ with, for arbitrary $C', C'' : \mathsf{CP}^2$ and $\gamma : \mathcal{C}$,

$$(C';C'')\,\gamma \;\equiv\; (\#\gamma = \infty \wedge C'\,\gamma) \vee \exists n : \mathcal{D}\,\gamma \,.\, C'\,(\gamma \rfloor (n+1)) \wedge C''\,(\gamma \lfloor n).$$

By predicate calculus one can show that composition is associative and that the predicate $\mathbb{1} : \mathsf{CP}$ with $\mathbb{1}\gamma \equiv \#\gamma = 1$ is a 2-sided, and hence unique unit element. Convention: Composition has precedence over $\sqcap$ and $\sqcup$, hence $C \sqcap C';C'' = C \sqcap (C';C'')$.

States are represented by sequences of length 1 (satisfying $\mathbb{1}$), for example, γ_α with $\gamma_\alpha = \tau\,\gamma_0$ for initial states and γ_ω with $\gamma_\omega = \tau\,\gamma_{\#\gamma-1}$ if $\#\gamma \neq \infty$ for final states. *State predicates* are predicates of type $\mathsf{SP} := \{P : \mathsf{CP} \mid P \sqsubseteq \mathbb{1}\}$ (observing that $P \sqsubseteq \mathbb{1} \;\equiv\; P = P \sqcap \mathbb{1}$). Predicate calculus shows that (i) $(P\,;\mathtt{T})\,\gamma \;\equiv\; P\,\gamma_\alpha$, (ii) $(\mathtt{T};P)\,\gamma \;\equiv\; \#\gamma \neq \infty \Rightarrow P\,\gamma_\omega$, (iii) $P\,;C = P\,;\mathtt{T} \sqcap C$, and (iv) $C\,;P = C \sqcap \mathtt{T};P$ for any $P : \mathsf{SP}$ and $C : \mathsf{CP}$.

Defining the *eternity* predicate $\mathtt{E} := \mathtt{T};\mathtt{F}$ and the *bounded* predicate $\mathtt{B} := \overline{\neg}\ \mathtt{E}$, clearly, $\mathtt{E}\,\gamma \;\equiv\; \#\gamma = \infty$ and $\mathtt{B}\,\gamma \;\equiv\; \#\gamma \neq \infty$. Now, R. Dijkstra's definition of Hoare-triples amounts to the following: For any A en P in SP en C in CP,

$$\{A\}\,C\,\{P\} \;\equiv\; A\,;C \sqsubseteq \mathtt{T};P \tag{26}$$

$$[A]\,C\,[P] \;\equiv\; A\,;C \sqsubseteq \mathtt{B};P \tag{27}$$

A calculation example within this theory is deriving a formula expressing $[A]\,C\,[P]$ as the conjunction of $\{A\}\,C\,\{P\}$ and some termination formula (to be discovered).

$$\begin{array}{lll}
[A]\,C\,[P] & \equiv \langle \text{Def. triple (27)} \rangle & A\,;C \sqsubseteq \mathtt{B};P \\
& \equiv \langle \text{Prop. (iv) above} \rangle & A\,;C \sqsubseteq \mathtt{B} \sqcap \mathtt{T};P \\
& \equiv \langle \text{Left distr. } \sqcap/\sqsubseteq \rangle & A\,;C \sqsubseteq \mathtt{B} \;\wedge\; A\,;C \sqsubseteq \mathtt{T};P \\
& \equiv \langle \text{Def. triple (26)} \rangle & A\,;C \sqsubseteq \mathtt{B} \;\wedge\; \{A\}\,C\,\{P\}.
\end{array}$$

Hence, $[A]\,C\,[P] \equiv A\,;C \sqsubseteq \mathtt{B} \wedge \{A\}\,C\,\{P\}$; the termination formula is $A\,;C \sqsubseteq \mathtt{B}$.

A calculation example spanning across theories is the "reverse engineering" of *systems equations*, the abstract form of the earlier program equations, capturing CC. Specifically, we calculate $\mathsf{R}\_ : \mathsf{CP} \to \mathbb{S}^2 \to \mathbb{B}$ en $\mathsf{T}\_ : \mathsf{CP} \to \mathbb{S} \to \mathbb{B}$ to satisfy

$$\{A\}\,C\,\{P\} \;\equiv\; \forall (\text{\textbackslash}s, s') : \mathbb{S}^2 \,.\, \mathsf{R}_C\,(\text{\textbackslash}s, s') \Rightarrow A\,(\tau\,\text{\textbackslash}s) \Rightarrow P\,(\tau\,s') \tag{28}$$

$$A\,;C \sqsubseteq \mathtt{B} \;\equiv\; \forall \text{\textbackslash}s : \mathbb{S} \,.\, A\,(\tau\,\text{\textbackslash}s) \Rightarrow \mathsf{T}_C\,\text{\textbackslash}s. \tag{29}$$

These are variants of Definition 7.1, with sequences of length one replacing states. Calculating:

$$\begin{array}{rll}
A\,;C \sqsubseteq \mathtt{B} & \equiv \langle \text{Prop. (iii)} \rangle & A\,;\ \mathtt{T} \sqcap C \sqsubseteq \mathtt{B} \\
& \equiv \langle \text{Def. } \sqsubseteq \rangle & \forall\, \gamma : \mathcal{C} \,.\, (A\,;\ \mathtt{T} \sqcap C)\, \gamma \Rightarrow \mathtt{B}\, \gamma \\
& \equiv \langle \text{Def. } \sqcap \rangle & \forall\, \gamma : \mathcal{C} \,.\, (A\,;\ \mathtt{T})\, \gamma \wedge C\, \gamma \Rightarrow \mathtt{B}\, \gamma \\
& \equiv \langle \text{Prop. (i)} \rangle & \forall\, \gamma : \mathcal{C} \,.\, A\, \gamma_\alpha \wedge C\, \gamma \Rightarrow \mathtt{B}\, \gamma \\
& \equiv \langle \text{Shunt } \wedge \rangle & \forall\, \gamma : \mathcal{C} \,.\, A\, \gamma_\alpha \Rightarrow C\, \gamma \Rightarrow \mathtt{B}\, \gamma \\
& \equiv \langle \gamma_\alpha = \tau\, \gamma_0 \rangle & \forall\, \gamma : \mathcal{C} \,.\, A\, (\tau\, \gamma_0) \Rightarrow C\, \gamma \Rightarrow \mathtt{B}\, \gamma \\
& \equiv \langle \text{One-pt. rule} \rangle & \forall\, \gamma : \mathcal{C} \,.\, \forall\, \backslash s : \mathbb{S} \,.\, \backslash s = \gamma_0 \Rightarrow A\, (\tau\, \backslash s) \Rightarrow C\, \gamma \Rightarrow \mathtt{B}\, \gamma \\
& \equiv \langle \text{Shunt } \Rightarrow \rangle & \forall\, \gamma : \mathcal{C} \,.\, \forall\, \backslash s : \mathbb{S} \,.\, A\, (\tau\, \backslash s) \Rightarrow C\, \gamma \Rightarrow \backslash s = \gamma_0 \Rightarrow \mathtt{B}\, \gamma \\
& \equiv \langle \text{Swap } \forall \rangle & \forall\, \backslash s : \mathbb{S} \,.\, \forall\, \gamma : \mathcal{C} \,.\, A\, (\tau\, \backslash s) \Rightarrow C\, \gamma \Rightarrow \backslash s = \gamma_0 \Rightarrow \mathtt{B}\, \gamma \\
& \equiv \langle \text{Ldst. } \Rightarrow / \forall \rangle & \forall\, \backslash s : \mathbb{S} \,.\, A\, (\tau\, \backslash s) \Rightarrow \forall\, \gamma : \mathcal{C} \,.\, C\, \gamma \Rightarrow \backslash s = \gamma_0 \Rightarrow \mathtt{B}\, \gamma .
\end{array}$$

So, defining $\mathsf{T}_C\, \backslash s \ \equiv\ \forall\, \gamma : \mathcal{C} \,.\, C\, \gamma \Rightarrow \gamma_0 \ =\ \backslash s \Rightarrow \#\, \gamma \ \neq\ \infty$ satisfies (29). Similarly expanding $\{A\}\ C\ \{P\}$ yields $\forall\, \gamma : \mathcal{C} \,.\, C\, \gamma \wedge \mathtt{B}\, \gamma \Rightarrow A\, \gamma_\alpha \Rightarrow P\, \gamma_\omega$ halfway, and finally, $\forall\, (\backslash s, s') : \mathbb{S}^2 \,.\, \exists\, (\gamma : \mathcal{C} \,.\, C\, \gamma \wedge\ \mathtt{B}\, \gamma \wedge \backslash s = \gamma_0 \wedge s' = \gamma_{\#\, \gamma - 1}) \Rightarrow A\, (\tau\, \backslash s) \Rightarrow P\, (\tau\, s')$. Hence,

$$\begin{array}{rcl}
\mathsf{R}_C\, (\backslash s, s') & \equiv & \exists\, \gamma : \mathcal{C}_C \,.\, \gamma_0 = \backslash s \wedge \#\, \gamma \neq \infty \wedge \gamma_{\#\, \gamma - 1} = s' \\
\mathsf{T}_C\, \backslash s & \equiv & \forall\, \gamma : \mathcal{C}_C \,.\, \gamma_0 = \backslash s \Rightarrow \#\, \gamma \neq \infty
\end{array}$$

satisfies (28) and (29). Both equations have a very direct intuitive interpretation.

The weakest antecondition operator can be expressed as follows. Let $\bullet : \mathsf{SP} \to \mathsf{CP}$ with $\bullet P \ =\ P\,;\ \mathtt{T}$. Then $\mathcal{A} : \mathsf{CP} \to \mathsf{SP}$ and $\mathcal{E} : \mathsf{CP} \to \mathsf{SP}$ are defined by

$$\bullet P \sqsubseteq C \ \equiv\ P \sqsubseteq \mathcal{A}\, C \ \text{ and } \ \mathcal{E}\, C \sqsubseteq P \ \equiv\ C \sqsubseteq \bullet P .$$

Proving uniqueness is easy; existence is shown by the explicit equations

$$\mathcal{A}\, C\, \gamma \equiv\ \mathbb{1}\, \gamma \wedge \forall\, \gamma' : \mathcal{C} \,.\, \gamma'_0 = \gamma_0 \Rightarrow C\, \gamma' \ \text{ and } \ \mathcal{E}\, C\, \gamma \ \equiv\ \mathbb{1}\, \gamma \wedge \exists\, \gamma' : \mathcal{C} \,.\, \gamma'_0 = \gamma_0 \wedge C\, \gamma' .$$

We find $\mathsf{Wla}\, C\, P \ =\ \mathcal{A}(C \multimap \exists\ \mathtt{T}\,; P)$, $\mathsf{Term}_C \ =\ \mathcal{A}(C \multimap \exists\ \mathtt{B})$, $\mathsf{Wa}\, C\, P \ =\ \mathcal{A}(C \multimap \exists\ \mathtt{B}\,; P)$.

7.3.5 *Comparing Some Design Decisions About the Mathematical Formalism.* A few comments on formalism design may avoid confusion due to possibly subtle differences.

(a) Our formalism deviates less than most semantic theories from common conventions in applied mathematics. This bonus emerged *a posteriori*, since in the design suitability for expression and calculation, and freedom from ambiguities and from inconsistencies were paramount.

Common operators have their familiar meaning and properties. By contrast, in Dijkstra and Scholten [1990] operators inherently extend to structures in the sense that, with + as an example[8], $(x + y).n$ **=** $x.n$ **+** $y.n$. Even $x = y$ does not denote a Boolean value, but rather a function with $(x = y).n \equiv x.n$ **=** $y.n$. Expressions that seem like formulas in common mathematics, for example, $x = y$ and $X \wedge Y \Rightarrow X$, are Boolean structures. A Boolean structure X being

[8]We use boldface for "standard" $+, =$. The dot in function application is Dijkstra's notation.

true everywhere is written $[X]$. Most formulas have this form, for example, $[[X] \vee Y] \equiv [X \vee [Y]]$.

Note that still others [Hoare and Jifeng 1998] use [ ] for universal quantification over "all" (e.g., state) variables, yielding deceptively (yet seldom dangerously) similar-looking formulas.

For reasons explained in Boute [2003], we prefer the selectivity of an explicit extension operator such as $\widehat{\ }$ from Definition (3). Since in our formalism structures are functions, we write $f \mathrel{\widehat{=}} g$ for point-by-point equality, so $f = g \equiv \mathcal{D}\, f = \mathcal{D}\, g \wedge \forall (f \mathrel{\widehat{=}} g)$ expresses "equality everywhere" for functions.[9] As in common mathematics, we always let "=" denote "equality everywhere," in the spirit of Recorde's justification for writing it with two parallel lines *"bicause noe 2 thynges can be moare equalle"* [Recorde 1557].

(b) Despite the similarity between our conventions for the propositional style and Hehner's [2004] theory some design decisions are necessarily different. Indeed, Hehner considers a specific language, whereas our purpose is deriving theories for other languages as well, requiring more generality in fundamental and technical aspects.

As mentioned, simply identifying language constructs with propositions precludes attaching other semantics to constructs, which is useful for certain theories. It also makes variable bindings less uniform even for similar-looking expressions; for instance, the free occurrences of variables in $p \vee q$ are those from both p and q, but in $p\,;q$ some occurrences from p and q are hidden. This would be confusing in a general formalism.

Hehner uniquely uses the propositional style, handling "arguments" by name, not position. This is helpful for individual programs where names are the familiar items. Yet, in developing general theories, functions, and hence, predicates can be used as higher-order objects in a cleaner way than expressions or propositions. They also handle arguments by position, which is less biased (more abstract) than names.

As in Gries and Schneider [1993], we decided using both styles is appropriate: functions and relations for generality, expressions and propositions when matching the style of a given theory.

7.4 Epilogue: Functional Predicate Calculus in Analysis

A small example shows how the functional predicate calculus together with generic functionals contributes to unifying continuous and discrete mathematics. It presents a calculational solution to a problem from a well-known textbook on analysis [Lang 1983].

As in other areas, we found that formulating the relevant concepts via predicates is more elegant than via sets. Hence, we express *adjacency* by a predicate transformer.

$$\textbf{def}\ \mathsf{ad} : (\mathbb{R} \to \mathbb{B}) \to (\mathbb{R} \to \mathbb{B})\ \textbf{with}\ \mathsf{ad}\, P\, v \equiv \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists x : \mathbb{R}_P \,.\, |x - v| < \epsilon$$

[9]These f and g may contain free variables, for example, as first applications of higher-order functions.

The usual concepts of open and closed sets are expressed by predicates on predicates.

$$\begin{array}{l}\textbf{def}\ \mathsf{open} : (\mathbb{R}\to\mathbb{B})\to\mathbb{B}\ \textbf{with}\\ \quad \mathsf{open}\ P\ \equiv\ \forall v:\mathbb{R}_P\,.\,\exists\epsilon:\mathbb{R}_{>0}\,.\,\forall x:\mathbb{R}\,.\,|x-v|<\epsilon \Rightarrow P\,x\\ \textbf{def}\ \mathsf{closed} : (\mathbb{R}\to\mathbb{B})\to\mathbb{B}\ \textbf{with}\ \mathsf{closed}\ P\ \equiv\ \mathsf{open}\,(\overline{\neg}\,P)\end{array}$$

The problem selected from the exercises in Lang [1983] is proving the *closure property* $\mathsf{closed}\ P\ \equiv\ \mathsf{ad}\ P = P$. Here is the calculation (using some more rules from Boute [2002, 2006] and generic operators from Boute [2003])

$$\begin{array}{ll}\mathsf{closed}\ P & \\ \equiv\ \langle\text{Def. closed}\rangle & \mathsf{open}\,(\overline{\neg}\,P)\\ \equiv\ \langle\text{Def. open}\rangle & \forall v:\mathbb{R}_{\overline{\neg}\,P}\,.\,\exists\epsilon:\mathbb{R}_{>0}\,.\,\forall x:\mathbb{R}\,.\,|x-v|<\epsilon\Rightarrow\overline{\neg}\,P\,x\\ \equiv\ \langle\text{Trading sub }\forall\rangle & \forall v:\mathbb{R}\,.\,\overline{\neg}\,P\,v\Rightarrow\exists\epsilon:\mathbb{R}_{>0}\,.\,\forall x:\mathbb{R}\,.\,|x-v|<\epsilon\Rightarrow\overline{\neg}\,P\,x\\ \equiv\ \langle\text{Cntrps., twice}\rangle & \forall v:\mathbb{R}\,.\,\overline{\neg}\,\exists(\epsilon:\mathbb{R}_{>0}\,.\,\forall x:\mathbb{R}\,.\,P\,x\Rightarrow\neg(|x-v|<\epsilon))\Rightarrow P\,v\\ \equiv\ \langle\text{Duality, twice}\rangle & \forall v:\mathbb{R}\,.\,\forall(\epsilon:\mathbb{R}_{>0}\,.\,\exists x:\mathbb{R}\,.\,P\,x\wedge|x-v|<\epsilon)\Rightarrow P\,v\\ \equiv\ \langle\text{Def. ad}\rangle & \forall v:\mathbb{R}\,.\,\mathsf{ad}\,P\,v\Rightarrow P\,v\\ \equiv\ \langle P\,v\Rightarrow\mathsf{ad}\,P\,v\rangle & \forall v:\mathbb{R}\,.\,\mathsf{ad}\,P\,v\ \equiv\ P\,v\ \text{(proving } P\,v\Rightarrow\mathsf{ad}\,P\,v\text{ is near-trivial)}\end{array}$$

By "contrapositive" we mean $p\Rightarrow\neg\,q\ \equiv\ q\Rightarrow\neg\,p$. Observe the similarity between this calculation and the other calculations in this article, especially those like (7.1.3).

It is hardly exaggerated to say that program semantics and mathematical analysis are just applications of predicate calculus, in an even more direct and practical way than quantum mechanics is often said to reduce chemistry to mathematics.

The style breach between the formality of calculations with derivatives and integrals on one hand, and the informality of the underlying logical arguments as deplored in Taylor [2000] on the other hand, has been overcome. The result appears to be removed only a very short distance (if any) away from the realization of Leibniz's dream [Dijkstra 2000].

8. CONCLUSIONS

We have demonstrated how the calculational approach helps in deriving theories and in discovering or elucidating semantic issues where intuitive considerations are uncertain, misleading, or otherwise inadequate.

Program equations at the same time provide the simplest possible and intuitively most satisfying description of program behavior, and a suitable basis for calculating more abstract theories in a convenient way. In passing, we have shown how a well-documented inadequacy of purely relational formulations of program behavior can be overcome by simply adding a separate equation expressing guaranteed termination.

We have shown the relationship to some other theories and extensions, and justified the major design decisions.

All this was done in a formalism that is equally suitable for continuous and discrete mathematics. This makes acquiring proficiency in formal calculation with predicates, quantifiers, and generic functionals a worthwhile intellectual investment.

Finally, we remark that during the past decade, many interesting variants or alternatives for axiomatic semantics and program derivation have been proposed, such as those referenced along the way in this article. Sadly, these have not yet found their way into most computing curricula, although they are sufficiently mature for that purpose since they represent valuable insights resulting from decades of research.

Of all these, only a few were discussed in some detail here, but the results obtained thus far indicate that our approach can similarly incorporate others in the calculational unification as well, thereby lowering the threshold for their dissemination, further emphasizing common principles, and facilitating comparison.

ACKNOWLEDGMENTS

The author is grateful to Eric Hehner for the enlightening discussions on his approach to programming theory. He also wishes to thank the anonymous reviewers for many useful remarks that helped in improving this article and for pointers to other sources with very exciting material on related topics.

REFERENCES

BACK, R.-J. 1983. A continuous semantics for unbounded nondeterminism. *Theor. Comput. Sci. 23,* 2, 187–210.

BACK, R.-J. AND VON WRIGHT, J. 1992. Combining angels, demons and miracles in program specifications. *Theor. Comput. Sci. 100,* 2, 365–383.

BACK, R.-J. AND VON WRIGHT, J. 1998. *Refinement Calculus: A Systematic Introduction*. Springer, New York.

BACKHOUSE, R. 2002. *Galois Connections*. Number 7 in Programming Algebra. Univ. of Nottingham. `http://www.cs.nott.ac.uk/~rcb/G53PAL/G53PAL.html`.

BASS, H. 2003. The Carnegie initiative on the doctorate: The case of mathematics. *Notices of the AMS 50,* 7 (Aug.), 767–776.

BOITEN, E. AND MÖLLER, B. 2002. 6th international conference on mathematics of program construction. Conference announcement: `http://www.cs.kent.ac.uk/events/conf/2002/mpc2002`.

BOUTE, R. 1988. Systems semantics: Principles, applications and implementation. *ACM Trans. Program. Languages Syst. 10,* 1 (Jan.), 118–155.

BOUTE, R. 1993. Funmath illustrated: A declarative formalism and application examples. Declarative Systems Series 1, Computing Science Institute, University of Nijmegen.

BOUTE, R. 2002. Functional mathematics: A unifying declarative and calculational approach to systems, circuits and programs — Part I. Ghent University. Course notes.

BOUTE, R. 2003. Concrete generic functionals: Principles, design and applications. In *Generic Programming*, J. Gibbons and J. Jeuring, eds. Kluwer Academic, Hingham, Mass, 89–119.

BOUTE, R. 2005. Functional declarative language design and predicate calculus: A practical approach. *ACM Trans. Program. Languages Syst. 27*, 5 (Sept.) 988–1047.

COHEN, E. 1990. *Programming in the 1990's: An Introduction to the Calculation of Programs*. Springer, New York.

DE BAKKER, J. W. 1980. *Mathematical Theory of Program Correctness*. Prentice-Hall, Upper Saddle River, N. J.

DEAN, C. N. AND HINCHEY, M. G. 1996. *Teaching and Learning Formal Methods*. Academic Press, London.

DIJKSTRA, E. W. 1976. *A Discipline of Programming*. Prentice-Hall, Upper Saddle River, N. J.

DIJKSTRA, E. W. 2000. Under the spell of Lcibniz's dream. Technical Note EWD1298. `http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1298.pdf`.

DIJKSTRA, E. W. AND SCHOLTEN, C. S. 1990. *Predicate Calculus and Program Semantics*. Springer, New York.

DIJKSTRA, R. M. 1994. Relational calculus and relational program semantics. Computing Science Reports CS-R9408, Dept. of Computer Science, University of Groningen.

DIJKSTRA, R. M. 1998. Computation calculus: Bridging a formalization gap. In *Proceedings of the Conference Mathematics of Program Construction*. LNCS, vol. 1422. Springer, New York, 151–174.

GORDON, M. 2003. *Specification and Verification I*. University of Cambridge. http://www.cl.cam.ac.uk/Teaching/mjcg/Lectures/SpecVer1/Notes03/Notes.pdf.

GRIES, D. 1996. The need for education in useful formal logic. *IEEE Computer 29,* 4 (Apr.), 29–30.

GRIES, D. AND SCHNEIDER, F. B. 1993. *A Logical Approach to Discrete Math*. Springer, New York.

HANCOCK, P. 2004. Refinement calculus: Some references and pointers. Technical note. http://homepages.inf.ed.ac.uk/v1phanc1/RC-bib.pdf.

HEHNER, E. 1999. Specifications, programs and total correctness. *Sci. Comput. Program. 34,* 3 (July), 191–205. Original version (1998) also on the web: http://www.cs.toronto.edu/~hehner/SPTC.pdf.

HEHNER, E. 2004. *A Practical Theory of Programming*, 2nd ed. Springer, New York. http://www.cs.toronto.edu/~hehner/aPToP/.

HESSELINK, W. H. 1992. *Programs, Recursion, and Unbounded Choice*. Cambridge, New York.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Comm. ACM 12,* 10 (Oct.), 576–580, 583.

HOARE, C. A. R. AND JIFENG, H. 1998. *Unifying Theories of Programming*. Prentice-Hall, Upper Saddle River, N. J.

LAMPORT, L. 2002. *Specifying Systems*. Addison-Wesley, Reading, Mass.

LAMPORT, L. 2004. All I really need to know I learned in high school. *Proceedings of the 2004 CoLogNET/FME Symposium on Teaching Formal Methods*. http://www.intec.UGent.be/groupsites/formal/Sympos2004/Sympos2004.htm.

LANG, S. 1983. *Undergraduate Analysis*. Springer, Berlin.

LEAVENS, G. 1995. Weakest preconditions. Course notes *Semantics Program. Languages (Com S 641)*. http://www.cs.iastate.edu/~leavens/ComS641-Hesselink.html.

LEE, E. A. AND MESSERSCHMITT, D. G. 1998. Engineering—An education for the future. *IEEE Computer 31,* 1 (Jan.), 77–85. http://www.gigascale.org/pubs/5/computermag.pdf.

LEE, E. A. AND VARAIYA, P. 2003. *Structure and Interpretation of Signals and Systems*. Addison-Wesley, Reading, Mass.

LEINO, K. R. M. AND MANOHAR, R. 1999. Joining specification statements. *Theor. Comput. Sci. 216,* 1–2 (Mar.), 375–394.

LOECKX, J. AND SIEBER, K. 1984. *The Foundations of Program Verification*. Wiley-Teubner.

MCCLUSKEY, E. J. 1965. *Introduction to the Theory of Switching Circuits*. McGraw Hill, New York.

MEYER, B. 1991. *Introduction to the Theory of Programming Languages*. Prentice Hall, Upper Saddle River, N. J.

MORGAN, C. 1994. *Programming from Specifications*, 2nd ed. Prentice Hall, Upper Saddle River, N. J.

MORRIS, J. M. 1987. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program. 9,* 3 (Dec.), 287–306.

NELSON, G. 1989. A generalization of Dijkstra's calculus. *ACM Trans. Prog. Lang. Syst. 11,* 4 (Oct.), 517–561.

PLOTKIN, G. D. 1980. Dijkstra's predicate transformers and Smyth's powerdomains. In *Abstract Software Specifications*, D. Björner, ed. LNCS, vol. 86. Springer, New York, 527–583.

RAVAGLIA, R., ALPER, T., ROZENFELD, M., AND SUPPES, P. 1999. Successful pedagogical applications of symbolic computation. In *Computer-Human Interaction in Symbolic Computation*, N. Kajler, ed. Springer, New York. http://www-epgy.stanford.edu/research/chapter4.pdf.

RECORDE, R. 1557. *The Whetstone of Witte*. http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Recorde.html.

TAYLOR, P. 2000. *Practical Foundations of Mathematics*, 2nd printing. Cambridge Studies in Advanced Mathematics, no. 59. Cambridge University Press, New York. Comment about chapter 1 of this book on http://www.dcs.qmul.ac.uk/~pt/Practical_Foundations/html/s10.html.

WIGNER, E. 1960. The unreasonable effectiveness of mathematics in the natural sciences. *Comm. Pure Appl. Math. 13,* I (Feb.), 1–14. http://nedwww.ipac.caltech.edu/level5/March02/Wigner/Wigner.html.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, Mass.

Received May 2004; revised January 2005; accepted February 2005