Declarative specification and calculation in view of software evolution[*]

Raymond T. Boute
INTEC, Ghent University, Belgium
boute@intec.rug.ac.be

Abstract

Software is always in evolution, not only as an engineering discipline, but also during system life cycles. Specification plays a central role in guiding this process in all phases from design, via use and maintenance, to future prospects. As opposed to program-like descriptions, a declarative approach yields more generality, compactness and insensitivity to paradigmatic, implementation and other changes.

The crucial characteristics of a formalism are not only its expressive qualities but, even more importantly, the support it provides for symbolic reasoning (property derivation, verification). Substantial leverage is obtained by starting from concepts that have proven their value in a wide variety of mathematics.

Whereas most formalisms in computing stem from formal logic, we present an approach that is rooted in classical applied (engineering) mathematics, yet captures and even generalizes the most important concepts normally associated with theoretical computer science.

For historical reasons, we start with the problem area in which our approach originated, namely unifying discrete and continuous mathematical models in systems engineering, with signal flow systems as a running example. This has led to a collection of generic functionals*, originally meant to derive realizations from abstract specifications, but whose generalization, with particular attention to the domains of the result functions, captures a wider range, computing included.*

These functionals also support an elegant functional formulation of predicate logic, with convenient algebraic laws for calculational reasoning in point-wise and point-free style, and handling type information in a pivotal way. We have found this calculus directly applicable in all investigated areas, ranging from mathematical analysis to software engineering. Such wide coverage ensures the ability to cope with evolution in software and paradigms over a long time period.

[*]Prepared for the 2002/01/18 meeting of the FWO Research Network "Foundations of Software Evolution".

0 Motivation and overview

Software systems and the paradigms underlying their design are incessantly evolving. Guiding this process requires a level of abstraction that transcends incidental aspects such as implementation details, programming language particularities, and the software engineering paradigm of the day.

Superficially, modern software engineering tools seem to reduce systems design to menu selection and mouse clicking. However, all but the simplest systems also need (a) *specifications* that are *concise* (sufficiently abstract, no extraneous detail), *precise* (complete, unambiguous), and *clear*, especially for humans; (b) methods for *symbolic reasoning* about issues such as the relation between specification and realization, and the extraction of properties of interest. The need for these elements also extends to the tools themselves!

Program-like code is often advertised as a form of specification. However, this is illusory, since code is *algorithmic*, containing particularities necessary for executability, but irrelevant at the abstract level inherent in the concept of specification. Documentation is expected to be more compact, human-centric and problem-oriented than the system it describes. More importantly, program code (at any level) is very ill-suited for reasoning about the specified systems.

Obviously the language of mathematics, being designed for human use and reasoning, is far superior for the stated purpose. In hardware design, electronics and other classical engineering disciplines, this is the basic formalism. Therefore it is unfortunate that languages such as VHDL, intended to cope with the long-term evolutionary nature of complex hardware systems, are patterned after programming languages, clearly motivated by communication with computers rather than among humans. The idea that, for instance, hardware engineers are to "think algorithmically' is clearly retrograde, and all the more ironic since, in the very area of programming, much effort is devoted to shifting away from the traditional imperative paradigm.

Functional and logical programming languages offer a distinctly higher degree of abstraction, yet are only an intermediate step. Indeed, they still fall short of being properly declarative [9, 12], meaning: suitable for specification and reasoning, free from operational concerns imposed by the language or the implementation.

The formalisms of mathematics provide a good starting point for declarativity, but form a very heterogeneous mixture of very well-designed parts in algebra and analysis (due to Descartes and Leibniz) to very ad hoc designs in discrete mathematics, logic and computer science, rare exceptions being the calculational style advocated by Dijkstra [4], Gries [7] and others.

As observed by Reynolds [13], *In designing a programming language, the central problem is to organize a variety of concepts in a way which exhibits uniformity and generality. Substantial leverage can be gained in attacking this problem if the concepts can be defined concisely within a framework which has already proven its ability to impose uniformity and generality upon a wide variety of mathematics.* This quote is all the more pertinent to declarative languages.

The results reported here stem from concerns that are not directly related to software and programming, but rather to the design of a formalism[0] aimed at unifying the continuous and discrete mathematical models in applied mathematics and engineering, in particular communications and automatic control, with each other and with the discrete concepts in computing science.

Since one of the purposes of this meeting is to make members of the software evolution community more familiar with each others' research interests, we reflect the historical development by taking the original problem area as a running example, namely the transformation of specifications into signal flow realizations.

At first, this may seem a rather concrete and specialized topic, but it turns out to require all features of a general-purpose declarative formalism, as becomes soon apparent to anyone with a taste for abstraction. Moreover, the same concepts yield an elegant functional formulation of predicate logic, which in turn is perhaps the most generally applicable formalism throughout mathematics and software engineering [11].

Therefore, we start with *signal flow systems* (Sec. 1), using certain operators that will appear familiar from classical applied mathematics, and augmenting them with a few new ones to support transformations that are awkward or impossible to express in other formalisms. The operators are generalized to *generic functionals* (Sec. 2), whose generic character stems from the special attention given to the domains of the functions, and also provides the basis for reformulating predicate logic in a functional framework that includes types and supports calculational reasoning (Sec. 3).

[0]By *formalism* we do not mean just a language or notation but also, and even primarily, a system of formal calculation and reasoning in a precise and convenient way by humans.

1 Functionals for transformation

1.0 Functional Mathematics

Functional Mathematics is the principle of (re)defining mathematical objects, whenever feasible, as functions. This has proved most useful especially where it is not (yet) commonplace. Apart from the conceptual virtue of uniformity, a major advantage is that a collection of general-purpose operators over functions (*generic functionals*) becomes widely shared by objects of otherwise disparate types (intrinsic polymorphism). This also justifies investing considerable effort in designing them judiciously, especially w.r.t. types, as shown later.

Here we present an extended example, namely the definition of sequences (including tuples, lists etc.), which has wide ramifications since it constitutes either a common ground or a serious gap between discrete and "continuous" mathematics, depending on whether or not sequences are defined as first-class functions.

We define all these structures as functions, in the sense that $(a, b)\,0 = a$ and $(a, b)\,1 = b$ etc. Whereas this step is intuitively trivial, browsing through the literature [6, 10, 14, 16] reveals that these objects are handled more often than not as entirely or subtly distinct from functions, and even in the few exceptions, the functional characteristics are left mostly unexploited.

For instance, one rarely (if ever) finds inverses of sequences as in $(a, b, c, d)^{-}\,c = 2$, or composition, as in $(0, 3, 5, 7) \circ (2, 3, 1) = 5, 7, 3$ and $f \circ (x, y) = f\,x, f\,y$, or transposition, as in $(f, g)^T x = f\,x, g\,x$, which will be seen to be universally useful once discovered.

Moreover, in most formal treatments, lists are defined recursively via a prefixing operator (say, *cons*), and not as functions, for instance: [] is a list and, if x is a list, so is $cons\,a\,x$. Indexing is then achieved via a separate operator, defined recursively on the structure, as in $ind\,(cons\,a\,x)\,0 = a$ and $ind\,(cons\,a\,x)\,(n+1) = ind\,x\,n$.

Our functional definition takes indexing as the basis, and hence covers infinite sequences as well (their domain is $\mathbb{N}$). Pairs like x, y in $f\,(x, y)$ are no exception. This makes every function 'of several arguments' essentially a functional of one argument, which is a function and thereby shares in all generic functionals. Note that this view does *not* invalidate any of the traditional calculation rules, it only adds more powerful ones.

A side remark: more generally, such conservational properties explain why *Funmath* [3], a formalism based on functional mathematics and used in this paper, so much resembles the usual mathematical conventions, except where the extra possibilites are exploited.

Henceforth, tuples and sequences are functions with domain $\{k : \mathbb{N} \mid k < n\}$ for $n : \mathbb{N}$ or $n := \infty$, written $\square\, n$. Operators for tuples and sequences, like $\#$ (length), $+\!\!+$ (concatenation), $>\!\!-$ (prefixing), σ (shift) are defined as functionals, e.g., $(a >\!\!- x)\, n = (n = 0)\,?\, a \nmid x\,(n-1)$ and $\sigma\, x\, n = x\,(n+1)$. Conditional expressions of the form $c\,?\, b \nmid a$ can be seen as syntactic sugar for $(a, b)\, c$.

In this discussion, we singled out sequences because of their central role in our main testing ground, namely unifying discrete systems (whose inputs and outputs are sequences) with their analog counterparts, where signals are modelled as functions to begin with.

1.1 Point-free formulations

As a running example, we consider *signal flow systems*, namely assemblies of interconnected components whose dynamical behavior is modelled by functionals mapping input signals to output signals. We are interested in deriving signal flow systems realizing specified functions.

The simplest basic blocks used initially are memoryless devices realizing arithmetic operations and, as memory devices, latches for the discrete case and integrators for the the continuous case. We model the arithmetic blocks as “abstract operational amplifiers” for instantaneous values, and make this explicit by expressing, for instance, the sum of two signals x and y as $x \mathbin{\widehat{+}} y$, with $(x \mathbin{\widehat{+}} y)\, t = x\, t + y\, t$. In control and communications theory, no hat $(\widehat{\ })$ is used, writing just $(x + y)\, t = x\, t + y\, t$ by overloading $+$. We will see it pays off making $\widehat{\ }$ explicit as a fully-fledged functional, called *direct extension*, and generalized later.

For our examples, it suffices considering the *synchronous* and *discrete* case, and the time variable will usually be written n (of type $\mathbb{N}$). A *latch* is one-cell device storing values between two subsequent clock cycles and parametrized by an initial condition. It is polymorphic with respect to the value stored. Its *behaviour* D can be expressed formally for any initial condition a and input sequence x as $\mathrm{D}_a\, x\, n = (n = 0)\,?\, a \nmid x\,(n - 1)$ or, without the time variable n, as $\mathrm{D}_a\, x = a >\!\!- x$.

In Fig. 0, (a) and (b) represent these blocks as they appear in typical textbooks on communications or automatic control, and (c) and (d) as they appear on screen in a graphical language like LabVIEW [2], designed for instrumentation purposes.

For *structurally* describing an assembly of components, we do not want a variable representing time in our expressions, since time is not a structural feature. If a signal representing time is needed, it is generated

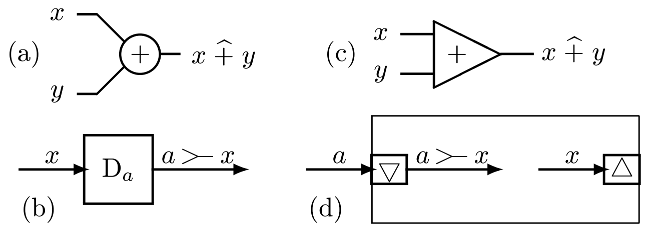

Figure 0: Typical basic building blocks

by a *time base* (a ramp generator in an oscilloscope, a counter in a digital system), which is structural.

Given a behavioural *specification* as a mapping from input to output signals, transformational *design* amounts to eliminating the time variable (and possibly further transformation) to obtain an expression whose form also has a structural interpretation [3].

In a more general mathematical context, the corresponding objective is calculating with functions algeraically without referring to points in their domain, which is called a *point-free* formulation. The functionals to support this will be introduced as we proceed.

1.2 A transformation example

Consider the following recursive specification. Assuming set A and $a : A$ and $g : A \to A$ given, we define

$$\textbf{def}\ f : \mathbb{N} \to A\ \textbf{with}\ f\, n = (n = 0)\,?\, a \nmid g\,(f\,(n-1)) \quad (0)$$

This equation be transformed calculationally as follows

$$\begin{aligned} f\, n &= \langle \text{Def. } f \rangle && (n = 0)\,?\, a \nmid g\,(f\,(n-1)) \\ &= \langle \text{Def. } \circ \rangle && (n = 0)\,?\, a \nmid (g \circ f)\,(n-1) \\ &= \langle \text{Def. D} \rangle && \mathrm{D}_a\,(g \circ f)\, n \\ &= \langle \text{Def. } \overline{} \rangle && \mathrm{D}_a\,(\overline{g}\ f)\, n \\ &= \langle \text{Def. } \circ \rangle && (\mathrm{D}_a \circ \overline{g})\, f\, n, \qquad (1) \end{aligned}$$

hence $f = (\mathrm{D}_a \circ \overline{g})\, f$ by function extensionality. Observe how function composition, i.e., $(f \circ g)\, x = f\,(g\, x)$ (ignoring types for now) moves n into the argument position to facilitate later elimination. We introduced $\overline{}$, defined by $\overline{g}\ x = g \circ x$, to provide direct extension for one-argument functions. The last step in (1) makes the expression interpretable structurally, since function composition structurally amounts to cascading, as shown in Fig. 1(a) for $h \circ g$. An expected algebraic property is $\overline{h \circ g} = \overline{h} \circ \overline{g}$.

Here we should mention that, since tuples are functions, $f \circ (a, b) = f\, a, f\, b$, yielding a quite different functional interpretation, as shown in Fig. 1(b) assuming x is a pair. Of course, all these interpretations correspond to the same abstract functional.

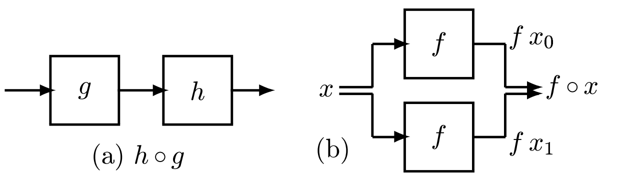

Figure 1: Structural interpretations of composition

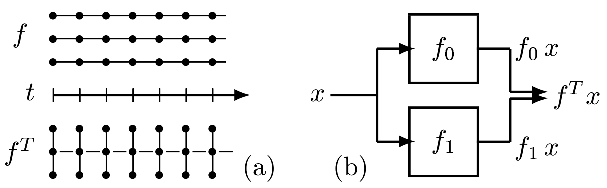

Figure 3: Structural interpretations of transposition

A realization of the fixpoint equation $f = (\mathrm{D}_a \circ \overline{g})\,f$ is shown in Fig. 2(a) as a "textbook" block diagram and in Fig. 2(b) as a LabVIEW on-screen wiring diagram.

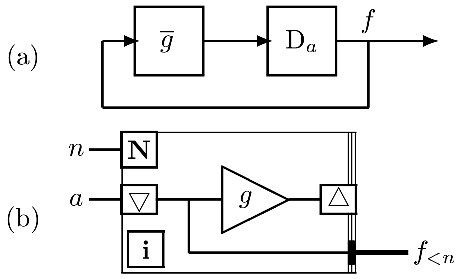

Figure 2: Signal flow realization of specification (0)

Since f is an infinite sequence, the length of the subsequence $f_{<n}$ to be generated by a working system as in Fig. 2(b) is specified by an extra parameter $n\,{:}\,\mathbb{N}_{\geq 1}$.

Of course, the transformation of equation (0) into the diagram of Fig. 2(a) is so simple that it could have been done 'on sight', but the point is that the auxiliary operators make it possible to formalize every little step of the process: without them, there is no handle, and one could only give a 'proof by inspection'.

Many situations require swapping the arguments of a higher-order function. A typical case is the transformation of a family[1] f of n signals into a single signal f^T with n-tuples as values satisfying $f^T t\, i = f\, i\, t$ at time t for all i in $\square\, n$, as illustrated in Fig. 3(a). This also subsumes the `zip` operator familiar from functional programming [1], viz. `zip[[a,b,c],[a',b',c']] = [[a,a'],[b,b'],[c,c']]`, taking lists as functions.

We call this operator ^{T} *transposition*, since it is a generalization of the well-known matrix concept to higher-order functions: for any family f of functions (not necessarily with discrete domain), we define $f^T y\, x = f\, x\, y$, again temporarily ignoring types.

Note that, tuples being functions, $(f,g)^T x = f\,x, g\,x$, yielding the structural interpretation of Fig. 3(b).

In a very real sense, composition and transposition are each other's dual, e.g., in the (untyped) lambda calculus, provided x is not free in M,

$$M \circ (\lambda x.N) = \lambda x.MN \quad \text{and} \quad (\lambda x.N)^T M = \lambda x.NM.$$

Together, comosition and transposition provide a generalization of direct extension (seen thus far only for functions with one and two arguments, counted the 'old' way) to functions with a tuple of any length for its argument. For any infix operator $\star$,

$$\begin{aligned}(f \mathbin{\widehat{\star}} f')\,x &= f\,x \star f'\,x = (\star)\,(f\,x, f'\,x) \\ &= (\star)\,((f,f')^T x) = ((\star) \circ (f,f'))^T x\end{aligned}$$

(hints omitted) and hence $f \mathbin{\widehat{\star}} f' = (\star) \circ (f,f')^T$. This makes it reasonable to define the generalized direct extension operator $\overset{<}{-}$ by

$$\overset{<}{g}\, h = g \circ h^T \tag{2}$$

for any function g whose argument is a function and any family h of functions.

Of course, much more can be said about transformations, but we have collected enough representative functionals, and we shall now make them generic.

2 Making the functionals generic

2.0 Conventions for functions

The notion of *function* is familiar, but since conventions in the literature are not uniform, we make ours explicit. A function is taken as a concept in its own right without identifying it with its set-theoretic representation via pairs. By definition, a function f is fully specified by its *domain* $\mathcal{D}\,f$ and its *mapping*, associating with every element x in $\mathcal{D}\,f$ a (unique) image $f\,x$.

Parentheses are used *only* for emphasis or for overruling precedence or affix conventions, *never* as part of some operator. Hence they are optional in $f(x)$ and in (a,b,c). Prefix operators have precedence over infix operators, so parentheses are optional in $(f\,x) + 1$ or $f\,(x) + 1$, but necessary in $f\,(x+1)$. For higher order functions, we write $f\,x\,y$ as a shorthand for $(f\,x)\,y$.

We also allow *partial application*: the application of an infix operator $\star$ to only one argument denotes a function on the other argument, viz., $(a\,\star)\,b = a{\star}b = (\star\,b)\,a$.

[1] In our functional framework, *family* is synonym with *function*, phrases like "a family of functions" being more appropriate than "a function-valued function".

Functions are *equal* iff their domains and mappings match. Formally, this amounts to Leibniz's principle

$$f = g \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x) \quad (3)$$

and function extensionality: using a fresh dummy x,

$$\frac{q \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x)}{q \Rightarrow f = g}. \quad (4)$$

As long as we have not yet elaborated quantifiers, we often specify a function f via a pair of axioms:

- a *domain axiom* of the form $x \in \mathcal{D}\, f \equiv x \in X \wedge p_x$
- a *mapping axiom* of the form $x \in \mathcal{D}\, f \Rightarrow q_{f,x}$

where x is a variable, X a set expression, p_x and $q_{f,x}$ propositions, where the subscripts are comments used (just once) to specify which of f and x may occur free.

An example is the *constant function specifier* $^\bullet$, defined for any set X and any e by the pair of axioms

$$\mathcal{D}\,(X^\bullet\, e) = X \quad \text{and} \quad x \in X \Rightarrow (X^\bullet\, e)\, x = e. \quad (5)$$

Why such a trivial example? We define *predicates* as *functions* taking only values 0 and 1. Our *quantifiers* are predicates over predicates: for any predicate P, $\forall\, P \equiv P = \mathcal{D}\, P^\bullet\, 1$ and $\exists\, P \equiv P \neq \mathcal{D}\, P^\bullet\, 0$ (see later).

If q has the explicit form $f\, x = e_x$, the function can be denoted by an *abstraction* $x : X \wedge p \,.\, e$, where $\wedge p$ is optional. More formally, the axioms for $x : X \wedge p \,.\, e$ are:

$$\begin{aligned} d \in \mathcal{D}\,(x : X \wedge p \,.\, e) &\equiv d \in X \wedge p^x_d \\ d \in \mathcal{D}\,(x : X \wedge p \,.\, e) &\Rightarrow (x : X \wedge p \,.\, e)\, d = e^x_d \end{aligned} \quad (6)$$

(for any d), where p^x_d denotes p with d properly substituted for any free occurrence of x. For instance, $n : \mathbb{Z} \,.\, 2 \cdot n$ doubles every natural number, and (5) can be written $X^\bullet\, e = x : X \,.\, e$ (choosing x not free in e).

The function *range* operator $\mathcal{R}$ is axiomatized by $y \in \mathcal{R}\, f \equiv \exists\, x : \mathcal{D}\, f \,.\, y = f\, x$, and the familiar $\rightarrow$ for function types by $f \in X \rightarrow Y \equiv \mathcal{D}\, f = X \wedge \mathcal{R}\, f \subseteq Y$.

2.1 Design criteria and method

The operators of interest are (generic) functionals, i.e, functions over functions. In functional mathematics, they are shared by many more kinds of objects than usual, and hence deserve judicious design to eliminate all unnecessary restrictions on their arguments.

For instance, the traditional definitions of $f \circ g$ require that $\mathcal{R}\, g \subseteq \mathcal{D}\, f$, in which case $\mathcal{D}\,(f \circ g) = \mathcal{D}\, g$. Similarly, the common inverse f^- requires f to be injective (not considering the variant where inverse images are subsets rather than elements of $\mathcal{D}\, f$).

In our design, we do not impose restrictions on the argument functions, but refine the domain of the result functions. These generalizations are *conservative*, i.e., if the traditional restriction is satisfied, our generalized operator coincides with the traditional one.

2.2 Some important generic functionals

All our functionals pertain to continuous as well as discrete mathematics, but most examples will be discrete.

The main transformation between the point-wise and point-free styles is the equality $f = x : \mathcal{D}\, f \,.\, f\, x$, obtained from (3), (4), (6). For instance, $P = x : \mathcal{D}\, P \,.\, P\, x$ and hence, again using (3), $\forall\, P \equiv \forall\, x : \mathcal{D}\, P \,.\, P\, x$.

The **filtering** operator ($\downarrow$) generalizes this as follows:

$$f \downarrow P \;=\; x : \mathcal{D}\, f \cap \mathcal{D}\, P \wedge P\, x \,.\, f\, x \quad (7)$$

for any function f and any predicate P. We often use f_P as shorthand for $f \downarrow P$, as in $f_{<n}$. This operator is also defined for sets by $x \in S_P \equiv x \in S \wedge P\, x$, giving convenient abbreviations like $\mathbb{R}_{\geq 0}$ a formal basis.

The **composition** operator ($\circ$) generalizes the familiar function composition: for *any* functions f and g,

$$\begin{aligned} x \in \mathcal{D}\,(f \circ g) &\equiv x \in \mathcal{D}\, g \wedge g\, x \in \mathcal{D}\, f \\ x \in \mathcal{D}\,(f \circ g) &\Rightarrow (f \circ g)\, x = f\,(g\, x). \end{aligned} \quad (8)$$

The conservational nature of this generalization is illustrated by the fact that, if the traditional requirement $\mathcal{R}\, g \subseteq \mathcal{D}\, f$ is satisfied, then $\mathcal{D}\,(f \circ g) = \mathcal{D}\, g$.

Since sequences are functions, $(0,3,5,7) \circ (2,3,1) = 5,7,3$ and $(0,3,5,7) \circ (2,3,5) = 5,7$, but note also that $(0,3,5,7) \circ (5,3,1) = (7,3) \circ (-1)$ (not a sequence).

Similarly, since $f \circ (x, y) = f\, x, f\, y$ (x and y in $\mathcal{D}\, f$), our $\circ$ subsumes the *map* operator @ from functional programming, viz., `f @ [x, y] = [f x, f y]`.

The **direct extension** operator ($\widehat{\;\;}$) is similarly designed such that, for any (infix) operator $\star$ and any functions f and g, the domain of $f \mathbin{\widehat{\star}} g$ contains exactly those values x for which the expression $f\, x \star g\, x$ does not contain any out-of-domain applications:

$$\begin{aligned} x \in \mathcal{D}\,(f \mathbin{\widehat{\star}} g) &\equiv x \in \mathcal{D}\, f \cap \mathcal{D}\, g \wedge (f\, x, g\, x) \in \mathcal{D}\,(\star) \\ x \in \mathcal{D}\,(f \mathbin{\widehat{\star}} g) &\Rightarrow (f \mathbin{\widehat{\star}} g)\, x = f\, x \star g\, x. \end{aligned} \quad (9)$$

For the **transposition** operator ($—^T$) with $f^T y\, x = f\, x\, y$, the simplest argument type is $A \rightarrow (B \rightarrow C)$ (given sets A, B, C). The image f^T of $f : A \rightarrow (B \rightarrow C)$ has type $B \rightarrow (A \rightarrow C)$ and property $(f^T)^T = f$. Note: one usually writes $A \rightarrow B \rightarrow C$ for $A \rightarrow (B \rightarrow C)$.

We want the argument of ^{T} to be *any* function family. In a liberal design, $\mathcal{D}\, f^T = \bigcup\, x : \mathcal{D}\, f \,.\, \mathcal{D}\,(f\, x)$ or, in point-free style, $\mathcal{D}\, f^T = \bigcup\,(\mathcal{D} \circ f)$. Elaboration is left as an exercise, since this is not our preference. Indeed recall from (2) that defining $\overset{<}{g}\, h = g \circ h^T$ to generalize (9) requires intersection, not union. This decision results in defining f^T for any family f of functions by

$$\begin{aligned} \mathcal{D}\, f^T &= \bigcap\,(x : \mathcal{D}\, f \,.\, \mathcal{D}\,(f\, x)) \\ y \in \mathcal{D}\, f^T &\Rightarrow x \in \mathcal{D}\, f \Rightarrow f^T y\, x = f\, x\, y \end{aligned} \quad (10)$$

or, in compact form, $f^T = y : \bigcap\,(\mathcal{D} \circ f) \,.\, x : \mathcal{D}\, f \,.\, f\, x\, y$.

3 Functional predicate calculus

3.0 Axioms

Recall that a *predicate* is a boolean-valued function. Here, the choice between ***false*** and ***true*** or 0 and 1 as boolean values is secondary. In a wider context (not discussed here) choosing 0 and 1 has many advantages.

We define the *quantifiers* $\forall$ and $\exists$ to be predicates over predicates. Informally, $\forall\, P$ means that P is the constant 1-valued predicate, and $\exists\, P$ means that P is *not* the constant 0-valued predicate. Hence the axioms

$$\forall\, P \equiv (P = \mathcal{D}\, P^{\bullet} 1) \text{ and } \exists\, P \equiv (P \neq \mathcal{D}\, P^{\bullet} 0). \quad (11)$$

These definitions are conceptually indeed as simple as they seem. Yet, they give rise to literally dozens of calculation rules, as is necessary in any predicate calculus for practical use [7]. This is typical for rich algebraic structures, often apparent even in basic mathematics.

3.1 Elastic operators and ramifications

Our quantifiers are examples of so-called *elastic operators*. These are functionals replacing the various kinds of ad hoc abstractors from common mathematics, e.g.,

$$\forall\, x\!:\!X \qquad \textstyle\sum_{i=m}^{n} \qquad \lim\limits_{x\to a}\,.$$

Elastic operators together with function abstraction (6) yield readily recognizable expressions such as

$$\forall\, x\!:\!X\,.\,P\,x \qquad \textstyle\sum\, i\!:\!m\,..\,n\,.\,x_i \qquad \mathsf{lim}\ (x\!:\!\mathbb{R}\,.\,f\,x)\,a$$

or, for less casual readers, point-free forms such as

$$\forall\, P \qquad \textstyle\sum\, x \qquad \mathsf{lim}\ f\,a$$

Expressions like $\forall\, x\!:\!\mathbb{R}\,.\,x^2 \geq 0$ obtain their familiar form and meaning, but also with a novel decomposition, e.g., into $\forall$ and $x\!:\!\mathbb{R}\,.\,x^2 \geq 0$, each being a *function* and, more importantly, additional calculation rules.

Perhaps most representative for the additional rules provided by the functional view is the smooth transition between point-free and point-wise expressions.

The fundamental importance of formulations of mathematical theories without variables is widely recognized in the more abstract branches [15]. However, as seen in Sec. 1, eliminating variables is also useful and sometimes even necessary in practical applications, which impose the extra requirement that one should not be confined to only one of these styles.

Our approach made it possible to reformulate predicate calculus as a calculus of *functions* which, to working mathematicians and engineers, is more familiar and calculation-friendly than the often ad hoc conventions in logic textbooks. The point-free style also conveys an elegant algebraic flavor to the calculation rules.

3.2 Derived calculation rules

Here we give a small sample of rules, sufficient to illustrate how the rest of them can be derived as an exercise.

A **first batch** are simple rules derived directly from the axioms (11) and function equality (3 and 4).

- $\forall\,(X^{\bullet} 1) \equiv 1$ and $\exists\,(X^{\bullet} 0) \equiv 0$
- $\forall\,\varepsilon \equiv 1$ and $\exists\,\varepsilon \equiv 0$
- For any non-constant P: $\forall\, P \equiv 0$ and $\exists\, P \equiv 1$

Here ε is the *empty* function or predicate with $\mathcal{D}\,\varepsilon = \emptyset$.

Illustrative of the algebraic equational style are the following theorems.

- *Duality*: $\forall\,(\overline{\neg}\ P) \equiv (\overline{\neg}\ \exists)\, P$
- *Meeting*: $\forall\, P \wedge \forall\, Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q)$. The converse is conditional: $\mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q) \Rightarrow \forall\, P \wedge \forall\, Q$.

A typical calculational proof for duality is

$$\begin{array}{lcl}
\forall\,(\overline{\neg}\ P) & & \\
\equiv & \langle \text{Def. } \forall \text{ (11)}, \mathcal{D}\,(\overline{\neg}\ P) = \mathcal{D}\, P\rangle & \overline{\neg}\ P = \mathcal{D}\, P^{\bullet} 1 \\
\equiv & \langle \overline{\neg}\ P = Q \equiv P = \overline{\neg}\ Q\rangle & P = \overline{\neg}\ (\mathcal{D}\, P^{\bullet} 1) \\
\equiv & \langle e \in \mathcal{D}\, g \Rightarrow \overline{g}\,(X^{\bullet} e) = X^{\bullet} (g\, e)\rangle & P = \mathcal{D}\, P^{\bullet} (\neg\, 1) \\
\equiv & \langle \neg\, 1 = 0, \text{ def. } \exists \text{ (11)}\rangle & \neg\ (\exists\, P) \\
\equiv & \langle x \in \mathcal{D}\,(\overline{g}\, f) \Rightarrow \overline{g}\, f\, x = g\,(f\, x)\rangle & \overline{\neg}\, \exists\, P
\end{array}$$

Using Feyen's convention, justifications are given between $\langle\rangle$. All are properties of generic functionals rather than quantification, and are left as simple exercises.

Other properties of constant predicates reveal the role of types; they are uncommon in logic textbooks

$$\forall\,(X^{\bullet} 0) \equiv X = \emptyset \quad \text{and} \quad \exists\,(X^{\bullet} 1) \equiv X \neq \emptyset$$

or, combined with the earlier properties,

$$\forall\,(X^{\bullet} x) \equiv x \vee X = \emptyset \quad \text{and} \quad \exists\,(X^{\bullet} x) \equiv x \wedge X \neq \emptyset$$

A general way to obtain many of properties of this kind are *case analysis* (a.) and *Shannon expansion* (b., c.).

a. $\forall\, P^v_0 \wedge \forall\, P^v_1 \Rightarrow \forall\, P$

b. $\forall\, P \equiv (v \wedge \forall\, P^v_1) \vee (\neg\, v \wedge \forall\, P^v_0)$

c. $\forall\, P \equiv (v \Rightarrow \forall\, P^v_1) \wedge (\neg\, v \Rightarrow \forall\, P^v_0)$

assuming v is a boolean variable in P. Similarly for $\exists$.

Important consequenses are *semidistributivity rules*:

- $\forall\,(x \mathbin{\overrightarrow{\wedge}} P) \equiv (x \wedge \forall\, P) \vee \mathcal{D}\, P = \emptyset$
- $\forall\,(x \mathbin{\overrightarrow{\Rightarrow}} P) \equiv x \Rightarrow \forall\, P$

- $\forall\,(P \overset{\leftharpoonup}{\Rightarrow} x) \equiv \exists\, P \Rightarrow x$

where $\overset{\rightharpoonup}{—}$ is the (right) half direct extension operator

$$x \overset{\rightharpoonup}{\star} f \quad = \quad (\mathcal{D}\, f^{\bullet}\, x)\, \widehat{\star}\, f \qquad (12)$$

A **second batch** are the following metatheorems, whose counterparts appear in logical textbooks as the axiom and inference rule for quantification, but here are again consequences of (11), (3) and (4).

- Instantiation: $\forall\, P \Rightarrow x \in \mathcal{D}\, P \Rightarrow P\, x$
- Generalization: $q \Rightarrow x \in \mathcal{D}\, P \Rightarrow P\, x \vdash q \Rightarrow \forall\, P$

This is the basis for proving all the properties usually appearing in logic textbooks, as well as the following important rules for practical applications, viz., *trading*

$$\forall\, P_R \equiv \forall\,(R \mathbin{\widehat{\Rightarrow}} P) \quad \text{and} \quad \exists\, P_R \equiv \exists\,(R \mathbin{\widehat{\wedge}} P) \qquad (13)$$

A **third batch** is just meant to show the correspondence between the preceding point-free formulations and their conventional counterparts, and a few new ones. Let P, Q be *predicates*, let $R : X \to Y \to \mathbb{B}$ for some X and Y.

Empty rule	$\forall\, \varepsilon = 1$
1-point rule	$\forall\,(x \mapsto y) = y$
Merge rule	$P \,\textcircled{c}\, Q \Rightarrow \forall\,(P \mathbin{\dot\cup} Q) = \forall\, P \wedge \forall\, Q$
Distribution	$\mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q) = \forall\, P \wedge \forall\, Q$
Transposition	$\forall\,(\forall \circ R) = \forall\,(\forall \circ R^T)$
Composition	$\forall\, P \equiv \forall\,(P \circ f)$ provided $\mathcal{D}\, P \subseteq \mathcal{R}\, f$
Trading	$\forall\,(P \downarrow Q) \equiv \forall\,(Q \mathbin{\widehat{\Rightarrow}} P)$

We find the familiar ^{T}, $\circ$ and $\widehat{\ }$. Other generic functionals used are the *function merge* with $x \in \mathcal{D}\,(f \mathbin{\dot\cup} g) \equiv x \in \mathcal{D}\, f \cup \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x)$ and $x \in \mathcal{D}\,(f \mathbin{\dot\cup} g) \Rightarrow (f \mathbin{\dot\cup} g)\, x = (x \in \mathcal{D}\, f)\,?\, f\, x \nmid g\, x$. Also, the *function compatibility* relation $\textcircled{c}$ is defined by $f \,\textcircled{c}\, g \equiv \forall\,(f \mathbin{\widehat{=}} g)$.

Observe the algebraic (operator calculus-like) flavor of the various rules. Similar rules exist for $\exists$.

Replacing predicates by abstractions with $\mathbb{B}$-valued *expressions*, we can transform the various laws into their (perhaps) more familiar-looking traditional counterparts. We assume that p, q and r are boolean expressions, and that certain conditions on types and free occurrences are satisfied (left as exercises).

Empty rule	$\forall\,(x : \emptyset \,.\, p) = 1$
1-point rule	$\forall\,(x : X \,.\, x = y \Rightarrow p) \equiv y \in X \wedge p^x_y$
Domain split	$\forall\,(x : X \cup Y \,.\, p)$
(if compat.)	$\equiv \forall\,(x : X \,.\, p) \wedge \forall\,(x : Y \,.\, p)$
Distribution	$\forall\,(x : X \,.\, p \wedge q)$
	$\equiv \forall\,(x : X \,.\, p) \wedge \forall\,(x : X \,.\, q)$
Dummy swap	$\forall\,(x : X \,.\, \forall\, y : Y \,.\, p)$
	$\equiv \forall\,(y : Y \,.\, \forall\, x : X \,.\, p)$
Dummy chng	$\forall\,(x : X \,.\, p) \equiv \forall\,(y : Y \,.\, p^x_{f\,y})$
Trading	$\forall\,(x : X \mathbin{\dot\wedge} p \,.\, q) \equiv \forall\,(x : X \,.\, p \Rightarrow q)$

3.3 Example: refined function typing

Predicate calculus is perhaps the most generally applicable formalism throughout pure and applied mathematics, especially software engineering [11]. A collection of examples that is sufficiently complete to be representative is clearly far beyond the scope of this paper.

Therefore we restrict this discussion to showing how the predicate calculus wraps up a few issues that were mentioned in passing, but could not be properly handled at that time, especially about the function range.

Using $\{\ \}$ as a fully equivalent alternative for $\mathcal{R}$ gives expressions like $\{a, b, c\}$ and $\{n : \mathbb{Z} \,.\, 2 \cdot n\}$ their usual meaning. With $x : X \mid p$ as shorthand $x : X \mathbin{\dot\wedge} p \,.\, x$, this also yields $\square\, n = \{k : \mathbb{N} \mid k < n\}$. Note that we never overload $\{\ \}$ as a singleton set operator[2], since doing so would violate Leibniz's principle, which is the hallmark of any well-designed formalism. Here it requires that $x = a, b \Rightarrow \{x\} = \{a, b\}$, which holds in our formalism.

From the axiom

$$y \in \mathcal{R}\, f \quad \equiv \quad \exists\, x : \mathcal{D}\, f \,.\, y = f\, x \qquad (14)$$

we can derive $y \in \{x : X \mid p\} \equiv y \in X \wedge p^x_y$, which turns out to be the most often used form in practice.

We illustrate the various forms by the definition of f^- for *any* f (not only injective f). We take $\mathcal{D}\, f^-$ to contain just the points corresponding to unique elements in $\mathcal{D}\, f$. To formalize this, we introduce the *bijectivity domain* and the *bijectivity range*:

$$\begin{aligned} \mathsf{Bdom}\, f \quad &= \quad \{x : \mathcal{D}\, f \mid \forall\, x' : \mathcal{D}\, f \,.\, f\, x = f\, x' \Rightarrow x = x'\} \\ \mathsf{Bran}\, f \quad &= \quad \{x : \mathsf{Bdom}\, f \,.\, f\, x\}. \end{aligned} \qquad (15)$$

Finally we define the generic function inverse $—^-$ by

$$\mathcal{D}\, f^- = \mathsf{Bran}\, f \ \wedge\ \forall\, x : \mathsf{Bdom}\, f \,.\, f^-\,(f\, x) = x. \qquad (16)$$

This concludes this illustration.

Let us now refine the range as an element for function specification in general. Most common function types have the form $X \to Y$, with uniform range.

Finer typing is provided by an operator designed to formalize the concept of *tolerance* for functions. Engineering in the analog domain assumes certain tolerances on components. To extend this to functions, we introduce a *tolerance function* T that specifies, for every value x in its domain, the set $T\, x$ of allowable values. More precisely, a function f *meets* the tolerance T iff

$$\mathcal{D}\, f = \mathcal{D}\, T \quad \wedge \quad x \in \mathcal{D}\, f \cap \mathcal{D}\, T \Rightarrow f\, x \in T\, x.$$

The principle is illustrated pictorially in Fig. 4, using the example that provided the original motivation, namely a radio frequency filter characteristic.

[2]For singletons we use the *singleton set injector* ι (with axiom $x \in \iota\, y \equiv x = y$), which is preferable for other reasons [5] as well.

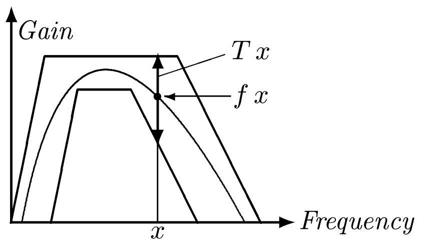

Figure 4: The function approximation paradigm

So we define an operator $\times$: for *any* family T of sets,

$$f \in \times T \equiv \mathcal{D}\,f = \mathcal{D}\,T \wedge \forall\,x\,{:}\,\mathcal{D}\,f \cap \mathcal{D}\,T\,.\,f\,x \in T\,x \quad (17)$$

Observe that, from the analogy with function *equality*:

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

it is easy to show $f = g \equiv f \in \times(\iota \circ g)$, i.e., our approximation operator also covers the exact case. We call $\times$ the *generalized functional Cartesian product*. It expresses the dependency of the result type on the domain value. If $\times T \neq \emptyset$, then $\times^{-}(\times T) = T$.

An instructive exercise is elaborating $\times(A, B)$ for sets A and B (since tuples are functions). This yields $\times(A, B) = A \times B$, the common Cartesian product defined by $(a, b) \in A \times B \equiv a \in A \wedge b \in B$. If $A \neq \emptyset$ and $B \neq \emptyset$, then $\times^{-}(A \times B)\,0 = A$ and $\times^{-}(A \times B)\,1 = B$.

Similarly, letting $T := a\,{:}\,A\,.\,B$ with a free in B, $\times(a\,{:}\,A\,.\,B) = \{f : A \to \bigcup a\,{:}\,A\,.\,B \mid \forall\,a\,{:}\,A\,.\,f\,a \in B\}$, known as a *dependent type* [8].

We write $A \ni a \to B_a$ as a suggestive shorthand for $\times\,a\,{:}\,A\,.\,B_a$, as in $A^{+} \ni x \to A^{\#\,x-1}$ which nicely characterizes the type of the aforementioned σ-operator. This shorthand is especially useful in chained dependencies, e.g., $A \ni a \to B_a \ni b \to C_{a,b}$.

4 Conclusion

It has been shown how mathematical concepts and operators arising from a seemingly specialized area of engineering (signal flow realizations) can be made generic and thereby extend their applicability to a much wider area of engineering and mathematics.

Here we have illustrated this by an algebraic and functional formulation of predicate calculus, providing a convenient formalism for specification and reasoning about software systems of an evolutionary nature.

References

[1] Richard Bird, *Introduction to Functional Programming using Haskell.* Prentice Hall, London (1998)

[2] Robert H. Bishop, *Learning with LabVIEW.* Addison Wesley Longman (1999)

[3] Raymond T. Boute, "Fundamentals of Hardware Description Languages and Declarative Languages", in: J. P. Mermet, ed., *Fundamentals and Standards in Hardware Description Languages*, pp. 3–38, Kluwer Academic Publishers (1993)

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

[5] T. E. Forster, *Set Theory with a Universal Set.* Clarendon Press, Oxford (1992)

[6] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove and D. S. Scott, *A Compendium of Discrete Lattices.* Springer-Verlag, Berlin (1980)

[7] David Gries, Fred B. Schneider, *A Logical Approach to Discrete Math.* Springer-Verlag, Berlin (1995)

[8] Keith Hanna and Neil Daeche and Gareth Howells, "Implementation of the Veritas design logic", in: Victoria Stavridou and Tom F. Melham and Raymond T. Boute, eds., *Theorem Provers in Circuit Design*, pp. 77–84. North Holland, Amsterdam (1992)

[9] Valerie Illingworth and Edward L. Glaser and I. C. Pyle, *Oxford Dictionary of Computing, 3rd.*. Oxford University Press (1991).

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

[11] David L. Parnas, "Predicate Logic for Software Engineering", *IEEE Trans. SWE 19*, 9, pp. 856–862 (Sept. 1993)

[12] Peter Rechenberg, "Programming Languages as Thought Models", *Structured Programming, 11*, pp. 105–111 (1990)

[13] John C. Reynolds,"Using Category Theory to Design Implicit Conversions and Generic Operators", in: Neil D. Jones, *Semantics-Directed Compiler Generation*, pp. 261-288, LNCS 94, Springer-Verlag, Berlin (1980)

[14] Richard A. Roberts and Clifford T. Mullis, *Digital Signal Processing.* Addison-Wesley Publishing Company (1987)

[15] Alfred Tarski and Steven Givant, *A Formalization of Set Theory Without Variables.* Colloquium Publications, Vol. 41. American Mathematical Society (1987)

[16] Wolfgang Wechler, *Universal Algebra for Computer Scientists.* Springer-Verlag, Berlin (1987)