

# SIGNAL PROCESSING FUNCTIONS, ALGORITHMS AND SMURFS: THE NEED FOR DECLARATIVITY

*Raymond Boute*

boute@intec.UGent.be

INTEC, Universiteit Gent, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium

## ABSTRACT

The gap between modelling techniques for DSP functionality and those for software implementations is widening. This impedes unifying formalisms for analog, digital and software systems. Recovering these opportunities requires declarativity.

A suitable formalism is outlined, based on a mathematical rather than a programming language. Examples show how it unifies continuous and discrete mathematics, from analysis to formal program semantics. The formalism provides crucial advantages in reasoning by calculation about all aspects of SP, and paves the way for software tools of the next generation.

## 1 INTRODUCTION

### 1.1 Motivation: the lost grail of declarativity

There is a widening gap between the modelling of SP functionality and of software implementation. The first typically uses well-established formalisms from classical engineering mathematics, but software practices around DSP systems resemble the early days of programming by not using mathematical models for programs or for calculationally deriving programs and their properties [25].

The main cause in DSP is a shift from essentially declarative mature engineering formalisms to “algorithmic thinking” induced by computer implementation, ignoring the declarative mathematical methods for software. This evolution is historically and educationally backward: mathematics evolved from algorithmic concerns (adding numbers) to declarativity (geometry, algebra, analysis), and so does education from grade school to university.

A well-designed declarative DSP language like Silage [18], although now superseded by new concepts, was “a significant improvement over most of its successors”, such as C++ (and SystemC), about which Lampert [21] aptly warns that it may harm the ability to think logically.

Another symptom is very sloppy terminology, blurring the difference between an “algorithm” and an abstract system characteristic to a degree that compares unfavorably with Peyo’s little blue dwarfs calling everything “smurf”.

Indeed, reducing the abstraction level to so-called “executable specifications” wastes valuable opportunities: no programming notation, but only a mathematical one, can achieve the required declarativity [21], especially in DSP.

### 1.2 Encouraging developments

Encouraging is the growth of hybrid systems formalisms [2, 13, 28], although their style is often too entrenched in traditional logic for linking conveniently to classical engineering mathematics. Here the more practical *calculational* logic advocated in [14, 16, 8] is better suited.

The need for declarativity has also been emphasized by eminent researchers in the DSP area [23], in the context of making Electrical Engineering and Computer Engineering into a more unified discipline, called ECE [22].

Our own research over the past 15 years is also aimed at unifying EE and CS, starting with mathematical modelling and reasoning. The style in classical engineering is mostly *calculational*: chaining expressions by relational operators, e.g., equality. In a classic engineering text [12],

$$\begin{aligned} F(s) &= \int_{-\infty}^{+\infty} e^{-|x|} e^{-i2\pi xs} dx \\ &= 2 \int_0^{+\infty} e^{-x} \cos 2\pi xs dx \\ &= 2 \operatorname{Re} \int_0^{+\infty} e^{-x} e^{i2\pi xs} dx \\ &= 2 \operatorname{Re} \frac{-1}{i2\pi s - 1} \\ &= \frac{2}{4\pi^2 s^2 + 1}, \end{aligned} \tag{1}$$

every step is based on a clear calculation rule.

By contrast, logical reasoning in everyday practice by mathematicians and engineers is highly informal, and often involves what Taylor [27] calls *syncopation*, namely using symbols as mere abbreviations of natural language, for instance the quantifier symbols  $\forall$  and  $\exists$  just meaning “for all” and “there exists”, without calculation rules.

We provide a formalism enabling engineers to calculate with quantifiers as fluently as with simple derivatives and integrals. The formalism is free of all defects of common conventions, also those outlined by Lee and Varaiya [23].

The reward is that mathematical models for software become as convenient as those for signals and systems.

### 1.3 Overview

The language and the calculation rules of the formalism are presented in section 2, Application examples are given in section 3, conclusions in section 4.

## 2 THE FORMALISM

### 2.1 Funmath language design

Poor notation is a stumbling block against formal calculation: if one has to be on guard for the defects, one cannot “let the symbols do the work”. For a critique of typical defects in conventions “everyone” uses, see [10, 23].

We do not patch defects ad hoc, but generate correct forms by orthogonal combination of just 4 constructs, gaining extra useful forms of expression. The basis is *functional*. A *function*  $f$  is fully defined by its *domain*  $\mathcal{D}f$  and its *mapping* (image definition). Here are the constructs.

**Identifiers:** nearly any symbol. They are *introduced* by *bindings*  $i: X \wedge p$ , where  $i$  is the (tuple of) identifier(s),  $X$  a set and  $p$  a proposition. The *filter*  $\wedge p$  (or **with**  $p$ ) is optional, e.g.,  $n: \mathbb{N}$  and  $n: \mathbb{Z} \wedge n \geq 0$  are interchangeable.

Shorthand:  $i := e$  stands for  $i: \iota e$ . We write  $\iota e$ , not  $\{e\}$ , for singleton sets, using  $\iota$  defined by  $e' \in \iota e \equiv e' = e$ .

Identifiers can be *variables* (in an *abstraction* as below) or *constants* (introduced by a *definition*: **def binding**).

**Application:** the default is  $f e$  for function  $f$  and argument  $e$ ; other conventions may be specified binding, e.g.,  $—\star—$  for infix. Parentheses are *never* operators, but only indicate parsing. Precedence are the usual ones. If  $f$  is a function-valued function,  $f x y$  stands for  $(f x) y$ .

*Partial application* of  $\star$  is  $a\star$  or  $\star b$ , with  $(a\star)b = a\star b = (\star b)a$ . *Variadic application* is  $a\star b\star c$  etc., always defined as  $F(a, b, c)$  for a suitable *elastic extension*  $F$  of  $\star$ .

**Abstraction:** the form is  $b.e$ , where  $b$  is a binding,  $e$  an expression;  $v: X \wedge p.e$  denotes a function whose domain is the set of  $v$  in  $X$  satisfying  $p$ , and mapping  $v$  to  $e$ . Syntactic sugar:  $e | b$  for  $b.e$  and  $v: X | p$  for  $v: X \wedge p.v$ .

A trivial example: if  $v$  does not occur (free) in  $e$ , we define  $\bullet$  by  $X \bullet e = v: X.e$  to denote *constant functions*. Special cases: the *empty function*  $\varepsilon := \emptyset \bullet e$  (any  $e$ ) and defining  $\mapsto$  by  $e' \mapsto e = \iota e' \bullet e$  for *one-point functions*.

We use abstractions in synthesizing familiar expressions such as  $\sum i: 0..n. q^i$  and  $\{m: \mathbb{Z} | m < n\}$ .

**Tupling:** the basic form is  $e, e', e''$  (any length), denoting a function with domain axiom  $\mathcal{D}(e, e', e'') = \{0, 1, 2\}$  and mapping axiom  $(e, e', e'') 0 = e$  and  $(e, e', e'') 1 = e'$  and  $(e, e', e'') 2 = e''$ . The empty tuple is  $\varepsilon$  and for singleton tuples we define  $\tau$  with  $\tau e = 0 \mapsto e$ . Parentheses are *not* part of tupling, and matrices are 2-dimensional tuples.

### 2.2 Equational and calculational reasoning

The equational style of (1) is generalized to the format

$$e \ R \langle \text{Justification} \rangle \ e' \quad (2)$$

where the  $R$  in successive lines are mutually transitive, for instance  $=, \leq$ , etc. in arithmetic,  $\equiv, \Rightarrow$  etc. in logic.

We write  $[\frac{v}{e}]$  for substitution, as in  $(x \cdot y)_{z+1}^x = (z+1) \cdot y$ .

### 2.3 Rules for calculating with propositions and sets

Assume the usual propositional operators  $\neg, \equiv, \Rightarrow, \wedge, \vee$ . A practical proposition calculus needs many rules [16]. Implication  $\equiv$  is associative,  $\Rightarrow$  is not. Parentheses in  $p \Rightarrow (q \Rightarrow r)$  are optional, so required in  $(p \Rightarrow q) \Rightarrow r$ . Embedded in arithmetic [4, 5], logic constants are 0, 1.

*Leibniz's principle* is  $e = e' \Rightarrow d[\frac{v}{e}] = d[\frac{v}{e'}]$ .

For sets,  $\in$  is the basis. Rules are derived ones, e.g., defining  $\cap$  by  $x \in X \cap Y \equiv x \in X \wedge x \in Y$  and  $\times$  by  $(x, y) \in X \times Y \equiv x \in X \wedge y \in Y$ . later we define  $\{—\}$ , enabling to prove  $y \in \{x: X | p\} \equiv y \in X \wedge p[\frac{x}{y}]$ .

### 2.4 Rules for functions and generic functionals

We omit the design decisions, to be found in [6] and [9]. In what follows,  $f$  and  $g$  are any functions,  $P$  any predicate ( $\mathbb{B}$ -valued function,  $\mathbb{B} := \{0, 1\}$ ),  $X$  any set,  $e$  arbitrary.

**Function equality and abstraction** *Equality* is defined by  $f = g \Rightarrow \mathcal{D}f = \mathcal{D}g \wedge (x \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow f x = g x)$  (Leibniz) and by *extensionality* for the converse.

Abstraction encapsulates substitution. Formally: *domain axiom*  $d \in \mathcal{D}(v: X \wedge p.e) \equiv d \in X \wedge p[\frac{v}{d}]$ ; *mapping axiom*  $d \in \mathcal{D}(v: X \wedge p.e) \Rightarrow (v: X \wedge p.e) d = e[\frac{v}{d}]$ . Equality is characterized via function equality (exercise).

**Generic functionals** Goals: (a) removing restrictions in common mathematical functionals, (b) making often-used implicit functionals from signal and systems theory explicit. The idea is defining the result domain judiciously.

Case (a) is illustrated by composition  $f \circ g$ , commonly requiring  $\mathcal{R}g \subseteq \mathcal{D}f$ . We define, for *any* functions:

$$f \circ g = x: \mathcal{D}g \wedge g x \in \mathcal{D}f. f(g x). \quad (3)$$

Note:  $\mathcal{D}(f \circ g) = \{x: \mathcal{D}g | g x \in \mathcal{D}f\}$ .

Case (b) is illustrated by the usual implicit generalization of arithmetic functions to signals, traditionally written  $(s + s')(t) = s(t) + s'(t)$ . We generalize this by (*duplex direct extension*  $\widehat{\ } \wedge$ ): for any functions  $\star$  (infix),  $f, g$ ,

$$f \widehat{\star} g = x: \mathcal{D}f \cap \mathcal{D}g \wedge (f x, g x) \in \mathcal{D}(\star). f x \star g x. \quad (4)$$

Similar is *half direct extension*: for function  $f$ , any  $e$ ,

$$f \overleftarrow{\star} e = f \widehat{\star} (\mathcal{D}f \bullet e) \quad \text{and} \quad e \overrightarrow{\star} f = (\mathcal{D}f \bullet e) \widehat{\star} f. \quad (5)$$

*Simplex direct extension*  $\overline{\ } \wedge$  is defined by  $\overline{f} g = f \circ g$ .

*Filtering* ( $\downarrow$ ) introduces or eliminates arguments:

$$f \downarrow P = x: \mathcal{D}f \cap \mathcal{D}P \wedge P x. f x. \quad (6)$$

We extend  $\downarrow$  to sets:  $x \in (X \downarrow P) \equiv x \in X \cap \mathcal{D}P \wedge P x$ . Writing  $a_b$  for  $a \downarrow b$  and using partial application, we get formal rules for useful shorthands like  $f_{<n}$  and  $\mathbb{R}_{>0}$ .

For the common *restriction* ( $\downarrow$ ):  $f \downarrow X = f \downarrow (X \bullet 1)$ .

A very important use of generic functionals is supporting the *point-free* style, i.e., without referring to domain points. The elegant algebraic flavor is illustrated next.

## 2.5 Rules for predicate calculus and quantifiers

**Axioms, forms of expression** For any predicate  $P$ ,

$$\forall P \equiv P = \mathcal{D}P \bullet 1 \quad \exists P \equiv P \neq \mathcal{D}P \bullet 0 \quad (7)$$

Letting  $P$  be an abstraction  $v : X . p$  yields the familiar form  $\forall v : X . p$ , as in  $\forall x : \mathbb{R} . x^2 \geq 0$ . Algebraic laws are most elegantly stated in point-free form. Each has a point-wise (familiar-looking) form using an abstraction.

**Derived rules** All follow from (7) and function equality. A practical collection is derived in [8, 10]. Here we give only some examples, starting by expressing  $f = g$  as

$$f = g \equiv \mathcal{D}f = \mathcal{D}g \wedge \forall x : \mathcal{D}f \cap \mathcal{D}g . f x = g x \quad (8)$$

Another example is *duality* (generalizing De Morgan law)

$$\neg \forall P = \exists (\neg P) \quad \neg (\forall v : X . p) \equiv \exists v : X . \neg p \quad (9)$$

Here are the main distributivity laws. All have duals.

Rule name	Point-free form
Distributiv. $\forall/\forall$	$q \vee \forall P \equiv \forall (q \overline{\vee} P)$
L(eft)-dist. $\Rightarrow/\forall$	$q \Rightarrow \forall P \equiv \forall (q \overline{\Rightarrow} P)$
R(ight)-dst. $\Rightarrow/\exists$	$\exists P \Rightarrow q \equiv \forall (P \overline{\Rightarrow} q)$
P(seudo)-d. $\wedge/\forall$	$\mathcal{D}P = \emptyset \vee (p \wedge \forall P) \equiv \forall (p \overline{\wedge} P)$

Pointwise:  $\exists (v : X . p) \Rightarrow q \equiv \forall (v : X . p \Rightarrow q)$  (new  $v$ ).

Here are a few additional laws, without comment.

Name of the rule	Point-free form
Distribut. $\forall/\wedge$	$\forall (P \overline{\wedge} Q) \equiv \forall P \wedge \forall Q$
One-point rule	$\forall P_{=e} \equiv e \in \mathcal{D}P \Rightarrow P e$
Trading	$\forall P_Q \equiv \forall (Q \overline{\Rightarrow} P)$
Transposition	$\forall (\forall \circ R) \equiv \forall (\forall \circ R^T)$

## 2.6 Wrapping up the rule package for function(al)s

**Function range** We define the range operator  $\mathcal{R}$  by

$$e \in \mathcal{R}f \equiv \exists x : \mathcal{D}f . f x = e. \quad (10)$$

A consequence is the composition rule  $\forall P \Rightarrow \forall (P \circ f)$  and  $\mathcal{D}P \subseteq \mathcal{R}f \Rightarrow (\forall (P \circ f) \equiv \forall P)$ ; in pointwise form  $\forall (y : \mathcal{R}f . p) \equiv \forall (x : \mathcal{D}f . p|_{f x}^y)$  (“dummy change”).

**Set comprehension** We define  $\{—\}$  as *fully interchangeable* with  $\mathcal{R}$ . This yields defect-free set notation: expressions like  $\{2, 3, 5\}$  and  $Even = \{2 \cdot m \mid m : \mathbb{Z}\}$  have familiar form and meaning, and all desired calculation rules follow from predicate calculus via Eq. (10). In particular, we can prove  $e \in \{v : X \mid p\} \equiv e \in X \wedge p|_e^e$  (exercise).

**Function typing** The familiar *function arrow* ( $\rightarrow$ ) is defined by  $f \in X \rightarrow Y \equiv \mathcal{D}f = X \wedge \mathcal{R}f \subseteq Y$ . A more refined type is the *Functional Cartesian Product* ( $\times$ ):

$$f \in \times T \equiv \mathcal{D}f = \mathcal{D}T \wedge \forall x : \mathcal{D}f . f x \in T x, \quad (11)$$

where  $T$  is a set-valued function. Note  $\times (X, Y) = X \times Y$  and  $\times (X \bullet Y) = X \rightarrow Y$ . We use  $X \ni x \rightarrow Y$  as shorthand for  $\times x : X . Y$ , where  $Y$  may depend on  $x$ .

## 3 EXAMPLES

### 3.1 Examples in analysis and continuous systems

**Analysis: calculation replacing syncopation** We show how traditional proofs that are tedious by syncopation [27] are done calculationally. The example is *adjacency* [20]. Since predicates (of type  $\mathbb{R} \rightarrow \mathbb{B}$ ) yield more elegant formulations than sets (of type  $\mathcal{P} \mathbb{R}$ ), we define the predicate transformer  $\mathbf{ad} : (\mathbb{R} \rightarrow \mathbb{B}) \rightarrow (\mathbb{R} \rightarrow \mathbb{B})$  and the predicates  $\mathbf{open} : (\mathbb{R} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$  and  $\mathbf{closed} : (\mathbb{R} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$  with

$$\begin{aligned} \mathbf{ad} P v &\equiv \forall \epsilon : \mathbb{R}_{>0} . \exists x : \mathbb{R}_P . |x - v| < \epsilon \\ \mathbf{open} P &\equiv \forall v : \mathbb{R}_P . \exists \epsilon : \mathbb{R}_{>0} . \forall x : \mathbb{R} . |x - v| < \epsilon \Rightarrow P x \\ \mathbf{closed} P &\equiv \mathbf{open} (\neg P) . \end{aligned}$$

We prove the *closure property*  $\mathbf{closed} P \equiv \mathbf{ad} P = P$ . The calculation, after the (easy) lemma  $P v \Rightarrow \mathbf{ad} P v$ , is

$$\begin{aligned} &\mathbf{closed} P \\ &\equiv \langle \mathbf{closed} \rangle \mathbf{open} (\neg P) \\ &\equiv \langle \mathbf{open} \rangle \forall v : \mathbb{R}_P . \exists \epsilon : \mathbb{R}_{>0} . \forall x : \mathbb{R} . |x - v| < \epsilon \Rightarrow \neg P x \\ &\equiv \langle \text{Trading } \forall \rangle \\ &\quad \forall v : \mathbb{R} . \neg P v \Rightarrow \exists \epsilon : \mathbb{R}_{>0} . \forall x : \mathbb{R} . |x - v| < \epsilon \Rightarrow \neg P x \\ &\equiv \langle \text{Contrapositive, i.e., } \neg p \Rightarrow q \equiv \neg q \Rightarrow p \rangle \\ &\quad \forall v : \mathbb{R} . \neg \exists (\epsilon : \mathbb{R}_{>0} . \forall x : \mathbb{R} . P x \Rightarrow \neg (|x - v| < \epsilon)) \Rightarrow P v \\ &\equiv \langle \text{Duality and } \neg (p \Rightarrow \neg q) \equiv p \wedge q \rangle \\ &\quad \forall v : \mathbb{R} . \forall (\epsilon : \mathbb{R}_{>0} . \exists x : \mathbb{R} . P x \wedge |x - v| < \epsilon) \Rightarrow P v \\ &\equiv \langle \text{Def. ad} \rangle \forall v : \mathbb{R} . \mathbf{ad} P v \Rightarrow P v \\ &\equiv \langle \text{Lemma} \rangle \forall v : \mathbb{R} . \mathbf{ad} P v \equiv P v . \end{aligned}$$

**Properties of systems** Signals over  $A$  are functions of type  $\mathcal{S}_A := \mathbb{T} \rightarrow A$  for time domain  $\mathbb{T}$ . A system is a function  $s : \mathcal{S}_A \rightarrow \mathcal{S}_B$ . Assume  $\mathbb{T}$  additive and  $A = B = \mathbb{C}$ .

Let  $\sigma$  be the *shift* operator with  $\sigma_\tau x t = x (t + \tau)$ . We characterize *time invariance* by  $\forall \tau : \mathbb{T} . s \circ \sigma_\tau = \sigma_\tau \circ s$  and *linearity* by  $\forall z : \mathcal{S}_\mathbb{C} . \forall c : \mathbb{C} . s (c \overline{\cdot} z) = c \overline{\cdot} s z$ .

We show that the response of a linear and time-invariant system to the parametrized exponential  $\mathbf{E}_- : \mathbb{C} \rightarrow \mathbb{T} \rightarrow \mathbb{C}$  with  $\mathbf{E}_c t = e^{c \cdot t}$  satisfies  $s \mathbf{E}_c = s \mathbf{E}_c \overline{\cdot} \mathbf{E}_c$ . We start by calculating  $s \mathbf{E}_c (t + \tau)$  in order to exploit all properties:

$$\begin{aligned} s \mathbf{E}_c (t + \tau) &= \langle \text{Definition } \sigma \rangle \sigma_\tau (s \mathbf{E}_c) t \\ &= \langle \text{Time inv. } s \rangle s (\sigma_\tau \mathbf{E}_c) t \\ &= \langle \text{Property } \mathbf{E}_c \rangle s (\mathbf{E}_c \tau \overline{\cdot} \mathbf{E}_c) t \\ &= \langle \text{Linearity } s \rangle (\mathbf{E}_c \tau \overline{\cdot} s \mathbf{E}_c) t \\ &= \langle \text{Defintion } \overline{\cdot} \rangle \mathbf{E}_c \tau \cdot s \mathbf{E}_c t . \end{aligned}$$

Substituting  $t := 0$  yields  $s \mathbf{E}_c \tau = (s \mathbf{E}_c \overline{\cdot} \mathbf{E}_c) \tau$  and hence, by function equality,  $s \mathbf{E}_c = s \mathbf{E}_c \overline{\cdot} \mathbf{E}_c$ .

### 3.2 Functions as a unifying paradigm in SP

Sequences are rarely viewed consistently as functions in DSP, which often even leads to inadequate conventions pointed out in [23], such as denoting a sequence by  $x[n]$ .

Continuous and discrete SP can be unified by always defining sequences as functions, making them inherit the rich collection of generic functionals. This issue is discussed in [9], including application examples to formal semantics and calculational reasoning for SP-related languages such as LabVIEW. Here we provide only the basics and a somewhat different example about automata.

**Sequences as functions** Define  $\square n = \{m : \mathbb{N} \mid m < n\}$  for any  $n : \mathbb{N}'$ , where  $\mathbb{N}' := \mathbb{N} \cup \iota \infty$ . The set of sequences of length  $n$  over a set  $A$  is defined by  $A \uparrow n = \square n \rightarrow A$ , with shorthand  $A^n$ . Note:  $A^0 = \iota \varepsilon$  and  $A^\infty = \mathbb{N} \rightarrow A$ . Also,  $A^* = \bigcup n : \mathbb{N}. A^n$  and  $A^+ = \bigcup n : \mathbb{N}_{>0}. A^n$  and  $A^\omega = \bigcup n : \mathbb{N}'. A^n$ . Finally, recall  $\tau a = 0 \mapsto a$ .

We define the *shift* ( $\sigma$ ) for any nonempty sequence  $x$  by  $\sigma x = n : \square (\#x - 1). x (n + 1)$ . Concatenation is  $++$ , e.g.,  $(7, e) ++ (3, d) = 7, e, 3, d$ , and  $x \prec a = x ++ \tau a$ .

**Causal systems** We define *prefix ordering*  $\leq$  on  $A^*$  by  $x \leq y \equiv \exists z : A^*. y = x ++ z$ . A systems  $s : A^* \rightarrow B^*$  is *sequential* iff  $x \leq y \Rightarrow s x \leq s y$ . This captures the notion of causal (better: “non-anticipatory”) behavior.

Function  $r : (A^*)^2 \rightarrow B^*$  is a *residual behavior* (rb) function for  $s$  iff  $s (x ++ y) = s x ++ r (x, y)$ .

**THEOREM:**  $s$  is sequential iff it has an rb function.

**Proof:** starting from the sequentiality side,

$$\begin{aligned} & \forall (x, y) : (A^*)^2. x \leq y \Rightarrow s x \leq s y \\ & \equiv (\text{Definit. } \leq) \forall (x, y) : (A^*)^2. \exists (z : A^*. y = x ++ z) \Rightarrow \\ & \quad \exists (u : B^*. s y = s x ++ u) \\ & \equiv (\text{Rdst } \Rightarrow / \exists) \forall (x, y) : (A^*)^2. \forall (z : A^*. y = x ++ z) \Rightarrow \\ & \quad \exists (u : B^*. s y = s x ++ u) \\ & \equiv (\text{Nest, swp}) \forall x : A^*. \forall z : A^*. \forall (y : A^*. y = x ++ z) \Rightarrow \\ & \quad \exists (u : B^*. s y = s x ++ u) \\ & \equiv (1\text{-pt, nest}) \forall (x, z) : (A^*)^2. \exists (u : B^*. s (x ++ z) = s x ++ u) \\ & \equiv (\text{Compreh.}) \exists r : (A^*)^2 \rightarrow B^*. \\ & \quad \forall (x, z) : (A^*)^2. s (x ++ z) = s x ++ r (x, z) . \end{aligned}$$

We used the *function comprehension* axiom: given any relation  $R : X \times Y \rightarrow \mathbb{B}$ , then  $\forall x : X. \exists y : Y. R (x, y)$  iff  $\exists f : X \rightarrow Y. \forall x : X. R (x, f x)$ .

**Application: derivatives and primitives** For sequential systems, we define the *derivative* operator  $D$  by  $D s \varepsilon = \varepsilon$  and  $s (x \prec a) = s x ++ D s (x \prec a)$ , so  $D s (x \prec a) = r (x, \tau a)$  where  $r$  is the (unique!) rb function of  $s$ .

*Primitivation*  $I$  is defined for any  $g : A^* \rightarrow B^*$  by  $I g \varepsilon = \varepsilon$  and  $I g (x \prec a) = I g x ++ g (x ++ a)$ . Properties are shown next, with a striking analogy in analysis (with respective  $D$  and  $I$ -operators, of course).

$s (x \prec a) = s x ++ D s (x \prec a)$	$s x = s \varepsilon ++ I (D s) x$
$f (x + h) = f x + D f x \cdot h$	$f x = f 0 + I (D f) x$

Finally,  $\{(y : A^*. r (x, y)) \mid x : A^*\}$  is the *state space*, on which automata realizing  $s$  can be defined (exercise).

### 3.3 Modelling programs

**Program equations** Define the *state*  $s$  as the tuple made of the program variables, and  $\mathbf{S}$  its type. We let  $\backslash s$  denote the state before and  $s'$  after executing a command;  $\backslash e = e[\backslash s]$  and  $e' = e[s']$ ; also,  $s \bullet e$  abbreviates  $s : \mathbf{S}. e$ . Let  $C$  be the set of commands.

We define  $R_- : C \rightarrow \mathbf{S}^2 \rightarrow \mathbb{B}$  and  $T_- : C \rightarrow \mathbf{S} \rightarrow \mathbb{B}$  such that the effect of command  $c$  is described by two equations:  $R_c (\backslash s, s')$  for state change,  $T_c \backslash s$  for termination. Example: for Dijkstra’s *guarded command* language [14],

Command $c$	State change $R_c (\backslash s, s')$
$v := e$ $c'; c''$ if $\square i : I. b_i \rightarrow c'_i$ fi	$s' = \backslash s \overset{v}{\leftarrow} e$ $\exists s \bullet R_{c'} (\backslash s, s) \wedge R_{c''} (\backslash s, s')$ $\exists i : I. \backslash b_i \wedge R_{c'_i} (\backslash s, s')$
Command $c$	Termination $T_c \backslash s$
$v := e$ $c'; c''$ if $\square i : I. b_i \rightarrow c'_i$ fi	1 $T_{c'} \backslash s \wedge \forall s \bullet R_{c'} (\backslash s, s) \Rightarrow T_{c''} s$ $\exists (i : I. \backslash b_i) \wedge \forall i : I. \backslash b_i \Rightarrow T_{c'_i} \backslash s$

For skip:  $R_{\text{skip}} (\backslash s, s') \equiv s' = \backslash s$  and  $T_{\text{skip}} \backslash s \equiv 1$ . For abort:  $R_{\text{abort}} (\backslash s, s') \equiv 0$  and  $T_{\text{abort}} \backslash s \equiv 0$ . For iteration:  $\text{do } b \rightarrow c' \text{ od}$  is defined to have the same effect as  $\text{if } \neg b \rightarrow \text{skip} \square b \rightarrow (c'; \text{do } b \rightarrow c' \text{ od}) \text{ fi}$ .

**Hoare semantics** A *Hoare triple* describes all possible computations for command  $c$  (i.e., those in  $(\mathbf{S}^2)_{R_c}$ ) starting in a state satisfying  $A : \mathbf{S} \rightarrow \mathbb{B}$  (*antecedent*) and terminating in a state satisfying  $P : \mathbf{S} \rightarrow \mathbb{B}$  (*postcondition*) by a predicate of type  $(\mathbf{S} \rightarrow \mathbb{B}) \times C \times (\mathbf{S} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$  defined next for partial correctness (12) and total correctness (13), using  $\text{Term}_- : C \rightarrow (\mathbf{S} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$  for termination (14).

$$\{A\} c \{P\} \equiv \forall \backslash s. \forall s'. R_c (\backslash s, s') \Rightarrow A \backslash s \Rightarrow P s' \quad (12)$$

$$[A] c [P] \equiv \{A\} c \{P\} \wedge \text{Term}_c A \quad (13)$$

$$\text{Term}_c A \equiv \forall s \bullet A s \Rightarrow T_c s \quad (14)$$

**Calculating Dijkstra semantics** We define the *weakest liberal antecedent* and *weakest antecedent* by predicate transformers  $\text{Wla}_- : C \rightarrow (\mathbf{S} \rightarrow \mathbb{B}) \rightarrow (\mathbf{S} \rightarrow \mathbb{B})$  and  $\text{Wla}_- : C \rightarrow (\mathbf{S} \rightarrow \mathbb{B}) \rightarrow (\mathbf{S} \rightarrow \mathbb{B})$  with implicit equations

$$\{A\} c \{P\} \equiv \forall s \bullet A s \Rightarrow \text{Wla}_c P s , \quad (15)$$

$$[A] c [P] \equiv \forall s \bullet A s \Rightarrow \text{Wa}_c P s . \quad (16)$$

To obtain explicit formulas, we calculationaly transform (12) and (13) to match the shape of (15) and (16):

$$\begin{aligned}
\{A\} c \{P\} &\equiv \langle \text{Definit. (12)} \rangle \forall s \bullet \forall s' \bullet \mathbf{R}_c(s, s') \Rightarrow A \backslash s \Rightarrow P s' \\
&\equiv \langle \text{Shunting} \Rightarrow \rangle \forall s \bullet \forall s' \bullet A \backslash s \Rightarrow \mathbf{R}_c(s, s') \Rightarrow P s' \\
&\equiv \langle \text{Ldistr.} \Rightarrow / \forall \rangle \forall s \bullet A \backslash s \Rightarrow \forall s' \bullet \mathbf{R}_c(s, s') \Rightarrow P s' \\
&\equiv \langle \text{Var. change} \rangle \forall s \bullet A s \Rightarrow \forall s' \bullet \mathbf{R}_c(s, s') \Rightarrow P s' , \\
[A] c [P] &\equiv \langle \text{Definit. (13)} \rangle \{A\} c \{P\} \wedge \text{Term}_c A \\
&\equiv \langle \text{Def. (15, 14)} \rangle \forall (s \bullet A s \Rightarrow \mathbf{Wla}_c P s) \wedge \forall s \bullet A s \Rightarrow \mathbf{T}_c s \\
&\equiv \langle \text{Distrib.} \forall / \wedge \rangle \forall s \bullet (A s \Rightarrow \mathbf{Wla}_c P s) \wedge (A s \Rightarrow \mathbf{T}_c s) \\
&\equiv \langle \text{Ldistr.} \Rightarrow / \wedge \rangle \forall s \bullet A s \Rightarrow \mathbf{Wla}_c P s \wedge \mathbf{T}_c s .
\end{aligned}$$

Matching yields (uniqueness being easy to show)

$$\mathbf{Wla}_c P s \equiv \forall s' \bullet \mathbf{R}_c(s, s') \Rightarrow P s' , \quad (17)$$

$$\mathbf{Wa}_c P \equiv \mathbf{Wla}_c P \hat{\wedge} \mathbf{T}_c . \quad (18)$$

Note the striking similarity of the calculations with those in the analysis example: everything is predicate calculus.

From (17) and (18) we can calculate properties that in the literature are always given as postulates [14]. The same holds for the results obtained by substituting the program equations for the various language constructs [11].

$$\begin{aligned}
\mathbf{Wa}_{v:=e} P s &\equiv P s \overset{v}{e} \\
\mathbf{Wa}_{c'; c''} &\equiv \mathbf{Wa}_{c'} \circ \mathbf{Wa}_{c''} \\
\mathbf{Wa}_{i \text{ f } \parallel i : I . b_i \rightarrow c'_i \text{ f } i} P s &\equiv \exists b \wedge \forall i : I . b_i \Rightarrow \mathbf{Wa}_{c'_i} P s \\
\mathbf{Wa}_{\text{do } b \rightarrow c' \text{ od}} P s &\equiv \exists n : \mathbb{N} . \mathbf{w}^n (s \bullet \neg b \wedge P s) s \\
\text{defining } \mathbf{w} \text{ by } \mathbf{w} Q s &\equiv (\neg b \wedge P s) \vee (b \wedge \mathbf{Wa}_{c'} Q s) .
\end{aligned}$$

**Computation sequences and program semantics** The final example shows how a model for R. Dijkstra's *Computation Calculus* (CC) [15] can be expressed and properties derived using operators for sequences which we originally developed for reasoning about signals and systems.

Preliminaries: for sequences in  $A^\omega$ , define *take* ( $\lfloor \rfloor$ ) by  $x \lfloor n = x_{<n}$  and *drop* ( $\lfloor \rfloor$ ) by  $x \lfloor n = \sigma^n x$  for any  $x : A^\omega$  and  $n : \square (\#x + 1)$ . For predicates with common domain  $X$ , define relation  $\sqsubseteq$  by  $P \sqsubseteq Q \equiv \forall x : X . P x \Rightarrow Q x$ . Letting  $\sqcap$  stand for  $\hat{\wedge}$ , note that  $P \sqsubseteq Q \equiv P = P \sqcap Q$ .

In CC, *computations* are elements of  $\mathcal{C} := \mathbb{S}^+ \cup \mathbb{S}^\infty$  for state space  $\mathbb{S}$ . Specifications and behaviors are expressed by *computation predicates* of type  $\mathbf{CP} := \mathcal{C} \rightarrow \mathbb{B}$ , e.g.,  $\mathbf{T} := \mathcal{C} \bullet 1$  and  $\mathbf{F} := \mathcal{C} \bullet 0$ . *Composition*  $\dashv\vdash; \dashv\vdash : \mathbf{CP}^2 \rightarrow \mathbb{B}$  is defined for arbitrary  $C', C'' : \mathbf{CP}^2$  and  $\gamma : \mathcal{C}$  by

$$\begin{aligned}
(C' ; C'') \gamma &\equiv (\# \gamma = \infty \wedge C' \gamma) \vee \\
&\quad \exists n : \mathcal{D} \gamma . C' (\gamma \lfloor (n+1)) \wedge C'' (\gamma \lfloor n) .
\end{aligned}$$

Predicate calculus shows that composition is associative and that the predicate  $\mathbb{1} : \mathbf{CP}$  with  $\mathbb{1} \gamma \equiv \# \gamma = 1$  is a 2-sided unit element. We give composition precedence over  $\sqcap$  and  $\sqcup$ , hence  $C \sqcap C' ; C'' = C \sqcap (C' ; C'')$ .

States are represented by sequences of length 1, e.g.,  $\gamma_\alpha := \tau \gamma_0$  for initial states and  $\gamma_\omega := \tau \gamma_{\# \gamma - 1}$  if  $\# \gamma \neq \infty$  for final states. *State predicates* are predicates of type  $\mathbf{SP} := \{P : \mathbf{CP} \mid P \sqsubseteq \mathbb{1}\}$ . Predicate calculus shows that (i)  $(P ; \mathbf{T}) \gamma \equiv P \gamma_\alpha$ , (ii)  $(\mathbf{T} ; P) \gamma \equiv \# \gamma \neq \infty \Rightarrow P \gamma_\omega$ ,

(iii)  $P ; C = P ; \mathbf{T} \sqcap C$  and (iv)  $C ; P = C \sqcap \mathbf{T} ; P$  for any  $P : \mathbf{SP}$  and  $C : \mathbf{CP}$ .

For the *eternity* predicate  $\mathbf{E} := \mathbf{T} ; \mathbf{F}$  and the *bounded* predicate  $\mathbf{B} := \neg \mathbf{E}$ , clearly  $\mathbf{E} \gamma \equiv \# \gamma = \infty$  and  $\mathbf{B} \gamma \equiv \# \gamma \neq \infty$ . CC defines, for any  $A$  en  $P$  in  $\mathbf{SP}$  en  $C$  in  $\mathbf{CP}$ ,

$$\{A\} C \{P\} \equiv A ; C \sqsubseteq \mathbf{T} ; P \quad (19)$$

$$[A] C [P] \equiv A ; C \sqsubseteq \mathbf{B} ; P \quad (20)$$

A first calculation example is bringing  $[A] C [P]$  into the form  $\{A\} C \{P\} \wedge T$ , where  $T$  is to be discovered.

$$\begin{aligned}
[A] C [P] &\equiv \langle \text{Def. (20)} \rangle A ; C \sqsubseteq \mathbf{B} ; P \\
&\equiv \langle \text{Prop. (iv)} \rangle A ; C \sqsubseteq \mathbf{B} \sqcap \mathbf{T} ; P \\
&\equiv \langle \text{Rdist.} \sqcap / \sqsubseteq \rangle A ; C \sqsubseteq \mathbf{B} \wedge A ; C \sqsubseteq \mathbf{T} ; P \\
&\equiv \langle \text{Def. (19)} \rangle A ; C \sqsubseteq \mathbf{B} \wedge \{A\} C \{P\} .
\end{aligned}$$

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

A calculation example spanning across theories is “reverse engineering” to find *systems equations*, i.e., abstract variants of program equations, capturing CC. So we calculate  $\mathbf{R}_- : \mathbf{CP} \rightarrow \mathbb{S}^2 \rightarrow \mathbb{B}$  en  $\mathbf{T}_- : \mathbf{CP} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$  such that

$$\{A\} C \{P\} \equiv \quad (21)$$

$$\begin{aligned}
&\quad \forall (s, s') : \mathbb{S}^2 . \mathbf{R}_C(s, s') \Rightarrow A(\tau \backslash s) \Rightarrow P(\tau s') , \\
A ; C \sqsubseteq \mathbf{B} &\equiv \forall s : \mathbb{S} . A(\tau \backslash s) \Rightarrow \mathbf{T}_C \backslash s . \quad (22)
\end{aligned}$$

These are variants of (12, 14) for  $\mathbf{SP}$ . Calculating:

$$\begin{aligned}
A ; C \sqsubseteq \mathbf{B} &\equiv \langle \text{Prop. (iii)} \rangle A ; \mathbf{T} \sqcap C \sqsubseteq \mathbf{B} \\
&\equiv \langle \text{Defin.} \sqsubseteq \rangle \forall \gamma : \mathcal{C} . (A ; \mathbf{T} \sqcap C) \gamma \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{Defin.} \sqcap \rangle \forall \gamma : \mathcal{C} . (A ; \mathbf{T}) \gamma \wedge C \gamma \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{Prop. (i)} \rangle \forall \gamma : \mathcal{C} . A \gamma_\alpha \wedge C \gamma \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{Shunt} \wedge \rangle \forall \gamma : \mathcal{C} . A \gamma_\alpha \Rightarrow C \gamma \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{Defin.} \gamma_\alpha \rangle \forall \gamma : \mathcal{C} . A(\tau \gamma_0) \Rightarrow C \gamma \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{1-pt. rule} \rangle \\
&\quad \forall \gamma : \mathcal{C} . \forall s : \mathbb{S} . \backslash s = \gamma_0 \Rightarrow A(\tau \backslash s) \Rightarrow C \gamma \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{Shunt} \Rightarrow \rangle \\
&\quad \forall \gamma : \mathcal{C} . \forall s : \mathbb{S} . A(\tau \backslash s) \Rightarrow C \gamma \Rightarrow \backslash s = \gamma_0 \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{Swap} \forall \rangle \\
&\quad \forall s : \mathbb{S} . \forall \gamma : \mathcal{C} . A(\tau \backslash s) \Rightarrow C \gamma \Rightarrow \backslash s = \gamma_0 \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{Ldst.} \Rightarrow / \forall \rangle \\
&\quad \forall s : \mathbb{S} . A(\tau \backslash s) \Rightarrow \forall \gamma : \mathcal{C} . C \gamma \Rightarrow \backslash s = \gamma_0 \Rightarrow \mathbf{B} \gamma \\
&\equiv \langle \text{Trading} \rangle \forall s : \mathbb{S} . A(\tau \backslash s) \Rightarrow \forall \gamma : \mathcal{C} . \backslash s = \gamma_0 \Rightarrow \mathbf{B} \gamma .
\end{aligned}$$

Hence the equation for  $\mathbf{T}$  satisfying (22) is

$$\mathbf{T}_C \backslash s \equiv \forall \gamma : \mathcal{C} . \gamma_0 = \backslash s \Rightarrow \# \gamma \neq \infty .$$

Expanding  $\{A\} C \{P\}$  yields  $\mathbf{R}_C$  satisfying (21):

$$\mathbf{R}_C(s, s') \equiv \exists \gamma : \mathcal{C} . \gamma_0 = \backslash s \wedge \# \gamma \neq \infty \wedge \gamma_{\# \gamma - 1} = s' .$$

Both equations have a very direct intuitive interpretation.

## 4 CONCLUSION

We have shown how a formalism, consisting of a very simple language of only 4 constructs, together with a powerful set of formal calculation rules, yields a notational and methodological unification of continuous and discrete

mathematics, especially in the areas directly relevant to signal processing.

Apart from the obvious scientific ramifications, the formalism provides a unified basis for education in ECE, as advocated in [22].

Future work will concentrate on applying the flexible calculational method to synchronous and asynchronous parallel operations in DSP systems.

## 5 REFERENCES

- [1] Vicki L. Almstrum, "Investigating Student Difficulties With Mathematical Logic", in: C. Neville Dean and Michael G. Hinchey, eds., *Teaching and Learning Formal Methods*, pp. 131–160. Academic Press (1996)
- [2] Rajeev Alur, Thomas A. Henzinger, Eduardo D. Sontag, eds., *Hybrid Systems III*, LNCS 1066. Springer-Verlag, Berlin Heidelberg (1996)
- [3] Eerke Boiten and Bernhard Möller, *Sixth International Conference on Mathematics of Program Construction* (Conference announcement), Dagstuhl (2002).  
[www.cs.kent.ac.uk/events/conf/2002/mpc2002](http://www.cs.kent.ac.uk/events/conf/2002/mpc2002)
- [4] Raymond T. Boute, "A heretical view on type embedding", *ACM Sigplan Notices* 25, pp. 22–28 (Jan. 1990)
- [5] Raymond T. Boute, *Funmath illustrated: A Declarative Formalism and Application Examples*. Declarative Systems Series No. 1, Computing Science Institute, University of Nijmegen (July 1993)
- [6] Raymond T. Boute, "Fundamentals of hardware description languages and declarative languages", in: Jean P. Mermet, ed., *Fundamentals and Standards in Hardware Description Languages*, pp. 3–38, Kluwer Academic Publishers (1993)
- [7] Raymond T. Boute, "Supertotal Function Definition in Mathematics and Software Engineering", *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, pp. 662–672 (July 2000)
- [8] Raymond Boute, *Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I: Basic Mathematics*. Course text, Ghent University (2002)
- [9] Raymond T. Boute, "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons and Johan Jeuring, eds., *Generic Programming*, pp. 89–119, Kluwer (2003)
- [10] Raymond Boute, "Functional declarative language design and predicate calculus: a practical approach" (to appear in *ACM Trans. Prog. Lang. and Syst.*).
- [11] Raymond T. Boute, "Calculational semantics: deriving programming theories from equations by functional predicate calculus" (to appear in *ACM Trans. Prog. Lang. and Syst.*)
- [12] Ronald N. Bracewell, *The Fourier Transform and Its Applications*, 2nd ed, McGraw-Hill (1978)
- [13] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: a framework for simulating and prototyping heterogeneous systems", *International Journal of Computer Simulation*, spec. issue on Simulation Software Development (Jan. 1994)
- [14] Edsger W. Dijkstra and Carel S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin (1990)
- [15] Rutger M. Dijkstra, "Computation Calculus: Bridging a Formalization Gap", *Proc. Mathematics of Program Construction, Springer LNCS 1422*, pp. 151–174 (1998)
- [16] David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, Berlin (1993)
- [17] David Gries, "The need for education in useful formal logic", *IEEE Computer* 29, 4, pp. 29–30 (April 1996)
- [18] Paul N. Hilfinger, *Silage Reference Manual, Rev. 1.3*. University of California (1987)
- [19] Kathleen Jensen and Niklaus Wirth, *PASCAL User Manual and Report*. Springer-Verlag, Berlin (1978)
- [20] Serge Lang, *Undergraduate Analysis*. Springer-Verlag, Berlin (1983)
- [21] Leslie Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education Inc. (2002)
- [22] Edward A. Lee and David G. Messerschmitt, "Engineering — an Education for the Future", *IEEE Computer*, Vol. 31, No. 1, pp. 77–85 (Jan. 1998), via [ptolemy.eecs.berkeley.edu/publications/papers/98/](http://ptolemy.eecs.berkeley.edu/publications/papers/98/)
- [23] Edward A. Lee and Pravin Varaiya, "Introducing Signals and Systems — The Berkeley Approach", *First Signal Processing Education Workshop*, Hunt, Texas (Oct. 2000), via [ptolemy.eecs.berkeley.edu/publications/papers/00/](http://ptolemy.eecs.berkeley.edu/publications/papers/00/)
- [24] Rex Page, *BESEME: Better Software Engineering through Mathematics Education*, project presentation <http://www.cs.ou.edu/~beseme/besemePres.pdf>
- [25] David L. Parnas, "Education for Computing Professionals", *IEEE COMPUTER* 23, 1, pp. 17–22 (Jan. 1990)
- [26] William Pugh, "Counting Solutions to Presburger Formulas: How and Why", *ACM SIGPLAN Notices* 29, 6, pp. 121–122 (June 1994)
- [27] Paul Taylor, *Practical Foundations of Mathematics* (second printing), No. 59 in *Cambridge Studies in Advanced Mathematics*, Cambridge University Press (2000); quoted from the introduction to Chapter 1 in [www.dcs.qmul.ac.uk/~pt/PracticalFoundations/html](http://www.dcs.qmul.ac.uk/~pt/PracticalFoundations/html)
- [28] Frits W. Vaandrager, Jan H. van Schuppen, eds., *Hybrid Systems: Computation and Control*, LNCS 1569. Springer-Verlag, Berlin Heidelberg (1999)
- [29] Jeannette M. Wing, "Weaving Formal Methods into the Undergraduate Curriculum", *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST)* pp. 2–7 (May 2000); file `amast00.html` in [www-2.cs.cmu.edu/afs/cs.cmu.edu/project/calder/www/](http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/calder/www/)