Formal Methods Unifying Computing Science and Systems Theory

Raymond BOUTE
INTEC, Universiteit Gent
B-9000 Gent, Belgium

ABSTRACT

Computing Science and Systems Theory can gain much from unified mathematical models and methodology, in particular formal reasoning ("letting the symbols do the work"). This is achieved by a wide-spectrum formalism.

The language uses just four constructs, yet suffices to synthesize familiar notations (minus the defects) as well as new ones. It supports formal calculation rules convenient for hand calculation and amenable to automation.

The basic framework has two main elements. First, a functional predicate calculus makes formal logic practical for engineers, allowing them to calculate with predicates and quantifiers as easily as with derivatives and integrals. Second, concrete generic functionals support smooth transition between pointwise and point-free formulations, facilitating calculation with functionals and exploiting formal commonalities between CS and Systems Theory.

Elaborating a few small but representative examples show how formal calculational reasoning about diverse topics such as mathematical analysis, program semantics, transform methods, systems properties (causality, LTI), data types and automata provides a unified methodology.

Keywords Calculation, Computing Science, Concrete Generic Functionals, Formal Methods, Functional Predicate Calculus, Quantifiers, Systems Theory, Unification

1. INTRODUCTION

Motivation: *ut faciant opus signa*
Computing Science and Systems Theory are fundamental to engineering in general [17] and ICT in particular. Complex systems heavily rely on both. Yet, the conceptual frameworks and modelling techniques are (still) very divergent. The crucial loss is that the benefits of formal reasoning are much underexploited. We briefly elaborate.

Whoever enjoyed physics will recall the excitement when judicious manipulation of formulas yielded results not obtainable by mere intuition. Such manipulation, from polynomial factorization in high school to calculation with derivatives and integrals in calculus, is essentially *formal*, i.e., guided by the shape of the expressions. The usual style is *calculational*, namely, chaining expressions by relational operators such as equality ("="). An example is

$$\begin{aligned} F(s) &= \int_{-\infty}^{+\infty} e^{-|x|} e^{-i2\pi xs} dx \\ &= 2\int_{0}^{+\infty} e^{-x} \cos 2\pi xs \; dx \\ &= 2 \text{ Re} \int_{0}^{+\infty} e^{-x} e^{i2\pi xs} dx \\ &= 2 \text{ Re } \tfrac{-1}{i2\pi s - 1} \\ &= \tfrac{2}{4\pi^2 s^2 + 1} \ , \end{aligned} \tag{1}$$

taken from a classic engineering text by Bracewell [10].

The typical formal rules used are those for arithmetic (associativity, distributivity etc.) plus those from calculus.

Exploiting formality and the calculational style are taken for granted throughout most of applied mathematics based on algebra and calculus (although, as shown later, common conventions still exhibit some serious defects).

By contrast, logical reasoning in everyday practice by mathematicians and engineers is highly informal, and often involves what Taylor [20] calls *syncopation*, namely using symbols as mere abbreviations of natural language, for instance the quantifier symbols $\forall$ and $\exists$ just standing for "for all" and "there exists", without calculation rules.

The result is a severe style breach between "regular calculus", usually done in an essentially formal way, and the logical justification of its rules, which even in the best analysis texts is done in words, with syncopation instead of calculation. As Taylor observes, the logical structure of the arguments is thereby often seriously obscured.

This style breach pervades applied mathematics, and is reflected in the methodological gap between classical Systems Theory, based on calculus, and Computer Science, based on logic. As explained by Gries [13], although formal logic exists as a separate discipline, its traditional form is drowned in technicalities that make it too cumbersome for practical use, but now calculational variants exist [12].

The rewards of bridging the gap are huge, namely making the symbols do the work, as nicely captured by the maxim "*Ut faciant opus signa*" of the conference series on Mathematics of Program Construction [2]. Here we do not mean only (nor even primarily) using software tools, but also the guidance provided by the shape of the expressions in mathematical reasoning, and the development of a "parallel intuition" to that effect. This complements the usual "semantic" intuition, especially when exploring areas where the latter is clueless or still in development.

Approach: Functional Mathematics (*Funmath*)
A unifying formalism is presented that spans a wide application spectrum. A *formalism* is a *language* (or notation) together with *formal rules* for symbolic manipulation.

The language [5] is *functional* in the sense that functions are first-class objects and also form the basis for unification. It supports declarative (abstract) as well as operational (implementation) aspects throughout all mathematics relevant to computer and systems engineering, and is free of all defects of common conventions, including those outlined by Lee and Varaiya [18] as discussed later.

The formal rules are *calculational*, supporting the same style from predicate logic through calculus. Thereby the conceptual and notational unification provided by the language is complemented by unified methodology.

In particular, this enables engineers to formally calculate with predicates and quantifiers with the same ease and algebraic flavor as with derivatives and integrals.

Overview
The formalism is presented in section 2, which introduces the language, its rationale and its four basic constructs, and in section 3, which gives the general-purpose formal rules, namely those for concrete generic functionals and for functional predicate calculus (quantifiers). Application examples are given in section 4 for Systems Theory, section 5 for Computing Science, and section 6 for common aspects. Some concluding remarks are given in section 7.

2. THE FORMALISM, PART A: LANGUAGE

Rationale: the need for defect-free notation
Notation is unimportant if and only if it is well-designed, but becomes a crucial stumbling block if it is deficient. The criterion is supporting formal calculation: if during expression manipulation one has to be on guard for the defects, one cannot let the symbols do the work.

In long-standing areas of mathematics such as algebra and analysis, conventions are largely problem-free, but not entirely. Most important are violations of Leibniz's principle, i.e., that equals may always be substituted for equals. An example is ellipsis: writing dots as in $a_0+a_1+\ldots+a_n$. By Leibniz's principle, if $a_i = i^2$ and $n = 7$, this should equal $0 + 1 + \ldots + 49$, which most likely is not intended. Other defects, also pointed out in [18], are related to writing function application when the function is intended, as in $y(t) = x(t) * h(t)$ where $*$ is convolution. This causes instantiation to be incorrect, e.g., $y(t-\tau) = x(t-\tau) * h(t-\tau)$.

In discrete mathematics the situation is worse, e.g., for the sum-$\sum$ many conventions are mutually inconsistent and calculation rules are rarely given. Poorest are the conventions in logic and set theory used in daily practice. A typical defect is abusing the set membership relation $\in$ for binding a dummy. Frequent patterns are $\{x \in X \mid p\}$, as in $\{m \in \mathbb{Z} \mid m < n\}$, and $\{e \mid x \in X\}$, as in $\{n \cdot m \mid m \in \mathbb{Z}\}$, where in the patterns p is boolean and e any expression. The ambiguity is shown by taking $y \in Y$ for p and e. Defects like these prohibit formal rules and explain why, for such expressions, syncopation prevails in the literature.

Funmath language design
We do not patch defects ad hoc, but generate correct forms by orthogonal combination of just 4 constructs, gaining new useful forms of expression for free. 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.

Identifier: any symbol or string except colon, filter mark, abstraction dot, parentheses, and a few keywords.

Identifiers are *introduced* by *bindings* $i : X \mathbin{\&} p$, read "i in X satisfying p", where i is the (tuple of) identifier(s), X a set and p a proposition. The *filter* $\mathbin{\&} p$ (or **with** p) is optional, e.g., $n : \mathbb{N}$ and $n : \mathbb{Z} \mathbin{\&} n \geq 0$ are interchangeable. Identifiers from i should not appear in expression X.

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

Identifiers can be *variables* (in an abstraction) or *constants* (declared by **def** *binding*). Well-established symbols, such as $\mathbb{B}$, $\Rightarrow$, $\mathbb{R}$, $+$, serve as predefined constants.

Application: for function f and argument e, the default is $f\,e$; other conventions are specified by dashes in the operator's binding, e.g., $-\star-$ for infix. For clarity, parentheses are *never* used as operators, but only for parsing. Rules for making them optional are the usual ones. If f is a function-valued function, $f\,x\,y$ stands for $(f\,x)\,y$.

Let $\star$ be infix. *Partial application* is of the form $a\star$ or $\star b$, defined by $(a\star)\,b = a \star b = (\star b)\,a$. *Variadic application* is of the form $a * b * c$ etc., and is *always* defined to equal $F\,(a, b, c)$ for a suitably defined *elastic extension* F of $\star$.

Abstraction: the form is $b\,.\,e$, where b is a binding and e an expression (extending after "$.$" as far as compatible with parentheses present). Intuitively, $v : X \mathbin{\&} p\,.\,e$ denotes a function whose domain is the set of v in X satisfying p, and mapping v to e (formalized in section 3). Syntactic sugar: $e \mid b$ for $b\,.\,e$ and $v : X \mid p$ for $v : X \mathbin{\&} p\,.\,v$.

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* $\varepsilon := \emptyset\,.\,e$ (any e) and defining $\mapsto$ by $e' \mapsto e = \iota\, e' \bullet e$ for *one-point functions*.

We shall see how abstractions help synthesizing familiar expressions such as $\sum i : 0\,..\,n\,.\,q^i$ and $\{m : \mathbb{Z} \mid m < n\}$.

Tupling: the 1-dimensional 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 ε and for singleton tuples we define τ with $\tau\, e = 0 \mapsto e$.

Parentheses are *not* part of tupling, and are as optional in (m, n) as in $(m+n)$. Matrices are 2-dimensional tuples.

3. THE FORMALISM, PART B: RULES

The formal calculation rules and gaining fluency with them is the topic of a full course [7], so here we must be terse.

Rules for equational and calculational reasoning
The equational style of Eq. (1) is generalized to the format

$$e \quad R'\,\langle\text{Justification}\rangle' \quad e' \ , \tag{2}$$

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

In general, for any theorem p we have the rule

$$\textsc{Instantiation}\text{: from } p, \text{ infer } p[^v_e \ . \tag{3}$$

We write $[^v_e$ to express substitution of e for v, for instance, $(x + y = y + x)[^{x,y}_{3,z+1} = 3 + (z + 1) = (z + 1) + 3$.

For equational reasoning (i.e., using $=$ or $\equiv$ only), the basic rules [12] are reflexivity, symmetry, transitivity and

$$\textsc{Leibniz's principle}\text{: from } e = e', \text{ infer } d[^v_e = d[^v_{e'} \ . \tag{4}$$

For instance, $x + 3 \cdot y \quad = \langle x = z^2 \rangle \quad z^2 + 3 \cdot y$. Eq. (4) is used by taking $d := v + 3 \cdot y$ and $e := x$ and $e' := z^2$.

Rules for calculating with propositions and sets
Assume the usual propositional operators $\neg$, $\equiv$, $\Rightarrow$, $\wedge$, $\vee$. For a practical calculus, a much more extensive set of rules is needed than given in classical texts on logic, so we refer to Gries [12]. Note that $\equiv$ is associative, but $\Rightarrow$ is not. We make parentheses in $p \Rightarrow (q \Rightarrow r)$ optional, hence required in $(p \Rightarrow q) \Rightarrow r$. Embedding binary algebra in arithmetic [3, 4], logic constants are 0 and 1, not FALSE and TRUE.

Leibniz's principle can be rewritten $e = e' \Rightarrow d[^v_e = d[^v_{e'}$.

For sets, the basic operator is $\in$. The 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 $(x, y) \in X \times Y \ \equiv \ x \in X \wedge y \in Y$. After defining $\{-\!-\}$, we shall be able to prove $y \in \{x : X \mid p\} \ \equiv \ y \in X \wedge p[^x_y$.

Set equality is defined via *Leibniz's principle*, written as an implication: $X = Y \Rightarrow (x \in X \equiv x \in Y)$ and the converse, *extensionality*, written here as an inference rule: from $x \in X \equiv x \in Y$, infer $X = Y$, with x a new variable. This rule is *strict*, i.e., the premiss must be a theorem.

Rules for functions and generic functionals

We omit the design decisions, to be found in [5] and [8]. 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 via *Leibniz's principle* (taking domains into account) $f = g \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x)$, and *extensionality* as a strict inference rule: with new x, from $p \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x)$, infer $p \Rightarrow f = g$.

Abstraction encapsulates substitution. Formal axiom: for the *domain* $d \in \mathcal{D}\,(v : X \wr p \,.\, e) \equiv d \in X \wedge p[^v_d]$, and for the *mapping*: $d \in \mathcal{D}\,(v : X \wr p \,.\, e) \Rightarrow (v : X \wr p \,.\, e)\, d = e[^v_d]$. Equality is characterized via function equality (exercise).

Generic functionals Our goal is (a) removing restrictions in common functionals from mathematics, (b) making often-used implicit functionals from systems theory explicit. The idea is defining the result domain to avoid out-of-domain applications in the image definition.

Case (a) is illustrated by composition $f \circ g$, whose common definition requires $\mathcal{R}\, g \subseteq \mathcal{D}\, f$; then $\mathcal{D}\,(f \circ g) = \mathcal{D}\, g$. Removing the restriction, we define $f \circ g$ for *any* functions:

$$f \circ g \quad = \quad x : \mathcal{D}\, g \wr g\, x \in \mathcal{D}\, f \,.\, f\,(g\, x) \ . \tag{5}$$

Observation: $x \in \mathcal{D}\,(f \circ g) \equiv x \in \mathcal{D}\, g \wedge g\, x \in \mathcal{D}\, f$ by the abstraction axiom, hence $\mathcal{D}\,(f \circ g) = \{x : \mathcal{D}\, g \mid g\, x \in \mathcal{D}\, f\}$.

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* ($\widehat{\ }$): for any functions $\star$ (infix), f, g,

$$f \mathbin{\widehat{\star}} g \ = \ x : \mathcal{D}\, f \cap \mathcal{D}\, g \wr (f\, x, g\, x) \in \mathcal{D}\,(\star) \,.\, f\, x \star g\, x \ . \tag{6}$$

Often we need *half direct extension*: for function f, any e,

$$f \mathbin{\overleftarrow{\star}} e = f \mathbin{\widehat{\star}} (\mathcal{D}\, f \bullet e) \quad \text{and} \quad e \mathbin{\overrightarrow{\star}} f = (\mathcal{D}\, f \bullet e) \mathbin{\widehat{\star}} f \ . \tag{7}$$

Simplex direct extension ($\overline{\ }$) is defined by $\overline{f}\, g = f \circ g$.

Function merge ($\cup$) is defined in 2 parts to fit the line: $\mathcal{D}\,(f \cup g) \ = \ \{x : \mathcal{D}\, f \cup \mathcal{D}\, g \mid x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x\}$ and $x \in \mathcal{D}\,(f \cup g) \Rightarrow (f \cup g)\, x = (x \in \mathcal{D}\, f)\, ?\, f\, x \mathbin{\dot{+}} g\, x$. (8)

Filtering ($\downarrow$) introduces/eliminates arguments:

$$f \downarrow P \quad = \quad x : \mathcal{D}\, f \cap \mathcal{D}\, P \wr P\, x \,.\, f\, x \ . \tag{9}$$

A particularization is *restriction* ($\restriction$): $f \restriction X \ = \ f \downarrow (X^\bullet 1)$. 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, this yields formal rules for useful shorthands like $f_{<n}$ and $\mathbb{Z}_{>0}$.

A relational generic functional is compatibility ($\copyright$) with $f \copyright g \equiv f \restriction \mathcal{D}\, g = g \restriction \mathcal{D}\, f$. For many other generic functionals and their elastic extensions, we refer to [8].

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.

Rules for predicate calculus and quantifiers

Axioms and forms of expression The *quantifiers* $\forall$, $\exists$ are predicates over predicates: for any predicate P,

$$\forall P \ \equiv \ P = \mathcal{D}\, P^\bullet 1 \quad \text{and} \quad \exists P \ \equiv \ P \neq \mathcal{D}\, P^\bullet 0 \tag{10}$$

Let p and q be propositions, then p, q is a predicate and $\forall\,(p, q) \equiv p \wedge q$. So $\forall$ is an elastic extension of $\wedge$ and we define variadic application by $p \wedge q \wedge r \ \equiv \ \forall\,(p, q, r)$ etc.

Letting P be an abstraction $v : X \,.\, p$ yields the familiar form $\forall\, v : X \,.\, p$, as in $\forall\, x : \mathbb{R} \,.\, x^2 \geq 0$. For every algebraic law, most elegantly stated in point-free form, a matching pointwise (familiar-looking) form is obtained in this way.

Derived rules All laws follow from Eq. (10) and function equality. A collection sufficient for practice is derived in [7]. Here we only give some examples, starting with a characterization of $f = g$ without inference rules:

$$f = g \ \equiv \ \mathcal{D}\, f = \mathcal{D}\, g \wedge \forall\, x : \mathcal{D}\, f \cap \mathcal{D}\, g \,.\, f\, x = g\, x \ . \tag{11}$$

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

$$\overline{\neg} \forall\, P = \exists\,(\overline{\neg}\, P) \qquad \neg\,(\forall\, v : X \,.\, p) \ \equiv \ \exists\, v : X \,.\, \neg\, p \ . \tag{12}$$

Here are the main distributivity laws. All have duals.

Name of the rule	Point-free form
Distributivity $\vee/\forall$	$q \vee \forall\, P \equiv \forall\,(q \mathbin{\overline{\vee}} P)$
L(eft)-distrib. $\Rightarrow/\forall$	$q \Rightarrow \forall\, P \equiv \forall\,(q \mathbin{\overrightarrow{\Rightarrow}} P)$
R(ight)-distr. $\Rightarrow/\exists$	$\exists\, P \Rightarrow q \equiv \forall\,(P \mathbin{\overleftarrow{\Rightarrow}} q)$
P(seudo)-dist. $\wedge/\forall$	$\mathcal{D}\, P = \emptyset \vee (p \wedge \forall\, P) \equiv \forall\,(p \mathbin{\overrightarrow{\wedge}} P)$

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

Here are a few additional illustrative laws.

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

Distributivity $\forall/\wedge$ assumes $\mathcal{D}\, P = \mathcal{D}\, Q$, otherwise only $\forall\, P \wedge \forall\, Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q)$. The one-point rule written pointwise is $\forall\,(v : X \,.\, v = e \Rightarrow p) \ \equiv \ e \in X \Rightarrow p[^v_e]$. For the last line, $R : S \to T \to \mathbb{B}$ and $(v : X \,.\, w : Y \,.\, e)^\mathsf{T} = w : Y \,.\, v : X \,.\, e$, hence $\forall\,(v : X \,.\, \forall\, w : Y \,.\, p) \ \equiv \ \forall\,(w : Y \,.\, \forall\, v : X \,.\, p)$ ($\forall$-swap). Duals and other pointwise forms are left as an exercise.

Sometimes the following rules are useful:
Instantiation: $\forall\, P \Rightarrow e \in \mathcal{D}\, P \Rightarrow P\, e$ and, with new x: *Generalization*: from $p \Rightarrow x \in \mathcal{D}\, P \Rightarrow P\, x$, infer $\forall\, P$.

Wrapping up the rule package for function(al)s

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

$$e \in \mathcal{R}\, f \ \equiv \ \exists\, x : \mathcal{D}\, f \,.\, f\, x = e \ . \tag{13}$$

A consequence is the composition rule $\forall\, P \Rightarrow \forall\,(P \circ f)$ and $\mathcal{D}\, P \subseteq \mathcal{R}\, f \ \Rightarrow \ (\forall\,(P \circ f) \ \equiv \ \forall\, P)$, whose pointwise form yields $\forall\,(y : \mathcal{R}\, f \,.\, p) \ \equiv \ \forall\,(x : \mathcal{D}\, f \,.\, p[^y_{f\, x}])$ ("dummy change").

Set comprehension We define $\{—\}$ as *fully interchangeable* with $\mathcal{R}$. This yields defect-free set notation: expressions like $\{2, 3, 5\}$ and $\mathit{Even} = \{2 \cdot m \mid m : \mathbb{Z}\}$ have familiar form and meaning, and all desired calculation rules follow from predicate calculus via Eq. (13). In particular, we can prove $e \in \{v : X \mid p\} \ \equiv \ e \in X \wedge p[^v_e]$ (exercise).

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

$$f \in \bigtimes T \ \equiv \ \mathcal{D}\, f = \mathcal{D}\, T \wedge \forall\, x : \mathcal{D}\, f \cap \mathcal{D}\, T \,.\, f\, x \in T\, x \tag{14}$$

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

4. EXAMPLES I: SYSTEMS THEORY

Analysis: calculation replacing syncopation
We show how traditional proofs rendered tediuos by syncopation [20] are done calculationally. The example is *adjacency* [15]. Since predicates (of type $\mathbb{R} \to \mathbb{B}$) yield more elegant formulations than sets (of type $\mathcal{P}\,\mathbb{R}$), we define the predicate transformer $\mathsf{ad} : (\mathbb{R} \to \mathbb{B}) \to (\mathbb{R} \to \mathbb{B})$ and the predicates open and closed both of type $(\mathbb{R} \to \mathbb{B}) \to \mathbb{B}$, by

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

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

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

An example about transform methods
We show how formally correct use of functionals, in particular avoiding common defective notations like $\mathcal{F}\,\{f(t)\}$ and writing $\mathcal{F}\,f\,\omega$ instead, enables formal calculation. In

$$\begin{aligned} \mathcal{F}\,f\,\omega &= \textstyle\int_{-\infty}^{+\infty} e^{-j\cdot\omega\cdot t} \cdot f\,t \cdot \mathrm{d}\,t \\ \mathcal{F}'\,g\,t &= \frac{1}{2\cdot\pi} \cdot \textstyle\int_{-\infty}^{+\infty} e^{j\cdot\omega\cdot t} \cdot g\,\omega \cdot \mathrm{d}\,\omega \end{aligned}$$

bindings are clear and unambiguous. The example formalizes Laplace transforms via Fourier transforms. We assume some familiarity with the usual informal treatments.

Given $\ell_{\_} : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ with $\ell_\sigma\,t = (t < 0)\,?\,0 \mathbin{\mathsf{i}} e^{-\sigma\cdot t}$, we define the Laplace-transform $\mathcal{L}\,f$ of a function f by:

$$\mathcal{L}\,f\,(\sigma + j\cdot\omega) \quad = \quad \mathcal{F}\,(\ell_\sigma \mathbin{\hat{\cdot}} f)\,\omega \qquad (15)$$

for real σ and ω, with σ such that $\ell_\sigma \mathbin{\hat{\cdot}} f$ has a Fourier transform. With $s := \sigma + j\cdot\omega$ we obtain $\mathcal{L}\,f\,s = \int_0^{+\infty} f\,t \cdot e^{-s\cdot t} \cdot \mathrm{d}\,t$.

The converse $\mathcal{L}'$ is specified by $\mathcal{L}'\,(\mathcal{L}\,f)\,t = f\,t$ for all $t \geq 0$ (weakened where $\ell_\sigma \mathbin{\hat{\cdot}} f$ is discontinous). For such t,

$$\begin{aligned} & \mathcal{L}'\,(\mathcal{L}\,f)\,t \\ = \;& \langle \text{Specific.} \rangle && f\,t \\ = \;& \langle a = 1 \cdot a \rangle && e^{\sigma\cdot t} \cdot \ell_\sigma\,t \cdot f\,t \\ = \;& \langle \text{Defin. } \hat{} \rangle && e^{\sigma\cdot t} \cdot (\ell_\sigma \mathbin{\hat{\cdot}} f)\,t \\ = \;& \langle \text{Weaken.} \rangle && e^{\sigma\cdot t} \cdot \mathcal{F}'\,(\mathcal{F}\,(\ell_\sigma \mathbin{\hat{\cdot}} f))\,t \\ = \;& \langle \text{Defin. } \mathcal{F}' \rangle && e^{\sigma\cdot t} \cdot \tfrac{1}{2\cdot\pi} \cdot \textstyle\int_{-\infty}^{+\infty} \mathcal{F}\,(\ell_\sigma \mathbin{\hat{\cdot}} f)\,\omega \cdot e^{j\cdot\omega\cdot t} \cdot \mathrm{d}\,\omega \\ = \;& \langle \text{Defin. } \mathcal{L} \rangle && e^{\sigma\cdot t} \cdot \tfrac{1}{2\cdot\pi} \cdot \textstyle\int_{-\infty}^{+\infty} \mathcal{L}\,f\,(\sigma + j\cdot\omega) \cdot e^{j\cdot\omega\cdot t} \cdot \mathrm{d}\,\omega \\ = \;& \langle \text{Factor} \rangle && \tfrac{1}{2\cdot\pi} \cdot \textstyle\int_{-\infty}^{+\infty} \mathcal{L}\,f\,(\sigma + j\cdot\omega) \cdot e^{(\sigma + j\cdot\omega)\cdot t} \cdot \mathrm{d}\,\omega \\ = \;& \langle s := \sigma + j\cdot\omega \rangle && \tfrac{1}{2\cdot\pi\cdot j} \cdot \textstyle\int_{\sigma - j\cdot\infty}^{\sigma + j\cdot\infty} \mathcal{L}\,f\,s \cdot e^{s\cdot t} \cdot \mathrm{d}\,s \end{aligned}$$

Characterization and properties of systems

General Signals over a value space A are functions of type $\mathcal{S}_A$ with $\mathcal{S}_A = \mathbb{T} \to A$ for some time domain $\mathbb{T}$.

A systems is a function $s : \mathcal{S}_A \to \mathcal{S}_B$. The response of s to input signal $x : \mathcal{S}_A$ at time $t : \mathbb{T}$ is $s\,x\,t$, read $(s\,x)\,t$.

Characteristics Let $s : \mathcal{S}_A \to \mathcal{S}_B$. Then s is *memoryless* iff $\exists\,f_{\_} : \mathbb{T} \to A \to B \,.\, \forall\,x : \mathcal{S}_A \,.\, \forall\,t : \mathbb{T} \,.\, s\,x\,t = f_t\,(x\,t)$.

Let $\mathbb{T}$ be additive, and the *shift* function $\sigma_{\_}$ be defined by $\sigma_\tau\,x\,t = x\,(t + \tau)$ for any t and τ in $\mathbb{T}$ and any signal x. Then system s is *time-invariant* iff $\forall\,\tau : \mathbb{T} \,.\, s \circ \sigma_\tau = \sigma_\tau \circ s$.

A system $s : \mathcal{S}_\mathbb{R} \to \mathcal{S}_\mathbb{R}$ is *linear* iff for all $(x, y) : \mathcal{S}_\mathbb{R}^2$ and $(a, b) : \mathbb{R}^2$ we have $s\,(a \mathbin{\vec{\cdot}} x \mathbin{\hat{+}} b \mathbin{\vec{\cdot}} y) = a \mathbin{\vec{\cdot}} s\,x \mathbin{\hat{+}} b \mathbin{\vec{\cdot}} s\,y$. Equivalently, extending s to $\mathcal{S}_\mathbb{C} \to \mathcal{S}_\mathbb{C}$ in the evident way, the system s is *linear* iff $\forall\,z : \mathcal{S}_\mathbb{C} \,.\, c : \mathbb{C} \,.\, s\,(c \mathbin{\vec{\cdot}} z) = c \mathbin{\vec{\cdot}} s\,z$. A system is LTI iff it is both linear and time-invariant.

Response of LTI systems Define the parametrized exponential $\mathsf{E}_{\_} : \mathbb{C} \to \mathbb{T} \to \mathbb{C}$ by $\mathsf{E}_c\,t = e^{c\cdot t}$. Then we have:

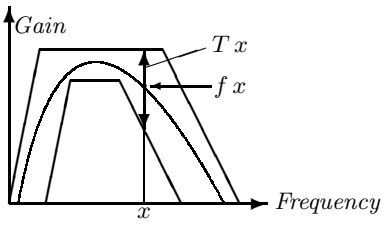
THEOREM: if s is LTI then $s\,\mathsf{E}_c = s\,\mathsf{E}_c\,0 \mathbin{\vec{\cdot}} \mathsf{E}_c$.

Proof: we calculate $s\,\mathsf{E}_c\,(t+\tau)$ to exploit all properties.

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

Substituting $t := 0$ yields $s\,\mathsf{E}_c\,\tau = s\,\mathsf{E}_c\,0 \cdot \mathsf{E}_c\,\tau$ or, using $\vec{}$, $s\,\mathsf{E}_c\,\tau = (s\,\mathsf{E}_c\,0 \mathbin{\vec{\cdot}} \mathsf{E}_c)\,\tau$, so $s\,\mathsf{E}_c = s\,E_c\,0 \mathbin{\vec{\cdot}} \mathsf{E}_c$ by function equality. The $\langle$Property $\mathsf{E}_c\rangle$ is $\sigma_\tau\,\mathsf{E}_c = \mathsf{E}_c\,\tau \mathbin{\vec{\cdot}} \mathsf{E}_c$ (easy). Note that this proof uses only the essential hypotheses.

Tolerances on specifications
Our first motivation for designing $\times$ was formalizing the concept of *tolerance* for functions, based on a common convention for specifying frequency/gain characteristics:



Clearly, with $T\,x$ specifying the desired interval for every x, the functions f satisfying $f \in \times T$ are precisely the desired ones. Next we show other uses of the same operator.

5. EXAMPLES II: COMPUTING SCIENCE

From data structures to query languages

Records as in PASCAL [14] are expressed by $\times$ as *functions* whose domain is a set of field labels (an *enumeration type*). Example: with field names $\mathtt{name}$ and $\mathtt{age}$,

$$\mathit{Person} := \times\,(\mathtt{name} \mapsto \mathbb{A}^* \cup \mathtt{age} \mapsto \mathbb{N})$$

defines a function type such that $\mathit{person} : \mathit{Person}$ satisfies $\mathit{person}\,\mathtt{name} \in \mathbb{A}^*$ and $\mathit{person}\,\mathtt{age} \in \mathbb{N}$. Obviously, by defining $\mathsf{record}\,F = \times\,(\bigcup F)$ ($\bigcup$: elastic extension of $\cup$), one can also write $\mathit{Person} := \mathsf{record}\,(\mathtt{name} \mapsto \mathbb{A}^*, \mathtt{age} \mapsto \mathbb{N})$.

Trees are functions whose domains are *branching structures*, i.e., sets of sequences describing the path from the root to a leaf in the obvious way (for any branch labeling). Other structures are covered similarly

Relational databases The following record type $\mathsf{record}\,(\mathsf{code} \mapsto \mathit{Code}, \mathsf{name} \mapsto \mathbb{A}^*, \mathsf{inst} \mapsto \mathit{Staff}, \mathsf{prrq} \mapsto \mathit{Code}^*)$ specifies the type of tables of the form

Code	Name	Instructor	Prerequisites
CS100	Elements of logic	R. Barns	none
MA115	Basic Probability	K. Jason	MA100
CS300	Formal Methods	R. Barns	CS100, EE150
...	...	...	

Generic functionals subsume all usual query-operators:

For the *selection*-operator (σ): $\sigma\,(S,P) = S \downarrow P$.

For *projection* (π): $\pi\,(S,F) = \{r \rceil F \mid r : S\}$.

For the *join*-operator ($\bowtie$): $S \bowtie T = S \otimes T$.

Here $\otimes$ is the generic *function type merge* operator, defined as in [8] by $S \otimes T = \{s \cup t \mid (s,t) : S \times T \wedge s \,©\, t\}$. Note that $\otimes$ is associative, although $\cup$ is not (exercise).

Formal semantics of programming languages
We show how the functional predicate calculus unifies the methodology for analysis (the ad example) and semantics.

The *state* s is the tuple made of the program variables, and $\mathbf{S}$ its type. We let $\backprime s$ denote the state before and s' after executing a command. This allows referring to different states in one equation. We write $s \bullet e$ for $s : \mathbf{S} \,.\, e$.

Program equations If C is the set of commands, $\mathsf{R} : C \to \mathbf{S}^2 \to \mathbb{B}$ and $\mathsf{T} : C \to \mathbf{S} \to \mathbb{B}$ are defined such that the effect of a command c can be described by two equations: $\mathsf{R}\,c\,(\backprime s, s')$ for state change and $\mathsf{T}\,c\,\backprime s$ for termination. We sometimes use s for $\backprime s$, writing $\mathsf{R}\,c\,(s,s')$ and $\mathsf{T}\,c\,s$. An example is Dijkstra's *guarded command* language [11].

Command c	State change $\mathsf{R}\,c\,(s,s')$
$v := e$	$s' = s[^v_e]$
$c' \,;\, c''$	$\exists\, t \bullet \mathsf{R}\,c'\,(s,t) \wedge \mathsf{R}\,c''\,(t,s')$
`if` [] $i : I \,.\, b_i$ `->` c_i' `fi`	$\exists\, i : I \,.\, b_i \wedge \mathsf{R}\,c_i'\,(s,s')$

Command c	Termination $\mathsf{T}\,c\,s$
$v := e$	1
$c' \,;\, c''$	$\mathsf{T}\,c'\,s \wedge \forall\, t \bullet \mathsf{R}\,c'\,(s,t) \Rightarrow \mathsf{T}\,c''\,t$
`if` [] $i : I \,.\, b_i$ `->` c_i' `fi`	$\exists\, b \wedge \forall\, i : I \,.\, b_i \Rightarrow \mathsf{T}\,c_i'\,s$

For `skip`: $\mathsf{R}\,\texttt{skip}\,(s,s') \equiv s' = s$ and $\mathsf{T}\,\texttt{skip}\,s \equiv 1$. For `abort`: $\mathsf{R}\,\texttt{abort}\,(s,s') \equiv 0$ and $\mathsf{T}\,\texttt{abort}\,s \equiv 0$. The loop `do` b `->` c' `od` stands for `if` $\neg b$ `->` `skip` [] b `->` $(c' \,;\, c)$ `fi` by definition, where c is the command itself.

Hoare semantics Let the state before and after executing c satisfy a (*antecondition*) and p (*postcondition*) respectively. Since all that is known about $\backprime s$ and s' is $a[^s_{\backprime s}]$ and $\mathsf{R}\,c\,(\backprime s, s')$, this must imply $p[^s_{s'}]$. This is the intuition behind the definitions of the following correctness criteria:

Partial: $\{a\}\ c\ \{p\} \equiv \forall\, \backprime s \bullet \forall\, s' \bullet a[^s_{\backprime s}] \wedge \mathsf{R}\,c\,(\backprime s, s') \Rightarrow p[^s_{s'}]$

Termination: $\mathit{Term}\,c\,a \equiv \forall\, s \bullet a \Rightarrow \mathsf{T}\,c\,s$

Total: $[a]\ c\ [p] \equiv \{a\}\ c\ \{p\} \wedge \mathit{Term}\,c\,a$

Calculating Dijkstra semantics We define the weakest liberal antecondition operator wla and the weakest antecondition operator wa by $\{a\}\ c\ \{p\} \equiv \forall\, s \bullet a \Rightarrow \mathsf{wla}\,c\,p$ and $[a]\ c\ [p] \equiv \forall\, s \bullet a \Rightarrow \mathsf{wa}\,c\,p$ (evident). To obtain explicit formulas, we calculate $[a]\ c\ [p]$ into this shape.

$$
\begin{array}{ll}
& \forall\, \backprime s \bullet \forall\, s' \bullet a[^s_{\backprime s}] \Rightarrow (\mathsf{R}\,c\,(\backprime s, s') \Rightarrow p[^s_{s'}]) \wedge \mathsf{T}\,c\,\backprime s \\
\equiv & \langle \text{Ldst. } \Rightarrow/\forall \rangle \quad \forall\, \backprime s \bullet a[^s_{\backprime s}] \Rightarrow \forall\, s' \bullet (\mathsf{R}\,c\,(\backprime s, s') \Rightarrow p[^s_{s'}]) \wedge \mathsf{T}\,c\,\backprime s \\
\equiv & \langle \text{Pdst. } \wedge/\forall \rangle \quad \forall\, \backprime s \bullet a[^s_{\backprime s}] \Rightarrow \forall\, (s' \bullet \mathsf{R}\,c\,(\backprime s, s') \Rightarrow p[^s_{s'}]) \wedge \mathsf{T}\,c\,\backprime s \\
\equiv & \langle \backprime s \text{ for } s \rangle \qquad \forall\, s \bullet a \Rightarrow \forall\, (s' \bullet \mathsf{R}\,c\,(s,s') \Rightarrow p[^s_{s'}]) \wedge \mathsf{T}\,c\,s
\end{array}
$$

Note the similarity with the ad-calculations. We proved: $\mathsf{wla}\,c\,p \equiv \forall\, s' \bullet \mathsf{R}\,c\,(s,s') \Rightarrow p[^s_{s'}]$ and $\mathsf{wa}\,c\,p \equiv \mathsf{wla}\,c\,p \wedge \mathsf{T}\,c\,s$.

Substituting the program equations for the various constructs, calculation in our predicate calculus yields [9]

$$
\begin{array}{rcl}
\mathsf{wa}\,[\![v := e]\!]\,p & \equiv & p[^v_e] \\
\mathsf{wa}\,[\![c' \,;\, c'']\!]\,p & \equiv & \mathsf{wa}\,c'\,(\mathsf{wa}\,c''\,p) \\
\mathsf{wa}\,[\![\texttt{if}\ [] \ i : I \,.\, b_i\ \texttt{->}\ c_i'\ \texttt{fi}]\!]\,p & \equiv & \exists\, b \wedge \forall\, i : I \,.\, b_i \Rightarrow \mathsf{wa}\,c_i'\,p \\
\mathsf{wa}\,[\![\texttt{do}\ b\ \texttt{->}\ c'\ \texttt{od}]\!]\,p & \equiv & \exists\, n : \mathbb{N} \,.\, w^n\,(\neg b \wedge p) \\
\text{defining } w \text{ by} \quad w\,q & \equiv & (\neg b \wedge p) \vee (b \wedge \mathsf{wa}\,c'\,q) \quad .
\end{array}
$$

6. EXAMPLES III: COMMON ASPECTS

Automata theory is a classical common ground between computing and systems theory. Yet, even here formalization yields unification and new insights. The example is sequentiality (capturing non-anticipatory behavior) and the derivation of properties by predicate calculus.

Preliminaries For any set A we define A^n by $A^n = \square\, n \to A$ where $\square\, n = \{m : \mathbb{N} \mid m < n\}$ for $n : \mathbb{N}$ or $n := \infty$, e.g., $(0,1,1,0) \in \mathbb{B}^4$. Also, $A^* = \bigcup\, n : \mathbb{N} \,.\, A^n$ (lists). Concatenation is $+\!\!+$, e.g., $(0,7,e) +\!\!+ (3,d) = 0,7,e,3,d$. Also, $x \prec\!\!- a = x +\!\!+ \tau\, a$. Next we consider systems $s : A^* \to B^*$.

Causal systems We define *prefix ordering* $\leq$ on A^* by $x \leq y \ \equiv\ \exists\, z : A^* \,.\, y = x +\!\!+ z$, and similarly for B^*. System s is *sequential* iff $x \leq y \Rightarrow s\,x \leq s\,y$. This captures the intuitive notion of causal (better: "non-anticipatory") behavior. Function $r : (A^*)^2 \to B^*$ is a *residual behavior* (rb) function for s iff $s\,(x +\!\!+ y) = s\,x +\!\!+ r\,(x,y)$. We show:

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

Proof: we start from the sequentiality side.

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

This completes the proof. Remarkably, the definition of $+\!\!+$ is used nowhere, illustrating the power of abstraction.

The last step uses the *function comprehension* axiom: $\forall\, (x : X \,.\, \exists\, y : Y \,.\, R\,(x,y)) \ \equiv\ \exists\, f : X \to Y \,.\, \forall\, x : X \,.\, R\,(x, f\,x)$ for any relation $R : X \times Y \to \mathbb{B}$.

Derivatives and primitives This framework leads to the following. An rb function is unique (exercise). We define the *derivative* operator D on sequential systems by $s\,(x \prec\!\!- a) = s\,x +\!\!+ \mathrm{D}\,s\,(x \prec\!\!- a)$, so $\mathrm{D}\,s\,(x \prec\!\!- a) = r\,(x, \tau\, a)$ where r is the rb function of s, and by $D\,s\,\varepsilon = \varepsilon$.

Primitivation I is defined for any $g : A^* \to B^*$ by $\mathrm{I}\,g\,\varepsilon = \varepsilon$ and $\mathrm{I}\,g\,(x \prec\!\!- a) = \mathrm{I}\,g\,x +\!\!+ g\,(x +\!\!+ a)$. Properties are shown next, with a striking analogy from analysis.

$s\,(x \prec\!\!- a) = s\,x +\!\!+ \mathrm{D}\,s\,(x \prec\!\!- a)$	$s\,x = s\,\varepsilon +\!\!+ \mathrm{I}\,(\mathrm{D}\,s)\,x$
$f\,(x+h) \approx f\,x + \mathrm{D}\,f\,x \cdot h$	$f\,x = f\,0 + \mathrm{I}\,(\mathrm{D}\,f)\,x$

Of course, in the second row, D is the derivation operator from analysis, and $\mathrm{I}\,g\,x = \int_0^x g\,y \cdot \mathrm{d}\,y$ for integrable g. Moreover, $f\,x + \mathrm{D}\,f\,x \cdot h$ is only approximate.

This and other differences confirm the observation in [18] that automata are easier than real functions.

Finally, $\{(y : A^* \,.\, r\,(x,y)) \mid x : A^*\}$ is the *state space*.

7. 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, not only yields a notational and methodological unification of computing science and systems theory, but also of a large part of mathematics.

Apart from the obvious scientific ramifications, the formalism provides a unified basis for education in ECE (Electrical and Computer Engineering), as advocated in [17].

The difficulties should be recognized as well. First, although calculational logic is easier than classical formal logic, the de-emphasis on proofs in education has caused students to find logic increasingly difficult [1, 19]. Second, conservatism of colleagues may even be a larger problem [1, 18, 19, 21], and there are known cases of censorship.

Yet, the wide scope of the formalism demonstrated in these few pages, with only minor gaps left for the reader to fill, provides ample evidence for the long-term advantages.

8. 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] 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`

[3] Raymond T. Boute, "A heretical view on type embedding", *ACM Sigplan Notices 25*, pp. 22–28 (Jan. 1990)

[4] Raymond T. Boute, *Funmath illustrated: A Declarative Formalism and Application Examples*. Declarative Systems Series No. 1, Computing Science Institute, University of Nijmegen (July 1993)

[5] 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)

[6] 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)

[7] Raymond Boute, *Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I: Basic Mathematics*. Course text, Ghent University (2002)

[8] Raymond T. Boute, "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons and Johan Jeuring, eds., *Generic Programming*, pp. 89–119, Kluwer (2003)

[9] Raymond T. Boute, "Calculational semantics: deriving programming theories from equations by functional predicate calculus", Technical note B2004/02, INTEC, Universiteit Gent (2004) (submitted for publication to *ACM TOPLAS*)

[10] Ronald N. Bracewell, *The Fourier Transform and Its Applications*, 2nd ed, McGraw-Hill (1978)

[11] Edsger W. Dijkstra and Carel S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin (1990)

[12] David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, Berlin (1993)

[13] David Gries, "The need for education in useful formal logic", *IEEE Computer 29*, 4, pp. 29–30 (April 1996)

[14] Kathleen Jensen and Niklaus Wirth, *PASCAL User Manual and Report*. Springer-Verlag, Berlin (1978)

[15] Serge Lang, *Undergraduate Analysis*. Springer-Verlag, Berlin (1983)

[16] Leslie Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education Inc. (2002)

[17] 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/`

[18] 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/`

[19] Rex Page, *BESEME: Better Software Engineering through Mathematics Education*, project presentation
`http://www.cs.ou.edu/~beseme/besemePres.pdf`

[20] 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/Practical_Foundations/html`

[21] 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/`