CONCRETE GENERIC FUNCTIONALS: PRINCIPLES, DESIGN AND APPLICATIONS

Raymond T. Boute
Department of Information Technology
Ghent University
boute@intec.UGent.be

Abstract Generic design is not only pertinent to programs, but *a fortiori* to declarative formalisms as well, and the ensuing generalizations often benefit programming. In the considered framework, generality is obtained by (re)defining traditionally non-functional objects uniformly as functions. This results in *intrinsic polymorphism*, i.e., designing a collection of functionals without restriction on their arguments makes it intrinsically applicable to all objects captured by the uniformization.

Unlike the abstract operators on "arrows" in category theory, these generic functionals are *concrete*, as their definition explicitly takes into account relevant properties of their arguments, yet such that it benefits generality. This is done by associating types in a new, uniform fashion. The generic functionals thus designed include generalized composition and inverse, direct extension, transposition, override, merge etc., plus one more for expressing types, the functional cartesian product. Their genericity is fully general and extends beyond discrete mathematics.

As illustrated, their first purpose was providing smooth transformation between the point-wise style of expression and the point-free one (without variables) in the formal manipulation of analog and discrete signal flow circuit descriptions. In passing, this shows how a widely used graphical language can be seen as a functional programming language.

Once made generic, said functionals prove useful and convenient in diverse other areas including, as demonstrated, functional programming, aggregate data types, various kinds of polymorphism, predicate calculus, formal semantics, relational databases, relation algebra.

Keywords: Concrete generic functionals, functional mathematics, generic programming, composition, transposition, inverse, direct extension, override, merge, functional cartesian product, polymorphism, data flow, visual languages, abstract syntax, formal semantics, functional programming, relational databases, relation algebra

Introduction: motivation and overview

Generic programs are characterized in the conference announcement by: (a) increased adaptability through generality, (b) embodying non-traditional kinds of polymorphism, yielding ordinary programs by parametrization, (c) parameters that are richer in structure: other programs, types or type constructors, class hierarchies, even paradigms.

This characterization pertains *a fortiori* to declarative formalisms as used in (pure, applied) mathematics and engineering. Indeed, the term *generic* is usually defined as "of, applied to, referring to a whole kind, class or group" [14] or "having a wide or general application" [21]. This is exactly the purpose of the generic functionals we will present.

The three elements mentioned appear in our framework as follows, reading "functions" for "programs". (a) Generality is obtained by redefining traditionally non-functional objects uniformly as functions. (b) By designing the functionals such that they are unrestrictedly defined for all (function) arguments, irrespective of their types or other properties, they are applicable to all objects captured by the functional uniformization (*intrinsic polymorphism*). (c) The parameters of the functionals are other function(al)s, including type constructors. Expressions can be in the usual point-wise form (with variables) and in point-free form.

Clearly, principles developed in a declarative setting can benefit programming, as observed by Reynolds [24]: *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.*

The results reported here stem from concerns less directly related to programming, but more to the design of a formalism to unify the continuous and discrete systems models in applied mathematics and engineering (especially communications and automatic control) with each other and with the discrete concepts in computing science.

Whereas, in [24], Reynolds proposes category theory as the generalizing approach, we consider functions that are *concrete* in the sense that their mappings are not hidden. Instead of working with partial functions, which fit in a categorical framework [3, page 18], we fully specify the domains, for reasons explained in [8]. At first sight, one would expect this to entail extra work, but in practice the opposite turns out to be the case, precisely due to the generic design of the functionals.

One of the essential differences is illustrated by composition, the basic operator in category theory, where the source of $f \circ g$ is that of g. The

concrete counterpart we shall define is more flexible. Yet, since useful algebraic properties such as $f \circ (g \circ h) = (f \circ g) \circ h$ still hold, one could call this *concrete category theory*. Composition is only one example.

In a setting where most items are functions, this approach increases generality and expressivity. On the other hand, its concrete nature precludes other interpretations, such as interpreting arrows as the subset relation. In view of the proven usefulness in a wide variety of mathematics and engineering, this tradeoff is worthwhile.

The presentation is organized as follows. Section 1 explains the uniform treatment of objects as functions, and the reason why smooth transformation between point-wise and point-free expressions is necessary for certain practical applications. The calculational design of *signal flow networks* is taken as an example, since that is where the need for our collection of functionals first emerged. This includes a few operators familiar from classical applied mathematics, but many new ones to support transformations awkward or impossible to express in other formalisms. In Section 2, all these operators are generalized to *generic functionals* by associating *types* that ensure judicious bookkeeping of the domains of the argument and result functions. We also show how a specification problem in analog electronics yields a very convenient generic functional for expressing the type of generic functionals. Section 3 illustrates applications to various aspects of programming, including functional languages, aggregate data types, various kinds of polymorphism, predicate calculus, abstract syntax, formal semantics and relational databases.

1. Functionals for transformation

1.1 Unification by 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 justifies investing effort in designing them judiciously, especially w.r.t. the domains of the result functions to be specified in the types.

As an example, we briefly discuss the definition of sequences, which has wide ramifications since it constitutes either a common ground for or a serious gap between discrete and "continuous" mathematics, depending on whether or not sequences are defined as first-class functions. We use *sequences* as a generic term for tuples, streams, lists and so on, not necessarily finite and not necessarily homogeneous.

We define *all* these structures as functions such that $(a, b, c)\,0 = a$, $(a, b, c)\,1 = b$ etc. Whereas this is intuitively evident, browsing in the literature [12, 20, 25, 31] reveals that sequences are usually handled as entirely or subtly distinct from functions. Even the few exceptions consider only homogeneous structures and, more importantly, leave the functional characteristics 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 $(g, h)^T x = g\,x, h\,x$, which will be seen later to be widely useful once discovered and made generic.

Moreover, in most formal treatments, lists are not defined as functions but recursively via a prefixing operator (say, *cons*), viz.: [] is a list and, if x is a list, so is $cons\,a\,x$. Indexing requires an extra operator, defined recursively 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 also covers infinite sequences (with domain $\mathbb{N}$). Pairs like x, y in $f\,(x, y)$ are no exception, so every function 'of many arguments' is in fact a functional whose argument is a function that inherits all generic functionals.

Henceforth, sequences are functions with domain $\{k : \mathbb{N} \mid k < n\}$, written $\square\,n$, for $n : \mathbb{N}$ or $n := \infty$. Recall that homogeneity (as in lists) is *not* assumed. Operators for sequences, like # (length), ++ (concatenation), $\succ\!-$ (prefixing), σ (shift, tail) are defined as functionals, e.g., $(a \succ\!- x)\,n = (n = 0)\,?\,a \nmid x\,(n-1)$ 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 (where signals are sequences) with their analog counterparts (where signals are functions). We shall see how this extends to arbitrary aggregate data types (records, trees), and sequences become particular instances.

More generally, the functional view does *not* invalidate any traditional calculation rules, it only adds more powerful ones. Such conservational properties explain why *Funmath* [7], the formalism used in this paper, so much resembles the common notational conventions, except where the extra possibilities are exploited, and why it needs no prior introduction.

Indeed, Funmath has only four notational constructs: identifier, function application, tuple denotation, function abstraction, whose orthogonal combination synthesizes the vast majority of the common notations in *appearance* as well as *meaning* (minus the inconsistencies) in arguably the simplest way possible, as will gradually become apparent.

For the same reason, our generic functionals are defined without any extra notation. However, the resulting extra possibilities support the point-free style, which includes less common-looking expressions, as does

the Bird-Meertens formalism. Unlike the latter, the generic functionals are not restricted to discrete mathematics but allow any kind of function.

1.2 Practical need for point-free formulations

Function definitions and manipulations referring to points in the domains (by dummies) are called *point-wise*, otherwise they are *point-free*.

Traditional mathematical discourse is mostly point-wise. This may be largely due to the lack of suitable generic functionals to support the point-free style. Perhaps as a consequence, point-free formulations mostly appear in rather theoretical contexts [2, 28]. Yet, experience in many application areas indicates that the point-free style is usually more terse and more elegant (algebraic), although excesses are better avoided since the point-free style may appear baffling to the uninitiated.

For practical use, a formalism should support *both* styles, and smooth transformation rules between them. Here we demonstrate these necessities by an extended example. In passing, we illustrate how an interesting programming paradigm, namely *graphical programming*, can be seen as a non-textual, point-free form of functional programming.

Signal flow systems are interconnections of components whose dynamical behavior is modelled by functionals from input to output signals. We consider deriving signal flow systems realizing specified functions.

The simplest basic blocks are memoryless devices realizing arithmetic operations, and memory devices such as latches for the discrete and integrators for the continuous case. We model arithmetic blocks as "abstract operational amplifiers" for instantaneous values, and make this explicit by expressing, for instance, the sum of signals x and y as $x \mathbin{\widehat{+}} y$, with $(x \mathbin{\widehat{+}} y)\, t = x\, t + y\, t$. In control and communications engineering, one just overloads $+$ by $(x+y)\, t = x\, t + y\, t$, but we will see it pays off making $\widehat{\ }$ explicit as a *direct extension* functional, made generic later on.

Here it suffices considering the *synchronous* and *discrete* case, and the time variable is customarily n (of type $\mathbb{N}$). A *latch* is a one-cell device storing values between two subsequent clock cycles and parametrized by an initial condition. It is polymorphic w.r.t. the value stored. Its *behaviour* D can be specified formally for any initial condition a and input sequence x as $\mathrm{D}_a\, x\, n = (n = 0)\, ?\, a \mathbin{\dagger} x\, (n-1)$ or $\mathrm{D}_a\, x = a \mathbin{\succ\!\!-} x$.

In Fig. 1, (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 [5]. This language is originally designed for instrumentation purposes, but has much wider possibilities and a vast user community. We assume no

prior knowledge of LabVIEW; the concepts are clear from the examples.

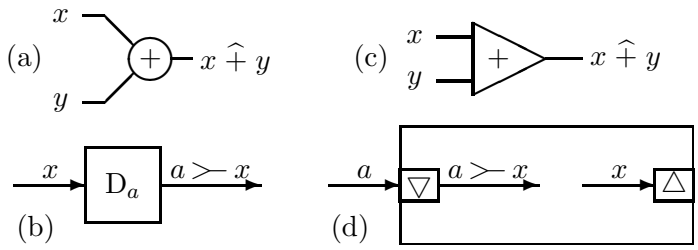

Figure 1. Typical basic building blocks

For *structurally* describing a signal flow system, no time variable should appear in our expressions, since time is not 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 [7].

1.3 A transformational design example

Consider the recursive specification

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

for a given set A, element a in A and function $g : A \to A$

This equation be transformed calculationally as follows

$$\begin{array}{rll} f\,n = & \langle \text{Def. } f \rangle & (n = 0)\,?\,a \nmid g\,(f\,(n-1)) \\ = & \langle \text{Def. } \circ \rangle & (n = 0)\,?\,a \nmid (g \circ f)\,(n-1) \\ = & \langle \text{Def. D} \rangle & \mathrm{D}_a\,(g \circ f)\,n \\ = & \langle \text{Def. } \overline{=} \rangle & \mathrm{D}_a\,(\overline{g}\ f)\,n \\ = & \langle \text{Def. } \circ \rangle & (\mathrm{D}_a \circ \overline{g}\,)\,f\,n, \end{array} \tag{2}$$

yielding the *fixpoint equation* $f = (\mathrm{D}_a \circ \overline{g}\,)\,f$ by extensionality. Note how function composition, i.e., $(g \circ f)\,x = g\,(f\,x)$ (considering types later), moves n into the argument position to enable subsequent elimination. We introduced $\overline{=}$, defined by $\overline{g}\ x = g \circ x$, for direct extension of one-argument functions. The form $\mathrm{D}_a \circ \overline{g}$ is interpretable structurally, since composition amounts to cascading, shown in Fig. 2(a) for $h \circ g$.

Here we should mention that, since tuples are functions, $f \circ (a, b) = f\,a, f\,b$, yielding a quite different structural interpretation, as shown in Fig. 2(b) assuming x is a pair. Note that all these interpretations correspond to the same abstract functional.

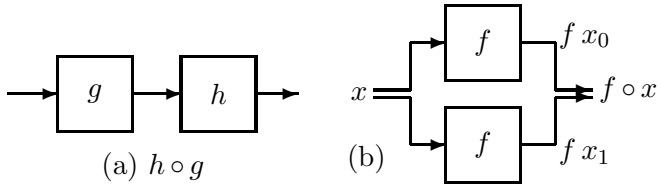

Figure 2. Structural interpretations of composition

Now the fixpoint equation $f = (\mathrm{D}_a \circ \overline{g})\, f$ has a direct realization, shown in Fig. 3(a) as a "textbook" block diagram and in Fig. 3(b) as a LabVIEW on-screen wiring diagram, with the conventions of Fig. 1.

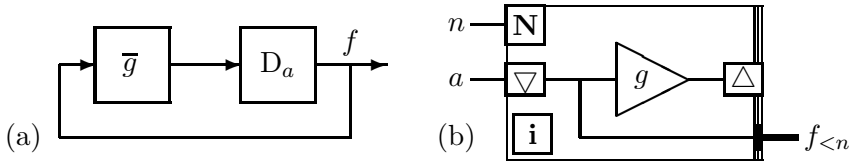

Figure 3. Signal flow realization of specification (1)

Aside: since f is an infinite sequence, the length of the subsequence $f_{<n}$ to be generated by a working system as in Fig. 3(b) is specified by an extra parameter $n : \mathbb{N}_{\geq 1}$. This is similar to using `take` in Haskell.

Of course, the transformation of equation (1) into the diagrams of Fig. 3 is so simple that it could have been done 'on sight', but the point is that the auxiliary operators make it possible to formalize even the smallest step of the process: without them, there is no 'handle', and one could only give a 'proof by inspection'.

Many situations require *swapping arguments* of a higher-order function. A typical case is the transformation of a family f of n signals into a single signal f^T with n-tuples as values such that $f^T t\, i = f\, i\, t$ at time t for all i in $\square\, n$, as illustrated in Fig. 4(a).

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 $f^T y\, x = f\, x\, y$, specifying the type as part of the generic design later on.

Again, tuples being functions, $(g, h)^T x = g\, x, h\, x$ yields the structural interpretation of Fig. 4(b), where f is a pair of functions, say, $f = g, h$. An important observation is that the variety of interpretations shown in Fig. 2 and 4 illustrates the generic nature of $\circ$ and ^{T}, which also depends on the unified definition of tuples and sequences as functions.

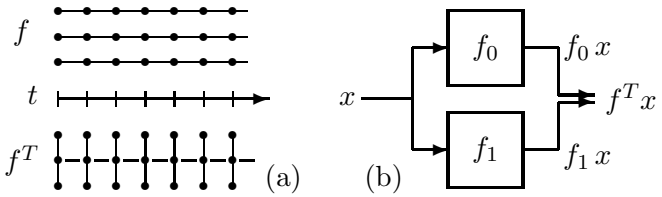

Figure 4. Behavioural and structural interpretations of transposition

Composition and transposition are each other's dual in the sense that $f \circ (x, y) = f\,x, f\,y$ and $(g, h)^T x = g\,x, h\,x$, but also in the (untyped) lambda calculus: provided x is not free in M, we find $M \circ (\lambda x.N) = \lambda x.MN$ and $(\lambda x.N)^T M = \lambda x.NM$.

Together, composition 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 $\star$,

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

(hints left as exercises) 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

$$\overset{<}{g}\ h = g \circ h^T \tag{3}$$

for any function g defined on functions and any family h of functions.

Much more is to be said about transformations, but we have collected enough representative functionals, and shall now make them generic.

2. Making the functionals generic

2.1 Conventions for functions; function equality

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 (the latter is just the *graph* of the function [20]). 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 an operator. Hence they are optional in $f(x)$ and in (a, b, c). Otherwise, the conventions and precedence rules are as in typical functional programming languages, including *partial application*: for any infix operator $\star$, we define $(a\star)\,b = a \star b = (\star b)\,a$. Here the details are of no interest, and always clear from the context.

Functions are *equal* iff their domains and mappings match. Formally, this amounts to Leibniz's principle and its converse. Including the *guard* $x \in \mathcal{D}\, f \cap \mathcal{D}\, g$ taking into account types [8], Leibniz's principle becomes

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

or $(q \Rightarrow f = g) \Rightarrow q \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x)$. The converse is function extensionality: using a fresh dummy x,

$$\frac{q \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x)}{q \Rightarrow f = g}. \tag{5}$$

The reason for inserting q is a proof-technical detail not discussed here.

As long as we have not yet elaborated the calculation rules for 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 (subscripts are comments specifying which of f and x may occur free).

An example is the *constant function specifier* $^\bullet$: for set X and any e,

$$\mathcal{D}\,(X^\bullet e) = X \quad \text{and} \quad x \in X \Rightarrow (X^\bullet e)\, x = e. \tag{6}$$

Why this 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 $\exists\, P \;\equiv\; P \neq \mathcal{D}\, P^\bullet 0$. These unprecedentedly simple definitions yield a powerful algebra (see later).

If q has the explicit form $f\, x = e_x$, the function can be denoted by an *abstraction* $x : X \mathbin{\mathbb{\wedge}} p \,.\, e$, where $\mathbin{\mathbb{\wedge}} p$ is optional. Expressed axiomatically,

$$\begin{aligned} d \in \mathcal{D}\,(x : X \mathbin{\mathbb{\wedge}} p \,.\, e) \quad &\equiv \quad d \in X \wedge p_d^x \\ d \in \mathcal{D}\,(x : X \mathbin{\mathbb{\wedge}} p \,.\, e) \quad &\Rightarrow \quad (x : X \mathbin{\mathbb{\wedge}} p \,.\, e)\, d = e_d^x \end{aligned} \tag{7}$$

(for any d), where p_d^x 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 (6) can be written $X^\bullet e = x : X \,.\, e$ (choosing x not free in e). Apart from compact function specifications for the "initiated", abstractions also reconstitute common notation for the uninitiated, for instance, since $P = x : \mathcal{D}\, P \,.\, P\, x$, clearly $\forall\, P = \forall\, x : \mathcal{D}\, P \,.\, P\, x$.

The *range* operator $\mathcal{R}$ is axiomatized by $y \in \mathcal{R}\, f \;\equiv\; \exists\, x : \mathcal{D}\, f \,.\, y = f\, x$. An equivalent operator symbol is $\{\text{—}\}$, so that application to tuples and abstractions yields expressions like $\{a, b, c\}$ and $\{n : \mathbb{Z} \,.\, 2{\cdot}n\}$ with familiar form and meaning. Using $x : X \mid p$ as syntactic sugar for $x : X \mathbin{\mathbb{\wedge}} p \,.\, x$ does the same for expressions like $\{m : \mathbb{N} \mid n < m\}$. Important is the

elimination of the ambiguities caused by the traditional double usage of $\in$ as set membership operator and as binding symbol; for instance, we can safely write $\{n : \mathbb{Z} \,.\, 2 \cdot n\} = \{n : \mathbb{Z} \mid n/2 \in \mathbb{Z}\}$. More is to be said about the ramifications, but this is not the topic of this paper. We just mention that singleton sets are denoted using Frege's *singleton set injector* ι [11] defined by $x \in \iota\, y \equiv x = y$.

A *type* for a function is its domain and partial information on images. For instance, $f \in A \to B \equiv \mathcal{D}\, f = A \wedge \mathcal{R}\, f \subseteq B$. For a family f of functions, the domains $\mathcal{D}\,(f\, x)$ of the images are part of the definition of f itself. For the functionals of interest, judicious design decisions regarding the image types is crucial to making the functionals fully generic.

2.2 Design criteria and method

The operators of interest here are functions over functions. In functional mathematics, they are shared by many more kinds of objects than usual, and hence deserve judicious design to make them generic by eliminating all restrictions on the argument functions.

In mathematics and computing, most functionals are *not* generic. 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 (ignoring the variant where inverse images are subsets of $\mathcal{D}\, f$), and then $\mathcal{D}\, f^- = \mathcal{R}\, f$.

As a design principle, we do not restrict the argument functions, but define the *domain of the result functions* to contain *exactly those points that do not cause out-of-domain applications* in the image definition.

This criterion supports the function specification discipline proposed in [8], namely, that all statements involving an application $f\, x$ are of the form $x \in \mathcal{D}\, f \Rightarrow P\,(f, x)$ or similar, which trivially holds if $x \notin \mathcal{D}\, f$. This simple convention handles out-of-domain applications without a separate calculus for "undefined". Out-of-domain applications may occur in regular mathematical discourse, e.g., for $\mathit{fac} : \mathbb{N} \to \mathbb{N}$, the image definition $\mathit{fac}\, 0 = 1 \wedge (n > 0 \Rightarrow \mathit{fac}\, n = \mathit{fac}\,(n-1) \cdot n)$ is reasonable, yet its instantiation for $n = 0$ contains the application $\mathit{fac}\,(-1)$. But since $n > 0 \equiv n - 1 \in \mathcal{D}\, \mathit{fac}$, the part of interest can be written $n - 1 \in \mathcal{D}\, \mathit{fac} \Rightarrow \mathit{fac}\, n = \mathit{fac}\,(n-1) \cdot n$. More details are given in [8].

A second(ary) design criterion is that, if our generic functional is a generalization of a commonly used functional, it is *conservational*, i.e., if the traditional restriction is satisfied, our generalization coincides with the "standard" case.

2.3 Functionals designed generically

All functionals pertain to continuous as well as discrete mathematics. Particularization to all data structures defined as functions is obvious.

The main transformation between the point-wise and point-free styles is function extensionality, captured by $f = x : \mathcal{D}\, f \,.\, f\, x$, obtained from (4), (5), (7). The first generic functional, *filtering*, provides a refinement. Most other generic functionals do not introduce or remove variables, but can move them to positions where we can apply extensionality.

Variables appear in our definitions, just as metavariables appear in combinator definitions [2]. Sometimes we use both the abstraction and the two-axiom format; the latter better highlights the design decisions.

Function and set filtering ($\downarrow$). For function f, predicate P,

$$\begin{array}{lcl} x \in \mathcal{D}\,(f \downarrow P) & \equiv & x \in \mathcal{D}\, f \cap \mathcal{D}\, P \wedge P\, x \\ x \in \mathcal{D}\,(f \downarrow P) & \Rightarrow & (f \downarrow P)\, x = f\, x. \end{array} \tag{8}$$

Readers familiar with subtyping in PVS [26] will notice the similarity. Moreover, for any set X we define $x \in (X \downarrow P) \equiv x \in X \cap \mathcal{D}\, P \wedge P\, x$.

We write a_b for $a \downarrow b$. With partial application, this yields a formal basis for shorthands like $f_{<n}$ and $\mathbb{Z}_{>0}$ that are *self-explanatory* without formal definitions, although only the latter support formal calculation.

Function restriction ($— \rceil —$). This is the usual domain restriction, a coarser variant of filtering: for any function f and set X,

$$f \rceil X \;=\; f \downarrow (X \bullet 1). \tag{9}$$

Function composition ($\circ$). For any functions f and g,

$$\begin{array}{lcl} x \in \mathcal{D}\,(f \circ g) & \equiv & x \in \mathcal{D}\, g \wedge g\, x \in \mathcal{D}\, f \\ x \in \mathcal{D}\,(f \circ g) & \Rightarrow & (f \circ g)\, x = f\,(g\, x). \end{array} \tag{10}$$

Note how the guard ensures that all applications are in-domain. If the traditional requirement $x \in \mathcal{D}\, g \Rightarrow g\, x \in \mathcal{D}\, f$ is satisfied, $\mathcal{D}\,(f \circ g) = \mathcal{D}\, g$.

Inversion ($—^{-}$). For any function f,

$$\mathcal{D}\, f^{-} = \mathsf{Bran}\, f \quad \text{and} \quad x \in \mathsf{Bdom}\, f \Rightarrow f^{-}\,(f\, x) = x. \tag{11}$$

For Bdom (*bijectivity domain*) and Bran (*bijectivity range*):

$$\mathsf{Bdom}\, f \;=\; \{x : \mathcal{D}\, f \mid \forall\, x' : \mathcal{D}\, f \,.\, f\, x' = f\, x \Rightarrow x' = x\} \tag{12}$$

$$\mathsf{Bran}\, f \;=\; \{x : \mathsf{Bdom}\, f \,.\, f\, x\}. \tag{13}$$

Note that, if the traditional injectivity condition is satisfied, $\mathcal{D}\, f^{-} = \mathcal{R}\, f$.

Dispatching (&) [22] and parallel (‖). For any functions f and g,

$$\mathcal{D}\,(f \,\&\, g) = \mathcal{D}\,f \cap \mathcal{D}\,g \qquad x \in \mathcal{D}\,(f \,\&\, g) \Rightarrow (f \,\&\, g)\,x = f\,x, g\,x \quad (14)$$
$$\mathcal{D}\,(f \parallel g) = \mathcal{D}\,f \times \mathcal{D}\,g \qquad x \in \mathcal{D}\,(f \parallel g) \Rightarrow (f \parallel g)\,(x,y) = f\,x, g\,y \quad (15)$$

(Duplex) direct extension ($\widehat{\overline{}}$). For any infix operator $\star$ and functions f and g,

$$\begin{array}{rcl} x \in \mathcal{D}\,(f \mathbin{\widehat{\star}} g) & \equiv & x \in \mathcal{D}\,f \cap \mathcal{D}\,g \wedge (f\,x, g\,x) \in \mathcal{D}\,(\star) \\ x \in \mathcal{D}\,(f \mathbin{\widehat{\star}} g) & \Rightarrow & (f \mathbin{\widehat{\star}} g)\,x = f\,x \star g\,x. \end{array} \quad (16)$$

Equivalently, $f \mathbin{\widehat{\star}} g = x : \mathcal{D}\,f \cap \mathcal{D}\,g \mathbin{\&} (f\,x, g\,x) \in \mathcal{D}\,(\star)\,.\,f\,x \star g\,x$. Often $x \in \mathcal{D}\,f \cap \mathcal{D}\,g \Rightarrow (f\,x, g\,x) \in \mathcal{D}\,(\star)$; then $f \mathbin{\widehat{\star}} g = x : \mathcal{D}\,f \cap \mathcal{D}\,g\,.\,f\,x \star g\,x$.

A noteworthy example is *equality*: $(f \mathbin{\widehat{=}} g) = x : \mathcal{D}\,f \cap \mathcal{D}\,g\,.\,f\,x = g\,x$, making $f \mathbin{\widehat{=}} g$ a predicate on $\mathcal{D}\,f \cap \mathcal{D}\,g$.

Sometimes we need only **half direct extension**: for any function f and any x, we define $f \mathbin{\overleftarrow{\star}} x = f \mathbin{\widehat{\star}} \mathcal{D}\,f^{\bullet}\,x$ and $x \mathbin{\overrightarrow{\star}} f = \mathcal{D}\,f^{\bullet}\,x \mathbin{\widehat{\star}} f$.

Recall that **simplex direct extension** ($\overline{}$) is defined by $\overline{f}\,g = f \circ g$.

Function override. For any functions f and g, $g \oslash f = f \oslash g$ and

$$\begin{array}{rcl} \mathcal{D}\,(f \oslash g) & = & \mathcal{D}\,f \cup \mathcal{D}\,g \\ x \in \mathcal{D}\,(f \oslash g) & \Rightarrow & (f \oslash g)\,x = x \in \mathcal{D}\,f \,?\, f\,x \mathbin{\dagger} g\,x \end{array} \quad (17)$$

Function merge (∪). For any functions f and g,

$$\begin{array}{rcl} x \in \mathcal{D}\,(f \cup g) & \equiv & x \in \mathcal{D}\,f \cup \mathcal{D}\,g \wedge (x \in \mathcal{D}\,f \cap \mathcal{D}\,g \Rightarrow f\,x = g\,x) \\ x \in \mathcal{D}\,(f \cup g) & \Rightarrow & (f \cup g)\,x = x \in \mathcal{D}\,f \,?\, f\,x \mathbin{\dagger} g\,x. \end{array} \quad (18)$$

Relational functionals: compatibility (©), subfunction (⊆).

$$\begin{array}{rcl} f \mathbin{©} g & \equiv & f \rceil \mathcal{D}\,g = g \rceil \mathcal{D}\,f \\ f \subseteq g & \equiv & f = g \rceil \mathcal{D}\,f. \end{array}$$

Typical algebraic properties are $f \subseteq g \equiv \mathcal{D}\,f \subseteq \mathcal{D}\,g \wedge f \mathbin{©} g$, the fact that $\subseteq$ is a partial order (reflexive, antisymmetric, transitive), and $f \mathbin{©} g \Rightarrow f \oslash g = f \cup g = f \otimes g$. Properties about equality are $f \mathbin{©} g \equiv \forall\,(f \mathbin{\widehat{=}} g)$ and $f = g \equiv \mathcal{D}\,f = \mathcal{D}\,g \wedge f \mathbin{©} g$.

Remark on algebraic properties. The operators presented entail a rich collection of algebraic laws that can be expressed in point-free form, yet preserve the intricate domain refinements (as can be verified calculationally). Examples are: for composition, $f \circ (g \circ h) = (f \circ g) \circ h$ and $\overline{h \circ g} = \overline{h} \circ \overline{g}$; for extension, $(\widehat{\star}) = \overline{(\star)} \circ (\&)$. Elaboration is beyond the scope of this paper, giving priority to later application examples.

2.4 Elastic extensions for generic operators

Introduction. *Elastic operators* are functionals that, combined with function abstraction (as the only kind), unobtrusively replace the many ad hoc abstractors from daily mathematics, such as $\forall\, x : X$ and $\sum_{i=m}^{n}$ and $\lim_{x\to a}$. Recall how our quantifier definition $\forall\, P \;\equiv\; P = \mathcal{D}\, P \bullet 1$, although point-free, allows writing familiar formulas like $\forall\, x : \mathbb{R} \,.\, x^2 \geq 0$, and also has properties like $\forall\,(x, y) = x \wedge y$ for boolean x and y. Another example is the function range operator. Summation is defined by

$$\sum \varepsilon = 0 \qquad \sum (a \mapsto c) = c \qquad \sum (f \cup g) = \sum f + \sum g \tag{19}$$

for any a, any numeric c and any number-valued functions with finite nonintersecting (but otherwise arbitrary) domains. In general, $\varepsilon = \emptyset \bullet x$ (any x) defines the *empty function* and $x \mapsto y = \iota\, x \bullet y$ (as in Z [27]) defines *one-point* functions. Observe how (19) shows $\sum$ to be a *merge homomorphism*, generalizing the well-known homomorphisms of the form $h\,(x +\!\!+\, y) = h\, x \oplus h\, y$ for data structures x and y (e.g., lists).

Elastic operators subsume the so-called Eindhoven Quantifier Notation $Q\, x : P.x : f.x$, but the underlying principle and the design are quite different and more general in many respects, e.g., support for the point-free style, no algebraic restrictions (associativity etc.), not restricted to discrete mathematics (many applications in mathematical analysis).

If an elastic operator F and (infix) operator $\star$ satisfy $F\,(x, y) = x \star y$, then F is an *elastic extension* of $\star$, e.g., $\forall\,(x, y) \;\equiv\; x \wedge y$ and $\sum\,(x, y) = x + y$. Such extensions are not unique, leaving room for judicious design.

Argument/operator alternations of the form $x \star y \star z$ are called *variadic shorthand* and (in our formalism) are *always* defined via an elastic extension: $x \star y \star z = F\,(x, y, z)$, for instance, $x + y + z = \sum\,(x, y, z)$. This is not restricted to associative or commutative operators. For instance, letting con and inj be the *constant* and *injective* predicates over functions, we define $x = y = z \;\equiv\; \mathsf{con}\,(x, y, z)$ and $x \neq y \neq z \;\equiv\; \mathsf{inj}\,(x, y, z)$. Especially the latter is interesting: it gives $x \neq y \neq z$ the most useful meaning (x, y, z distinct), which traditional conventions cannot do.

Variadic shorthand for an operator $\star$ depends on the judicious design of its elastic extension F. For instance, if $\star$ is associative, it is wise to impose this property on the variadic shorthand by requiring that the restriction of F to lists is a *list homomorphism*, viz., $F\,(x +\!\!+\, y) = F\, x \star F\, y$. This ensures that $a \star b \star c = (a \star b) \star c$ etc. for any number of arguments. To appreciate this, the reader may consider designing the elastic extension of $\equiv$ (logical equivalence) reflecting associativity.

Full elaboration of these issues is far beyond the scope of this paper.

Generic functionals with "two" function arguments also have elastic extensions. We illustrate this only for the most interesting designs.

Transposition ($—^T$). Observing that $(g \,\&\, h)\, x\, i = (g, h)\, i\, x$ for i in $\{0, 1\}$ suggests taking the earlier *transposition* operator as the basis for the elastic extension, in view of the argument swap in $f^T y\, x = f\, x\, y$. The main step in making $—^T$ generic is deciding on the definition of $\mathcal{D}\, f^T$ for *any* function family f. We consider two of the many options.

Recall from (3) that defining $\overset{<}{g}\ f = g \circ f^T$ to generalize (16) depends on intersection. This is captured by taking $\mathcal{D}\, f^T = \bigcap x : \mathcal{D}\, f \,.\, \mathcal{D}\,(f\, x)$ or $\mathcal{D}\, f^T = \bigcap (\mathcal{D} \circ f)$. Hence $f^T = y : \bigcap (\mathcal{D} \circ f) \,.\, x : \mathcal{D}\, f \,.\, f\, x\, y$ or

$$\begin{array}{rcl} \mathcal{D}\, f^T & = & \bigcap x : \mathcal{D}\, f \,.\, \mathcal{D}\,(f\, x) \\ y \in \mathcal{D}\, f^T & \Rightarrow & \mathcal{D}\,(f^T y) = \mathcal{D}\, f \\ & \wedge & (x \in \mathcal{D}\,(f^T y) \Rightarrow f^T y\, x = f\, x\, y) \end{array} \tag{20}$$

We found this variant to be the most frequently useful in practice.

In a more liberal design, called *uniting transposition* ($—^U$), we take $\mathcal{D}\, f^U = \bigcup (\mathcal{D} \circ f)$. Elaboration of this option subject to the design criterion yields $f^U = y : \bigcup (\mathcal{D} \circ f) \,.\, x : \mathcal{D}\, f \wedge y \in \mathcal{D}\,(f\, x) \,.\, f\, x\, y$ or

$$\begin{array}{rcl} \mathcal{D}\, f^U & = & \bigcup x : \mathcal{D}\, f \,.\, \mathcal{D}\,(f\, x) \\ y \in \mathcal{D}\, f^U & \Rightarrow & \mathcal{D}\,(f^U y) = \{x : \mathcal{D}\, f \mid y \in \mathcal{D}\,(f\, x)\} \\ & \wedge & (x \in \mathcal{D}\,(f^U y) \Rightarrow f^U y\, x = f\, x\, y) \end{array} \tag{21}$$

Parallel ($\|$). For any function family F and function f,

$$\| F\, f \quad = \quad x : \mathcal{D}\, F \cap \mathcal{D}\, f \wedge f\, x \in \mathcal{D}\,(F\, x) \,.\, F\, x\,(f\, x) \tag{22}$$

This can be seen as a typed variant of the well-known $\boldsymbol{S}$-combinator [2].

Elastic merge. For any function family f,

$$\begin{array}{rcl} y \in \mathcal{D}\,(\bigcup f) & \equiv & \\ y \in \bigcup (\mathcal{D} \circ f) \wedge \forall\, (x, x') : (\mathcal{D}\, f)^2 \,.\, y \in \mathcal{D}\,(f\, x) \cap \mathcal{D}\,(f\, x') & \Rightarrow & f\, x\, y = f\, x'\, y \end{array} \tag{23}$$

$$y \in \mathcal{D}\,(\bigcup f) \quad \Rightarrow \quad \forall\, x : \mathcal{D}\, f \,.\, y \in \mathcal{D}\,(f\, x) \Rightarrow \bigcup f\, y = f\, x\, y \tag{24}$$

$\mathcal{D}\, f$ need not be discrete, e.g., $g = \bigcup x : \mathcal{D}\, g \,.\, x \mapsto g\, x$ for any function g. A very interesting theorem is $g^- = \bigcup x : \mathcal{D}\, g \,.\, g\, x \mapsto x$ for any function g (irrespective of injectivity), which demonstrates how the generic design criterion leads to fine intermeshing of the operators.

Elastic compatibility. For any function family f

$$\copyright\, f \quad \equiv \quad \forall\, (x, y) : (\mathcal{D}\, f)^2 \,.\, f\, x \copyright f\, y \tag{25}$$

In general, $\cup$ is not associative, but $\copyright\, (f, g, h) \Rightarrow (f \cup g) \cup h = f \cup (g \cup h)$.

2.5 A generic functional refining function types

The most common function typing operator is the *function arrow* ($\rightarrow$) defined by $f \in X \rightarrow Y \;\equiv\; \mathcal{D}\, f = X \wedge \mathcal{R}\, f \subseteq Y$. It specifies the type of $f\, x$ uniformly as Y.

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 in Fig. 5, using the example that provided the original motivation, namely a radio frequency filter characteristic.

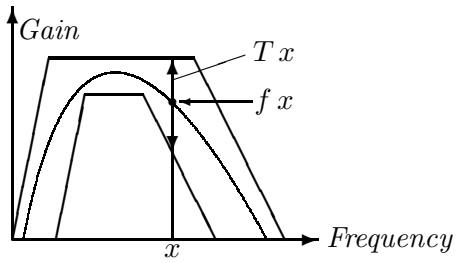

Figure 5. The function approximation paradigm

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

$$f \in \bigtimes T \;\equiv\; \mathcal{D}\, f = \mathcal{D}\, T \wedge \forall\, x : \mathcal{D}\, f \cap \mathcal{D}\, T \,.\, f\, x \in T\, x \tag{26}$$

Equivalently, $\bigtimes T = \{f : \mathcal{D}\, T \rightarrow \bigcup T \mid \forall\, (f\, \widehat{\in}\, T)\}$.

Observe that, from the analogy between (26) and function *equality*:

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

$f = g \equiv f \in \bigtimes (\iota \circ g)$, i.e., our approximation operator also covers the exact case. We call $\bigtimes$ the *generalized functional Cartesian product*.

It is instructive to elaborate $\bigtimes (A, B)$ for sets A and B (tuples being functions). This yields $\bigtimes (A, B) = A \times B$, the common Cartesian product defined (for tuples as functions) by $(a, b) \in A \times B \;\equiv\; a \in A \wedge b \in B$.

Clearly, $\bigtimes (a : A \,.\, B_a) \;=\; \{f : A \rightarrow \bigcup\, a : A \,.\, B_a \;\mid\; \forall\, a : A \,.\, f\, a \;\in\; B_a\}$. This point-wise form is a *dependent type* [17] or *product of sets* [29].

We write $A \ni a \rightarrow B_a$ as a suggestive shorthand for $\bigtimes a : A \,.\, B_a$, which is especially useful in chained dependencies, e.g., $A \ni a \rightarrow B_a \ni b \rightarrow C_{a,b} =$

$\times a : A \,.\, \times b : B_a \,.\, C_{a,b}$. The 3-position macro $— \ni — \to —$ should not be confused with the function arrow $(\to)$, but merges conveniently with it, as in $A \ni a \to B_a \to C_a = \times a : A \,.\, B_a \to C_a$.

An illustration are types for generic operators. Letting $\mathcal{F}$ be the function universe, $\mathcal{T}$ the type universe (not critically dependent on the chosen axiomatization [11]), and fam an operator such that, for any set Y and function $f \in \mathsf{fam}\ Y \equiv \mathcal{R}\, f \subseteq Y$,

$$\begin{array}{lcl} (\circ) & \in & \mathcal{F}^2 \ni (f, g) \to \{x : \mathcal{D}\, g \mid g\, x \in \mathcal{D}\, f\} \to \mathcal{R}\, f \\ ({}^T) & \in & \mathsf{fam}\ \mathcal{F} \ni f \to \bigcap(\mathcal{D} \circ f) \to \mathcal{D}\, f \ni x \to \mathcal{R}\,(f\, x). \end{array}$$

3. Applications in programming

Section 1 contains an extended application example for signal flow networks. The same functionals apply to analog circuit description at the signal (block diagram) level consisting of summers, multipliers (modulators), integrators, filters and so on. We have also shown how filters can be specified within a given tolerance, and how the same functional can express types. From here on, we only consider programming.

3.1 Functional programming

All generic operators introduced are directly applicable in functional programming for function arguments. This can be extended in a useful and interesting way to sequences (tuples, lists, etc.) and other structures, provided these are also defined as functions. Although this condition is generally not satisfied (lists and other structures usually being defined recursively), it is a worthwhile language design alternative, so let us consider the potential benefits. First of all, nothing would be lost.

Indeed, defining sequences as functions and operators like $\succ\!\!-$ by

$$a \succ\!\!- x \quad = \quad i : \square\,(\#\, x + 1) \,.\, (i = 0)\,?\, a \nmid x\,(i - 1) \tag{27}$$

does not preclude inductive reasoning in the familiar style. The induction principle whereby, for given set A and predicate $P : A^* \to \mathbb{B}$,

$$\forall\, P \quad \equiv \quad P\, \varepsilon \wedge \forall\, x : A^* \,.\, P\, x \Rightarrow \forall\, a : A \,.\, P\,(a \succ\!\!- x).$$

holds as a theorem, and existing inductive proofs based on recursive definitions require no change. All function properties derived from (27) come *in addition to* rather than *instead of* the induction principle. The same observations, of course, hold for infinite structures and coinduction.

In this functional context, generic operators like composition and transposition have interesting applications for sequences and, as we shall see, other structures as well.

For sequences, $(0,3,5,7) \circ (2,3,1) = 5,7,3$ and $(0,3,5,7) \circ (2,3,5) = 5,7$. Note also that $(0,3,5,7) \circ (5,3,1) = (7,3) \circ (-1)$.

Since $f \circ (x,y) = f\,x, f\,y$ (for x and y in $\mathcal{D}\,f$), our $\circ$ subsumes the `map` operator [4], viz., `map f [x, y] = [f x, f y]` (for lists *and* tuples).

Since $(f,g)^T x = f\,x, g\,x$, transposition subsumes Meyer's &-operator. However, by its generic nature it also subsumes the `zip` operator [4], e.g., `zip [a,b,c] [a',b',c'] = [[a,a'],[b,b'],[c,c']]` becomes, up to currying, $((a,b,c),(a',b',c'))^T = ((a,a'),(b,b'),(c,c'))$. Most functional languages support direct definition of the transposition operator for the (very) special case of function families with type of the form $A \to (B \to C)$. The image f^T of $f : A \to (B \to C)$ has type $B \to (A \to C)$ and property $(f^T)^T = f$.

Sequences have inverses: $(3,3,7)^-\ 7 = 2$, but so do certain operators over sequences (and other structures). A ramification is the following.

The familiar *pattern matching*, as in *head* $(a \succ\!\!- x) = a$ or in recursive definitions like $f\,(a \succ\!\!- x) = h\,(a, f\,x)$, can be seen as a particularization of equational definition using function inverses. Indeed, given the definition format $f\,(g\,(a,x)) = e\,(a,x)$, then application of f to an actual parameter y satisfies $f\,y = e\,(g^-\,y\,0, g^-\,y\,1)$ for any y in the bijectivity range of g. In our example, $\succ\!\!-^-\,(a \succ\!\!- x)\,0 = a$ and $\succ\!\!-^-\,(a \succ\!\!- x)\,1 = x$.

For direct extension, application opportunities abound. For sequences, direct extension can be seen as pairwise map in the sense that `direx (*) [[a,b,c],[a',b',c']] = [a * a', b * b', c * c']` or (up to currying), `direx (*) = (map (*)) . zip`), which is a particularization of $(\widehat{\star}) = \overline{(\star)} \circ (\&)$. More on this topic is said in the next section.

3.2 Aggregate data types and structures

Apart from sequences, the most ubiquitous aggregate data type are *records* in the sense of PASCAL [19]. One approach for expressing records functionally is using *selector functions* corresponding to the field labels, where the records themselves appear as arguments. We have explored this alternative some time ago in a different context [6], and it is also currently used in Haskell [18]. However, it does not make records themselves into functions and has a rather heterogeneous flavor.

Therefore our preferred alternative is the generalized functional cartesian product (abbreviated *funcart*) operator from (26), whereby records are defined as *functions* whose domain is a set of field labels constituting an *enumeration type*. For instance,

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

where *name* and *age* are elements of an enumeration type, defines a function type such that every constant or variable $person : Person$ sat-

isfies *person name* $\in \mathbb{A}^*$ and *person age* $\in \mathbb{N}$. The syntax can be made more attractive by defining, for instance, a type definition functional $\mathsf{Record} : \mathsf{fam}\,(\mathsf{fam}\,\mathcal{T}) \to \mathcal{P}\,\mathcal{F}$ with $\mathsf{Record}\,F = \bigtimes(\bigcup F)$, so we can write $Person := \mathsf{Record}\,(name \mapsto \mathbb{A}^*, age \mapsto \mathbb{N})$.

In fact, the funcart operator is the "workhorse" for typing all structures unified by functional mathematics. Obviously, $A \to B = \bigtimes(A^\bullet B)$ and $A \times B = \bigtimes(A, B)$. For any set A and n in $\mathbb{N} \cup \iota\infty$, we define $A \uparrow n$ (abbreviated A^n) by $A \uparrow n = \square\, n \to A$, so A^n is the n-fold Cartesian product. We also define $A^* = \bigcup n : \mathbb{N} \,.\, A^n$. This completes the functional unification of aggregates (sequences, records etc.) at the type level.

Having $\bigtimes$ as a genuine functional rather than an ad hoc abstractor yields many useful algebraic properties. Most noteworthy is the inverse. By the axiom of choice, $\bigtimes T \neq \emptyset \equiv \forall x : \mathcal{D}\,T \,.\, T\,x \neq \emptyset$. This also characterizes the bijectivity domain of $\bigtimes$ and, if $\bigtimes T \neq \emptyset$, then $\bigtimes^-(\bigtimes T) = T$. For the usual cartesian product this implies that, if $A \neq \emptyset$ and $B \neq \emptyset$, then $\bigtimes^-(A \times B) = A, B$, hence $\bigtimes^-(A \times B)\,0 = A$ and $\bigtimes^-(A \times B)\,1 = B$. Finally, an explicit image definition is

$$\bigtimes^- S = x : Dom\,S \,.\, \{f\,x \mid f : S\} \tag{28}$$

for any nonempty S in the range of $\bigtimes$, where $Dom\,S$ is the common domain of the functions in S (extracted, e.g., by $Dom\,S = \bigcap f : S \,.\, \mathcal{D}\,f$).

As mentioned, other structures are also defined as functions. For instance, *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. This covers any kind of branch labeling. For instance, for a binary tree, the branching structure is a subset of $\mathbb{B}^*$. Classes of trees are characterized by restrictions on the branching structures. The $\bigtimes$ operator can even specify types for leaves individually.

Aggregates defined as functions inherit all elastic operators for which the images are of suitable type. For instance, $\sum s$ sums the fields or leaves of any number-valued record, tree or other structure s.

Of course, generic functionals are inherited whatever the image type.

Especially important for structures is direct extension. In fact, Dijkstra [10] considers all operators (even equality) implicitly extended to "structures" in a similar fashion, yet without elaborating the domain since this is always the program state space. LabVIEW [5] building blocks are similarly extended, which is referred to as *polymorphism*. Such implicit extensions are convenient in a particular area of discourse, but for a broader application range the finer tuning offered by an explicit generic operator is preferable.

3.3 Overloading and polymorphism

Overloading a symbol (identifier) means using it for designating "different" objects. Of course, for simple objects, this is possible only in different contexts, or in very informal contexts where the intended designation can be inferred, since otherwise ambiguity would result.

For operators, i.e., function symbols, properties of the designated objects can be used for disambiguation, even in formal contexts. If the functions designated by the overloaded operator have different types but formally the same image definition, this form of overloading is called *polymorphism*. Hence considering general overloading (or *ad hoc* polymorphism in Haskell) also suffices for covering polymorphism.

Overloading involves two main issues: *disambiguation*, making the application of the overloaded operator to all its possible arguments unambiguous, and *refined typing*, reflecting the type information of the designated functions in the operator's type. The first requires that the different functions represented by the operator be *compatible*. The second requires a suitable type operator whose design is discussed next.

Overloading by explicit parametrization. Using a single operator to represent various functions can be done in a trivial way by an auxiliary parameter that directly or indirectly indicates the intended function. The operator for expressing its type is already available, namely $\times$. An example is the *binary addition* function adding two binary words of equal length to obtain a result that is one (overflow) bit longer.

$$\textbf{def}\ \mathit{binadd}\_ : \times n : \mathbb{N} \,.\, (\mathbb{B}^n)^2 \to \mathbb{B}^{n+1}\ \textbf{with}\ \mathit{binadd}_n\,(x, y) = \ldots$$

Only the type is relevant. Note: $\mathit{binadd}_n \in (\mathbb{B}^n)^2 \to \mathbb{B}^{n+1}$ for any $n : \mathbb{N}$.

Overloading without auxiliary parameter. We want a type operator $\otimes$ expressing overloading without auxiliary parameter in the manner exemplified for *binadd* by

$$\textbf{def}\ \mathit{binadd} : \otimes n : \mathbb{N} \,.\, (\mathbb{B}^n)^2 \to \mathbb{B}^{n+1}\ \textbf{with}\ \mathit{binadd}\,(x, y) = \ldots$$

Note that $n : \mathbb{N} \,.\, (\mathbb{B}^n)^2 \to \mathbb{B}^{n+1}$ is a family of function types. The domain of *binadd* is $\bigcup\, n : \mathbb{N} \,.\, (\mathbb{B}^n)^2$, and the type $\bigcup\,(n : \mathbb{N} \,.\, (\mathbb{B}^n)^2) \ni (x, y) \to \mathbb{B}^{\#\,x+1}$, but this must be obtainable from $\otimes\, n : \mathbb{N} \,.\, (\mathbb{B}^n)^2 \to \mathbb{B}^{n+1}$.

Among many variants, an interesting design is obtained as follows. Clearly *binadd* is a *merge* of functions of type $(\mathbb{B}^n)^2 \to \mathbb{B}^{n+1}$, one for each n. The family of functions merged is taken from $\times\, n : \mathbb{N} \,.\, (\mathbb{B}^n)^2 \to \mathbb{B}^{n+1}$, subject to the *compatibility* requirement (trivially satisfied in this example because of nonintersecting domains). This reasoning is directly

generalizable from the function type family in the example to any function type family, yielding the *function type merge* ($\bigotimes$) operator [30].

$$\textbf{def} \bigotimes : \mathsf{fam}\,(\mathcal{P}\,\mathcal{F}) \to \mathcal{P}\,\mathcal{F}\ \textbf{with}\ \bigotimes F = \{\bigcup\, f \mid f : (\times F)_{\copyright}\}. \qquad (29)$$

This is not the original form, since the functionals available at the time of the design of $\bigotimes$ were not generic; to the contrary: non-generic effects were exploited to enforce compatibility. The current form (29) nicely illustrates the use of generic functionals.

In a (declarative) language context, incremental definition can be supported as follows. Regular definitions have the form **def** $x : X$ **with** p_x, allowing only one definition for x. For defining overloaded operators incrementally, **defo** indicates that this condition is dropped, but replaced by the stipulation that a collection of definitions of the form **defo** $f : F_i$ **with** $P_i\, f$ (for $i : I$) requires that the derived collection of definitions **def** $g_i : F_i$ **with** $P_i\, g_i$ is *compatible*, i.e., $\copyright\, i : I \,.\, g_i$ (often satisfied trivially by nonintersecting domains). Then the given collection defines an operator f of type $\bigotimes i : I \,.\, F_i$ with $f = \bigcup i : I \,.\, g_i$.

A rough analogy with Haskell is illustrated by an example in [18]:

```
class Eq a where (==) ::  a -> a -> Bool
instance Eq Integer where x == y = x `integerEq` y
instance Eq Float where x == y = x `floatEq` y
```

This is approximately (and up to currying; infix operators are not implicitly curried) rendered by

$$\begin{array}{l}\textbf{def}\ \mathit{Eq} : \mathcal{T} \to \mathcal{P}\,\mathcal{F}\ \textbf{with}\ \mathit{Eq}\, X = X^2 \to \mathbb{B}\\ \textbf{defo}\ \text{—==—} : \mathit{Eq}\ \mathit{Integer}\ \textbf{with}\ x == y \equiv x\ \mathit{integerEq}\ y\\ \textbf{defo}\ \text{—==—} : \mathit{Eq}\ \mathit{Float}\ \textbf{with}\ x == y \equiv x\ \mathit{floatEq}\ y.\end{array}$$

The type of == is $\mathit{Eq}\ \mathit{Integer} \otimes \mathit{Eq}\ \mathit{Float}$ and, observing that nonintersecting domains ensure compatibility, $(==) = (\mathit{integerEq}) \cup (\mathit{floatEq})$. An essential difference is that Haskell attaches operators to a class.

3.4 Functional predicate calculus

Predicate calculus is a basic mathematical tool in software engineering [23]. A faithful elaboration would take an extensive paper by itself. Indeed, as demonstrated in [13], any predicate calculus for practical use requires a fairly complete "toolkit" of calculation rules, not just the basic axioms found in typical logic textbooks. For our approach, such a toolkit is presented in [9]. Here we shall only provide a few examples demonstrating the algebraic (point-free) formulation style enabled by the use of generic functionals, and how traditional-looking forms are obtained

by writing predicates as function abstractions, e.g., $P = x : X \,.\, p_x$ (with the evident (meta-)convention that the subscript indicates possible free occurrences of x, so $x : X \,.\, p$ is constant).

Recall that a *predicate* is a boolean-valued function. Here, the choice ***false***/***true*** versus 0/1 is secondary. In a wider context (not discussed here), 0/1 is advantageous.

Axioms. 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:

$$\begin{array}{lll} \forall\, P \;\equiv\; (P = \mathcal{D}\, P^\bullet 1) & \forall\, (x : X \,.\, p_x) \;\equiv\; (x : X \,.\, p_x) = (x : X \,.\, 1) \\ \exists\, P \;\equiv\; (P \neq \mathcal{D}\, P^\bullet 0) & \exists\, (x : X \,.\, p_x) \;\equiv\; (x : X \,.\, p_x) \neq (x : X \,.\, 0) \end{array} \quad (30)$$

These definitions are conceptually indeed as simple as they look, but they give rise to a rich algebra of calculation rules, all derived using function equality (4) and (5).

Direct consequences. Immediate examples are shown in the table below. The first one allows deriving later rules for $\exists$ from those for $\forall$.

Duality	$\forall\, (\overline{\ni}\, P) \;\equiv\; (\overline{\ni}\, \exists)\, P$	$\forall\, (x : X \,.\, \neg\, p_x) \;\equiv\; \neg\, (\exists\, x : X \,.\, p_x)$
Meeting	$\forall\, P \wedge \forall\, Q \Rightarrow \forall\, (P \mathbin{\widehat{\wedge}} Q)$	$\forall\, (x : X \,.\, p_x) \wedge \forall\, (x : Y \,.\, q_x) \Rightarrow \forall\, (x : X \cap Y \,.\, p_x \wedge q_x)$
Constant	$\forall\, (X^\bullet\, p) \;\equiv\; X = \emptyset \vee p$	$\forall\, (x : X \,.\, p) \;\equiv\; X = \emptyset \vee p$

The converse of *meeting* is $\mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow \forall\, (P \mathbin{\widehat{\wedge}} Q) \Rightarrow \forall\, P \wedge \forall\, Q$.

The rules in most logic textbooks are untyped and pay little attention to (possibly empty) domains. For the finite structures in software and algorithmics, such attention turns out to be especially important. Obvious particular cases are $\forall\, \varepsilon \;\equiv\; 1$ and $\exists\, \varepsilon \;\equiv\; 0$ and $\forall\, (X^\bullet 1) \;\equiv\; 1$ and $\exists\, (X^\bullet 0) \;\equiv\; 0$ and $\forall\, (X^\bullet 0) \;\equiv\; X = \emptyset$ and $\exists\, (X^\bullet 1) \;\equiv\; X \neq \emptyset$.

Semidistributivity rules. Examples are shown in the next table.

$\forall\, (p \mathbin{\overrightarrow{\wedge}} P) \;\equiv\; (p \wedge \forall\, P) \vee \mathcal{D}\, P = \emptyset$	$\forall\, (x : X \,.\, p \wedge q_x) \;\equiv\; (p \wedge \forall\, x : X \,.\, q_x) \vee X = \emptyset$
$\forall\, (p \mathbin{\overrightarrow{\Rightarrow}} P) \;\equiv\; p \Rightarrow \forall\, P$	$\forall\, (x : X \,.\, p \Rightarrow q_x) \;\equiv\; p \Rightarrow \forall\, (x : X \,.\, q_x)$
$\forall\, (P \mathbin{\overleftarrow{\Rightarrow}} p) \;\equiv\; \exists\, P \Rightarrow p$	$\forall\, (x : X \,.\, p_x \Rightarrow q) \;\equiv\; \exists\, (x : X \,.\, p_x) \Rightarrow q$

By duality: $\exists\, (p \mathbin{\overrightarrow{\wedge}} P) \;\equiv\; p \wedge \exists\, P$ and $\exists\, (p \mathbin{\overrightarrow{\Rightarrow}} P) \;\equiv\; (p \Rightarrow \exists\, P) \wedge \mathcal{D}\, P \neq \emptyset$ and $\exists\, (P \mathbin{\overleftarrow{\Rightarrow}} p) \;\equiv\; (\forall\, P \Rightarrow p) \wedge \mathcal{D}\, P \neq \emptyset$.

Instantiation, generalization and final examples. The following metatheorems, whose counterparts appear in logical textbooks as the axiom and inference rule for quantification, here again are direct consequences of (30), (4) and (5).

$$\begin{array}{lll} \text{(I)} & \forall\, P \Rightarrow x \in \mathcal{D}\, P \Rightarrow P\, x & \forall\, (x : X\,.\, p_x) \Rightarrow x \in X \Rightarrow p_x \\ \text{(G)} & q \Rightarrow x \in \mathcal{D}\, P \;\Rightarrow P\, x \;\vdash\; q \Rightarrow \forall\, P & q \Rightarrow x \in X \;\Rightarrow p_x \;\vdash\; q \Rightarrow \forall\, (x : X\,.\, p_x) \end{array}$$

This is the basis for proving all properties usually appearing in logic textbooks, as well as other important rules for practical applications, such as *trading*:

$$\begin{array}{lll} \textit{Trading}\ \forall & \forall\, (P \downarrow Q) \;\equiv\; \forall\, (Q \mathrel{\widehat{\Rightarrow}} P) & \forall\, (x : X \wedge q_x\,.\, p_x) \;\equiv\; \forall\, (x : X\,.\, q_x \Rightarrow p_x) \\ \textit{Trading}\ \exists & \exists\, (P \downarrow Q) \;\equiv\; \exists\, (Q \mathrel{\widehat{\wedge}} P) & \exists\, (x : X \wedge q_x\,.\, p_x) \;\equiv\; \exists\, (x : X\,.\, q_x \wedge p_x) \end{array}$$

We conclude with the *composition rule* $\forall\, P \;\equiv\; \forall\, (P \circ f)$ if $\mathcal{D}\, P \subseteq \mathcal{R}\, f$, whose pointwise variant is *dummy change*. The generic operators and the algebraic style were evident in all point-free formulas.

3.5 Formal semantics

The first examples illustrate applications for conventional programming languages, the last one pertains to graphical languages like LabVIEW.

Abstract syntax. The following is a functional unification of the various conventions introduced by Meyer [22] for expressing abstract syntax. For *aggregate constructs* and *list productions*, we use the functional Record and list types, which are just embodiments of the $\times$-operator as shown by $\mathsf{Record}\, F = \times (\bigcup F)$ and $A^* = \bigcup\, n : \mathbb{N}\,.\, \times (\square\, n^\bullet\, A)$. For *choice productions* where a disjoint union is needed, we define a generic operator $\big|$ such that, for any family F of types,

$$\Big| F = \bigcup\, x : \mathcal{D}\, F\,.\, \{x \mapsto y \mid y : F\, x\} \tag{31}$$

simply by analogy with $\bigcup\, F = \bigcup\, (x : \mathcal{D}\, F\,.\, F\, x) = \bigcup\, x : \mathcal{D}\, F\,.\, \{y \mid y : F\, x\}$. Using $x \mapsto y$ rather than the common x, y yields uniformity, which facilitates using the same three type operators for describing directory and file structures. For program semantics, however, the disjoint union often amounts to overengineering, since the syntactic categories seldom overlap and regular union suffices.

Typical examples are (with field labels from an enumeration type):

def *Program* := Record (*declarations* $\mapsto$ *Dlist*, *body* $\mapsto$ *Instruction*)
def *Dlist* := D^*
def D := Record ($v \mapsto$ *Variable*, $t \mapsto$ *Type*)
def *Instruction* := *Skip* $\cup$ *Assignment* $\cup$ *Compound* $\cup$ etc.

The few items left undefined in our examples are easily inferred by the reader. As mentioned, for disjoint union one can write *Skip* | *Assignment* | *Compound* etc.

Instances of programs, declarations, etc. can be defined as

$$\textbf{def}\, p : \mathit{Program}\ \textbf{with}\ p = \mathit{declarations} \mapsto dl \cup \mathit{body} \mapsto \mathit{instr}$$

Semantics. Casting Meyer's formulation [22] into a functional framework advantageously uses generic functionals. An example: for static semantics, *validity* of declaration lists (no double declarations) and the variable inventory are expressed by

$$\begin{aligned}&\textbf{def}\ \mathit{Vdcl} : \mathit{Dlist} \to \mathbb{B}\ \textbf{with}\ \mathit{Vdcl}\ dl = \mathsf{inj}\,(dl^T v)\\ &\textbf{def}\ \mathit{Var} : \mathit{Dlist} \to \mathcal{P}\ \mathit{Variable}\ \textbf{with}\ \mathit{Var}\ dl = \mathcal{R}\,(dl^T\ v)\end{aligned}$$

The *type map* of a valid declaration list (mapping variables to their types) is then

$$\begin{aligned}&\textbf{def}\ \mathit{typmap} : \mathit{Dlist}_{Vdcl} \ni dl \to \mathit{Var}\ dl \to \mathit{Tval}\ \textbf{with}\\ &\quad \mathit{typmap}\ dl = \mathit{tval} \circ (dl^T\ t) \circ (dl^T\ v)^-\end{aligned}$$

or, equivalently, $\mathit{typmap}\ dl = \bigcup d : \mathcal{R}\ dl\,.\,d\,v \mapsto \mathit{tval}\,(d\,t)$. A type map can be used as a context parameter for expressing validity of expressions and instructions, shown next.

In both static and dynamic semantics, the function *merge* obviates case expressions. For instance, assume

$$\begin{aligned}&\textbf{def}\ \mathit{Expression} := \mathit{Constant} \cup \mathit{Variable} \cup \mathit{Applic}\\ &\textbf{def}\ \mathit{Constant} := \mathit{IntCons} \cup \mathit{BoolCons}\\ &\textbf{def}\ \mathit{Applic} := \mathsf{Record}\,(\mathit{op} \mapsto \mathit{Operator}, \mathit{term} \mapsto \mathit{Expression},\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{term}' \mapsto \mathit{Expression})\end{aligned}$$

Letting $\mathit{Tmap} := \bigcup\ dl : \mathit{Dlist}_{Vdcl}\,.\,\mathit{typmap}\ dl$ (a style measure to avoid a dl parameter in the definition of *Texp* below) and $\mathit{Tval} := \{it, bt, ut\}$ (integer, boolean, undefined), the type of expressions is defined by

$$\begin{aligned}&\textbf{def}\ \mathit{Texp} : \mathit{Tmap} \to \mathit{Expression} \to \mathit{Tval}\ \textbf{with}\\ &\quad \mathit{Texp}\ tm \;=\; (c : \mathit{IntCons}\,.\,it) \cup (c : \mathit{BoolCons}\,.\,bt)\\ &\qquad\qquad \cup\ \ (v : \mathit{Variable}\,.\,ut) \lhd tm\\ &\qquad\qquad \cup\ \ (a : \mathit{Applic}\,.\,(a\ op \in \mathit{Arith\_op})\,?\,it \nmid bt)\end{aligned}$$

jointly with the expression validity function

$$\begin{aligned}&\textbf{def}\ \mathit{Vexp} : \mathit{Tmap} \to \mathit{Expression} \to \mathbb{B}\ \textbf{with}\\ &\quad \mathit{Vexp}\ tm \;=\; (c : \mathit{Constant}\,.\,1) \cup (v : \mathit{Variable}\,.\,v \in \mathcal{D}\ tm)\\ &\qquad\qquad \cup\ \ (a : \mathit{Applic}\,.\,\mathit{Vexp}\ tm\,(a\ term) \wedge \mathit{Vexp}\ tm\,(a\ term') \wedge\\ &\qquad\qquad\quad \mathit{Texp}\ tm\,(a\ term) = \mathit{Texp}\ tm\,(a\ term')\\ &\qquad\qquad\quad = (a\ op \in \mathit{Bool\_op})\,?\,bt \nmid it))\end{aligned}$$

Data flow languages. In Section 1 we have shown how the generic operators first arose in the semantics of descriptions of interconnected data flow components. Our original experiments were with the (textual) language Silage [16], but in view of its more widespread use we switched to LabVIEW, also because its graphical format provides an interesting testing ground for less common kinds of semantics. Here we provide an illustrative example of expressing the types of parametrized components.

Consider the LabVIEW block *Build Array*, which is interesting because it can be configured (via the menu) in different ways w.r.t. the number of inputs, and the kind of input, namely *element* or an *array*. An *element* is inserted in the array at the place indicated by the input wire, and an *array* is concatenated at that place. Fig. 6 shows the block configured for three element inputs and one array input. In the formal description, we let the configuration be described

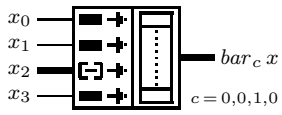

Figure 6. Typical *Build Array* configuration in LabVIEW

by a (nonempty) list of boolean values, 0 indicating an *element* input and 1 indicating an *array* input, e.g., $0, 0, 1, 0$ for the configuration of Fig. 6. An illustration is $bar_{0,0,1,0}\,(a, b, (c, d), e) = a, b, c, d, e$. With this convention, one can see that the type of the build array operator is $\mathbb{B}^+ \ni c \to \times\,(i : \mathcal{D}\,c\,.\,(V, V^*)\,(c\,i)) \to V^*$ for a given base type V or, removing i and adding polymorphism, $\mathbb{B}^+ \ni c \to \otimes\, V : \mathcal{T}\,.\,\times((V, V^*) \circ c) \to V^*$. The image part is $bar_c\, x = +\!\!+\, i : \mathcal{D}\,c\,.\,(\tau\,(x\,i), x\,i)\,(c\,i)$, noting that τ is the *singleton tuple injector* ($\tau\ x = 0 \mapsto x$ for any x). Hence

$$\begin{aligned}&\textbf{def}\ bar\_ : \mathbb{B}^+ \ni c \to \otimes\, V : \mathcal{T}\,.\,\times((V, V^*) \circ c) \to V^*\ \textbf{with}\\ &\quad bar_c = +\!\!+ \circ \parallel((\tau, id) \circ c).\end{aligned}$$

To programmers this (2/3) point-free style may look cryptic, but for data flow interpretation it is the evident one: bar_c is the cascade connection of a "concatenator" $+\!\!+$ preceded by a range of $\#\,c$ "preformatting" blocks $prf_c = (\tau, id) \circ c$ in parallel.

We oversimplified a little since the LabVIEW *Build Array* design is slightly unorthogonal (requiring a conditional, omitted to avoid clutter).

However, an important advantage of formalizing the semantics is precisely that it forces all hidden issues into the open. Especially in visual languages, subtleties are easily glossed over in user's guides. In fact, finding the exact semantics of some blocks in LabVIEW often requires some experimentation. Formalization improves precision.

3.6 Relational databases in functional style

We consider database systems intended to store information and present a convenient interface to the user for retrieving the desired parts and for constructing and manipulating "virtual tables" containing precisely the information of interest in tabular form.

Code	Name	Instructor	Prerequisites
CS100	Basic Mathematics for CS	R. Barns	none
MA115	Introduction to Probability	K. Jason	MA100
CS300	Formal Methods in Engineering	R. Barns	CS100, EE150
...	...	...	

A *relational database* presents the tables as relations. One can view each row as a tuple, and a collection of tuples of the same type as a relation.

However, in the traditional nonfunctional view of tuples, components can be accessed only by a separate indexing function using natural numbers. This is less convenient than, for instance, the column headings. The usual patch consists in "grafting" onto the relational scheme so-called *attribute names* corresponding to column headings. Disadvantages are that the mathematical model is not purely relational any more, and that operators for handling tables are ad hoc.

Viewing the table rows as *records* in functional sense of $\mathsf{Record}\,\mathrm{F} = \bigtimes(\bigcup \mathrm{F})$ based on (26) allows embedding in a more general framework with useful algebraic properties and inheriting the generic operators. For instance, the table shown can be declared as $GCI : \mathcal{P}\ CID$, a set of *course information descriptors* whose type is defined by

$$\textbf{def}\ CID := \mathsf{Record}\,(\mathsf{code} \mapsto \mathit{Code}, \mathsf{name} \mapsto \mathbb{A}^*, \mathsf{inst} \mapsto \mathit{Staff}, \mathsf{prrq} \mapsto \mathit{Code}^*).$$

The user accesses a database by suitably formulated *queries*, such as

(a) Who is the instructor for CS300?
(b) At what time is K. Jason normally teaching a course?
(c) Which courses is R. Barns teaching in the Spring Quarter?

The first query suggests a virtual *subtable* of GCI, containing only the columns labelled "Code" and "Instructor". The second requires joining table GCI with another table (containing the time schedule) in an appropriate way. All require selecting relevant rows. By definition, queries are expressed in some suitable *query language*.

The ability to handle virtual tables (in our formalism: sets of records) is clearly an important aspect of constructing queries, the main operations being *selection*, *projection* and *natural join* [13]. Our generic functionals directly provide this functionality.

- The *selection* operator (σ) selects for any table $S : \mathcal{P}\ R$ of records of type R precisely those records satisfying a given predicate $P : R \to \mathbb{B}$.

This is achieved by the set filtering operator in the sense that $\sigma\,(S, P) = S \downarrow P$. For instance, $\mathit{GCI} \downarrow (r : \mathit{CID} \,.\, r\,\mathsf{code} = \text{CS300})$ selects the row pertaining to question (a).

- The *projection* operator (π) yields for any table S of records a subtable containing only the columns corresponding to a given set F of field names. This is just a variant of function domain restriction that can be defined by $\pi\,(S, F) = \{r \rceil F \mid r : S\}$. For instance, $\pi\,(\mathit{GCI}, \{\mathsf{code}, \mathsf{inst}\})$ selects the columns pertaining to question (a). The complete question "Who is the instructor for CS300?" is represented by $\pi\,(\mathit{GCI} \downarrow (r : \mathit{CID} \,.\, r\,\mathsf{code} = \text{CS300}), \iota\,\mathsf{inst})$.
- The *("natural") join* operator ($\bowtie$) combines tables S and T by uniting the domains of the elements (field name sets), but keeping only those records for which the same field name in both tables have the same contents, i.e., only *compatible* records are combined: $S \bowtie T = \{s \cup t \mid (s, t) : S \times T \wedge s \mathbin{©} t\}$. This is precisely the *function type merge* from definition (29), i.e., $S \bowtie T = S \otimes T$. For instance, $\mathit{GCI} \bowtie \mathit{CS}$ combines table *GCI* with the *course schedule* table *CS* (exemplified below) in the desired manner.

Code	Semester	Day	Time	Location
CS100	Autumn	TTh	10:00	Eng. Bldg. 3.11
MA115	Autumn	MWF	9:00	Pólya Auditorium
CS300	Spring	TTh	11:00	Eng. Bldg. 1.20
...	...	...	...	...

Observe that $f \mathbin{©} g \wedge (f \cup g) \mathbin{©} h \;\equiv\; f \mathbin{©} g \wedge f \mathbin{©} h \wedge g \mathbin{©} h$ and, similarly (by symmetry), $(g \mathbin{©} h) \wedge f \mathbin{©} (g \cup h) \;\equiv\; g \mathbin{©} h \wedge f \mathbin{©} g \wedge f \mathbin{©} h$, which can be used to show $Ⓒ\,(f, g, h) \Rightarrow (f \cup g) \cup h = f \cup (g \cup h)$. Hence, although $\cup$ is not associative, $\otimes$ (and hence $\bowtie$) is.

3.7 Relation algebra

Our framework smoothly accommodates most relation algebras [1]. Relations are predicates on a Cartesian product. We let the usual notation $B \sim A$ stand for $B \times A \to \mathbb{B}$. Given a (dyadic) relation $R : B \sim A$, its *source* and *target* are defined by $\mathsf{src}\,R = A$ and $\mathsf{tgt}\,R = B$ (assuming both nonempty), and its *relational domain* and *relational range* by

$$\mathsf{dom}\,R = \{x : A \mid \exists\, y : B \,.\, R\,(y, x)\} \qquad \mathsf{ran}\,R = \{y : B \mid \exists\, x : A \,.\, R\,(y, x)\}.$$

As usual, declaring $—R— : B \sim A$ allows writing $y\,R\,x$ (infix operator).

The *relational representation* of a function f is the relation $\rho\,f$ on $\mathcal{R}\,f \times \mathcal{D}\,f$ with $y\,\rho f\,x \;\equiv\; y = f\,x$. Except for the occasional presence

of this operator, the familiar form and laws of the relational algebra are preserved. For instance, we define *composition* ($\bullet$) for relations by

$$\mathsf{tgt}\,(R \bullet S) = \mathsf{tgt}\,R \qquad \mathsf{src}\,(R \bullet S) = \mathsf{src}\,S$$
$$y\,R{\bullet}S\,x \;\equiv\; \exists\, z : \mathsf{src}\,R \cap \mathsf{tgt}\,S \,.\, y\,R\,z \wedge z\,S\,x$$

and the *subrelation* operator ($\sqsubseteq$) by

$$R \sqsubseteq S \quad \equiv \quad \forall\,(y,x) : \mathcal{D}\,R \,.\, R\,(y,x) \Rightarrow (y,x) \in \mathcal{D}\,S \wedge S\,(y,x).$$

The generic functionals provide additional forms and properties.

For instance, consider Backhouse's definition of a *pair algebra* [1] as a relation $R : B \sim A$ between posets $(A, \leq)$ and $(B, \preceq)$ such that there exist $f : A \to B$ and $g : B \to A$ satisfying $(\rho\, g)^{\smile} \bullet (\leq) = R = (\preceq) \bullet (\rho\, f)$. Straightforward calculation yields the equivalent characterization

$$(\leq) \circ (g \parallel id_A) = R = (\preceq) \circ (id_B \parallel f).$$

Similarly, consider the definition in [1] for an operator $\to$ such that $R{\to}S$ is a relation between functions defined by

$$\mathsf{tgt}\,(R{\to}S) = \mathsf{tgt}\,R \to \mathsf{tgt}\,S \qquad \mathsf{src}\,(R{\to}S) = \mathsf{src}\,R \to \mathsf{src}\,S$$
$$(R{\to}S)\,(g,f) \;\equiv\; R \sqsubseteq (\rho\, g)^{\smile} \bullet S \bullet (\rho\, f).$$

By calculation as before, the latter formula can be transformed into

$$(R{\to}S)\,(g,f) \;\equiv\; R \sqsubseteq S \circ (g \parallel f).$$

3.8 Final considerations: data as functions

Our framework uses a unified view of data structures as first-class functions, especially those for everyday use such as tuples, lists, records. This yields considerable conceptual and calculational unification, and in this context the *intrinsic polymorphism* to share all generic functionals.

The functional unification is *conservational* in the sense that it does not cause the loss of any properties that these structures may possibly have in the non-functional view; it only augments them. This observation holds in particular for decidability. Although, for instance, our definition of function equality is extensional and equality is undecidable for arbitrary functions in general, none of the issues that are decidable for the data structures in a non-functional framework (e.g., where they are defined recursively) become undecidable by the functional packaging.

Currently, however, even so-called *functional* languages do not support such data structures as functions. Considering the observed advantages, it is certainly worthwhile exploring this possibility.

A major design question is how to present the functional view in the programming language, including an elegant distinction between general functions and specific functions for which decidability issues are simple by state of the art compiler or interpreter technology. In the near future it will be investigated how far a shallow embedding of some of these concepts in Haskell is possible. Even in the long run there will always be restrictions imposed by the implementation, but this is a moving situation, as many difficult implementation problems have been overcome in the past decades. Further research in this direction is needed.

In the meantime, the absence of the functional view on these data types in *implementations* need not prevent us from exploiting this view in *reasoning* about functional programs. We illustrate this by an example about infinite lists, where a theorem that is normally proven by co-induction is proven here by means of generic functionals.

The following problem was suggested to me by Steven Johnson. Given the following definition for an interleaving variant of `zip`

$$\texttt{zap [a : x] [b : y] = a : b : zap x y}, \tag{32}$$

prove that

$$\texttt{map f (zap x y) = zap (map f x) (map f y)}. \tag{33}$$

With infinite sequences as functions with domain $\mathbb{N}$, the definition amounts to $zap\,x\,y\,(2\cdot n) = x\,n$ and $zap\,x\,y\,(2\cdot n+1) = y\,n$ and the proof of (33) is immediate by simply using the laws of function composition, recalling that $map\,f\,x = f \circ x = \overline{f}\,x$. However, such a proof uses a domain variable and case distinction (for even and odd argument).

Observing that $zap\,x\,y\,n = (x\,\frac{n}{2}, y\,\frac{n-1}{2})\,(n \dagger 2)$ (where $\dagger$ is the modulo operator), we can write the definition without domain variable as

$$zap\,x\,y \quad = \quad \|\,(x \circ \alpha, y \circ \beta)^U \gamma \tag{34}$$

using calculations very similar to those in (2). For the auxiliary functions, $\alpha : \{2\cdot n \mid n : \mathbb{N}\} \to \mathbb{N}$ with $\alpha\,n = n/2$ and $\beta : \{2\cdot n+1 \mid n : \mathbb{N}\} \to \mathbb{N}$ with $\beta\,n = (n-1)/2$, whereas $\gamma : \mathbb{N} \to \mathbb{B}$ with $\gamma\,n = n \dagger 2$. Note that $x \circ \alpha$ and $y \circ \beta$ are not sequences, but the functional formalism takes this in its stride. There is an obvious variant using flooring ($\lfloor\ \rfloor$), which allows replacing $—^U$ by $—^T$. This is rather artificial, so we do not use it.

One can prove that, for any function f and function family F

$$\overline{\overline{f}}\,(\|\,F^U) \quad = \quad \|\,(\overline{\overline{f}}\,F)^U. \tag{35}$$

This is left as an exercise, the interesting part being the types.

The proof of the original theorem now proceeds as follows.

$$\begin{array}{lcll} map\ f\ (zap\ x\ y) & = & \langle map\ f = \overline{f}\ , (34)\rangle & \overline{f}\ (\|\ (x \circ \alpha, y \circ \beta)^U \gamma) \\ & = & \langle \overline{f}\ (g\ x) = \overline{\overline{f}}\ g\ x\rangle & \overline{\overline{f}}\ (\|\ (x \circ \alpha, y \circ \beta)^U)\ \gamma \\ & = & \langle \text{Theorem (35)}\rangle & \|\ (\overline{\overline{f}}\ (x \circ \alpha, y \circ \beta))^U\ \gamma \\ & = & \langle \overline{f}\ (x, y) = f\ x, f\ y\rangle & \|\ (\overline{f}\ (x \circ \alpha), \overline{f}\ (y \circ \beta))^U\ \gamma \\ & = & \langle \overline{f}\ (g \circ h) = \overline{f}\ g \circ h\rangle & \|\ (\overline{f}\ x \circ \alpha, \overline{f}\ y \circ \beta)^U\ \gamma \\ & = & \langle map\ f = \overline{f}\ , (34)\rangle & zap\ (map\ f\ x)\ (map\ f\ y) \end{array}$$

At first sight, such a proof looks cryptic, but so do proofs in all point-free styles (including the Bird-Meertens formalism). Here we have the additional advantage that the generic functionals are equally useful in other areas of mathematics, not even discrete. This makes developing some practice with their use and manipulation quite rewarding.

More important than the proof of theorem (33) itself is establishing theorem (35) as a functional generalization thereof. The fact that defining sequences as functions allows handling "sequences with holes" (such as $x \circ \alpha$ and $y \circ \beta$) without any problem may also prove useful.

Conclusion

We have shown how a small collection of (roughly a dozen) functionals is directly useful in a wide range of applications, from continuous mathematics to programming.

This generic nature depends on two elements: a unifying view on the application-specific objects by (re)defining them as functions, and the judicious specification of the domains for the result functions 'produced' by the functionals. Even with this pointwise bookkeeping in the design, the functionals themselves conveniently support the point-free style via useful algebraic properties and rules for calculational reasoning.

The examples shown cover the basic mathematics of software engineering (predicate calculus), aspects of programming languages (formal semantics and unifying design) and quite different application areas (data flow systems and relational data bases). Not demonstrated here, but equally interesting, are the many other applications in non-discrete ("continuous") mathematics.

A valuable side-effect for organizing human knowledge is that similarities between disparate fields can be exploited (without detracting from essential differences), for instance, in reducing conceptual and notational overhead, and making the transition easier by providing analogies, not just in a vague informal sense, but on a mathematical basis.

References

[1] Kevin and Roland Backhouse, "Logical Relations and Galois Connections", in: Eerke A. Boiten and Bernhard Möller, eds., *Mathematics of Program Construction (MPC2002)*, pp. 23–39. LNCS 2386, Springer-Verlag, Berlin (July 2002)

[2] Henk P. Barendregt, *The Lambda Calculus — Its Syntax and Semantics*, North-Holland, Amsterdam (1984)

[3] Michael Barr, Charles Wells, *Category Theory for Computing Science* (2nd. ed.). Prentice Hall International Series in Computer Science, London (1995)

[4] Richard Bird, *Introduction to Functional Programming using Haskell*. Prentice Hall International Series in Computer Science, London (1998)

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

[6] Raymond T. Boute, "On the requirements for dynamic software modification", in: C. J. van Spronsen and L. Richter, eds., *MICROSYSTEMS: Architecture, Integration and Use* (Euromicro Symposium 1982), pp. 259-271. North Holland, Amsterdam (1982)

[7] 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)

[8] 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)

[9] Raymond T. Boute, *Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I: Basic Mathematics*. Course text, Ghent University (2001)

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

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

[12] 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)

[13] David Gries, Fred B. Schneider, *A Logical Approach to Discrete Math*. Springer-Verlag, Berlin (1994)

[14] David B. Guralnik, ed., *Webster's New World Dictionary of the American Language*. William Collins + World Publishing Co., Inc., Cleveland, Ohio (1976)

[15] John V. Guttag, et al., "The design of data type specifications", in: Raymomd T. Yeh, ed., *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, pp. 60–79. Prentice-Hall, Inc., Englewood Cliffs, N. J. (1978)

[16] Paul N. Hilfinger, *Silage Reference Manual*, Univ. of California, Berkeley (1993)

[17] 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)

[18] Paul Hudak, John Peterson and Joseph H. Fasel, *A Gentle Introduction to Haskell 98*. `http://www.haskell.org/tutorial/` (Oct. 1999)

[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] Lawrence T. Lorimer, *New Webster's Dictionary and Thesaurus of the English Language*. Lexicon Publications, Inc., Danbury, CT (1995)

[22] Bertrand Meyer, *Introduction to the Theory of Programming Languages*. Prentice Hall, New York (1991)

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

[24] 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)

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

[26] John Rushby, Sam Owre and Natarajan Shankar, "Subtypes for Specifications: Predicate Subtyping in PVS", *Transactions on Software Engineering vol. 24*, 9, pp. 709–720 (September 1998)

[27] J. Michael Spivey, *The Z notation: A Reference Manual*. Prentice-Hall (1989)

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

[29] R. D. Tennent, *Semantics of Programming Languages*. Prentice Hall, New York (1991)

[30] Frank van den Beuken, *A Functional Approach to Syntax and Typing*, PhD thesis. School of Mathematics and Informatics, University of Nijmegen (1997)

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