

# Concrete Generic Functionals: Principles, Design and Applications<sup>1</sup>

Raymond Boute

INTEC — Ghent University

---

## Overview

0. **Introduction: motivation**
1. **Functionals for transformation** (induction: collecting tools)
2. **Making the functionals generic** (design: generalizing the tools)
3. **Applications in programming** (deduction: applying the new tools)
4. **Conclusion**

Separately (time allowing): **Formal calculation examples** (from preceding topics)

---

<sup>1</sup>Prepared for the 2002 Working Conference on Generic Programming.

## 0 Motivation

- **Genericity in programming** (reference: conference announcement)
  - 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 is a fortiori relevant to declarative formalisms**

Replacing “programs” by (general) “functions”, we propose correspondingly:

  - a. Generality by recasting traditionally non-functional objects uniformly as functions
  - b. Designing functionals without argument restrictions,  $\Rightarrow$  applicable to all objects captured by the functional uniformization (*intrinsic polymorphism*)
  - c. Parameters of functionals are other function(al)s, e.g., type constructors  
Expressions in both point-wise form (with variables) and point-free form

- **Observation regarding benefits to programming** (Reynolds):

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

- **Approach: “concrete” variant of category theory**

- Domains and codomains may depend on the mappings  
e.g., domain of  $f \circ g$  not necessarily domain of  $g$
- Yet algebraic properties expressed by simple formulas remain valid  
e.g.,  $f \circ (g \circ h) = (f \circ g) \circ h$  (hence **calculation rules remain simple**)
- Motivated by unification of basic formalisms for “continuous” and discrete mathematics and systems modelling.

# 1 Functionals for transformation

## 1.1 Functional Mathematics as the unifying principle

- **Principle:** (re)defining mathematical objects, whenever feasible, as functions.  
Remark: found especially advantageous where this is not yet a commonplace
- **Advantages**
  - **Conceptual:** uniformity in treatment while respecting essential differences
  - **Practical:** sharing general-purpose (**generic**) operators over functions
- **Example: sequences** (generic for tuples, lists, etc., “homogeneous” or not)
  - **Why this example?** “interface” between discrete and continuous math
  - **Wide ramifications:** (also in the design of mathematical software)
    - \* Removal of all conventions having poor calculational properties  
Worst kind of violation: against Leibniz’s principle. **Example: ellipsis**  
 $a_0 + a_1 + \cdots + a_7$  where  $a_i = i^2$  yields  $0 + 1 + \cdots + 49$
    - \* Replacement by well-defined operators and algebraic calculation rules

- **Example (continued): sequences as functions**

- **Principle:**  $(a, b, c)_0 = a$  and  $(a, b, c)_1 = b$  and  $(a, b, c)_2 = c$
- **Intuitively trivial, yet:**
  - \* commonly handled as entirely or subtly distinct from functions,
  - \* in the few exceptions, functional properties left unexploited.
- **Examples** of (underexploited) functional properties of sequences
  - \* **Inverses:**  $(a, b, c, d)^{-c} = 2$  (provided  $c \notin \{a, b, d\}$ )
  - \* **Composition:**  $(0, 3, 5, 7) \circ (2, 3, 1) = 5, 7, 3$  and  $f \circ (x, y) = f x, f y$
  - \* **Transposition:**  $(f, g)^T x = f x, g x$

Seemingly secondary, but very useful once discovered

- **Not obtainable by the usual formal treatments of lists, e.g.,**
  - \* recursive definition:  $[]$  is a list and, if  $x$  is a list, so is  $\text{cons } a x$
  - \* index function separate:  $\text{ind } (\text{cons } a x) (n + 1) = \text{ind } x n$  e.g., in Haskell:  $\text{ind } [a:x] 0 = a$  and  $\text{ind } [a:x] (n + 1) = x n$

- **Function(al)s for sequences** (“user library”)

- Domain specification: “block”  $\square$

$$\square n = \{k : \mathbb{N} \mid k < n\} \text{ for } n : \mathbb{N} \text{ or } n := \infty$$

- Length:  $\#$

$$\# x = n \equiv \mathcal{D} x = \square n, \text{ equivalently: } \# x = \square^- (\mathcal{D} x)$$

- Prefix:  $\succ$  characterized by domain and mapping:

$$\# (a \succ x) = \# x + 1 \quad i \in \mathcal{D} (a \succ x) \Rightarrow (i = 0) ? a \dagger x (i - 1)$$

Note: for the conditional expression:  $c ? b \dagger a = (a, b) c$

- Shift:  $\sigma$  characterized by domain and mapping: for nonempty  $x$ ,

$$\# (\sigma x) = \# x - 1 \quad i \in \mathcal{D} (\sigma x) \Rightarrow \sigma x i = x (i + 1)$$

- The usual induction principle is a *theorem* (not an axiom)

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

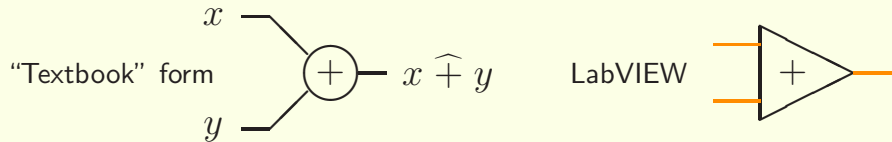
## 1.2 Practical need for point-free formulations

- **Point-free formulations** traditionally seen as only relevant to pure theory. Any (general) practical formalism needs **both** point-wise and point-free style.
- **Example: signal flow systems:** assemblies of interconnected components. Dynamical behavior modelled by functionals from input to output signals. Here taken as an opportunity to introduce “embryonic” generic functionals, i.e., arising in a specialized context, to be made generic later for general use. Extra feature: **LabVIEW** (a graphical language) taken as an opportunity for
  - Presenting a language with uncommon yet interesting semantics
  - Using it as one of the application examples of our approach (functional description of the semantics using generic functionals)

- **Basic building blocks**

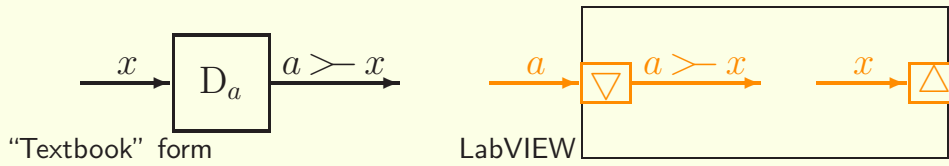
- Memoryless devices realizing arithmetic operations

- \* Sum (product, . . . ) of signals  $x$  and  $y$  modelled as  $(x \hat{+} y) t = x t + y t$
    - \* Explicit *direct extension* operator  $\hat{+}$  (in engineering often left implicit)



- Memory devices: latches (discrete case), integrators (continuous case)

$$D_a x n = (n = 0) ? a \dagger x (n - 1) \text{ or, without time variable, } D_a x = a \succ x$$



- **Time is not structural**

Hence transformational design = elimination of the time variable

### 1.3 A transformational design example

- **From specification to realization**

- Recursive specification: given set  $A$  and  $a : A$  and  $g : A \rightarrow A$ ,

$$\text{def } f : \mathbb{N} \rightarrow A \text{ with } f\ n = (n = 0) ? a \dagger g(f(n - 1)) \quad (1)$$

- Calculational transformation

$$\begin{aligned} f\ n &= \langle \text{Def. } f \rangle (n = 0) ? a \dagger g(f(n - 1)) \\ &= \langle \text{Def. } \circ \rangle (n = 0) ? a \dagger (g \circ f)(n - 1) \\ &= \langle \text{Def. } D \rangle D_a(g \circ f)\ n \\ &= \langle \text{Def. } \overline{=} \rangle D_a(\overline{g}\ f)\ n \\ &= \langle \text{Def. } \circ \rangle (D_a \circ \overline{g})\ f\ n, \end{aligned} \quad (2)$$

yielding the **fixpoint equation**  $f = (D_a \circ \overline{g})\ f$  by function extensionality.

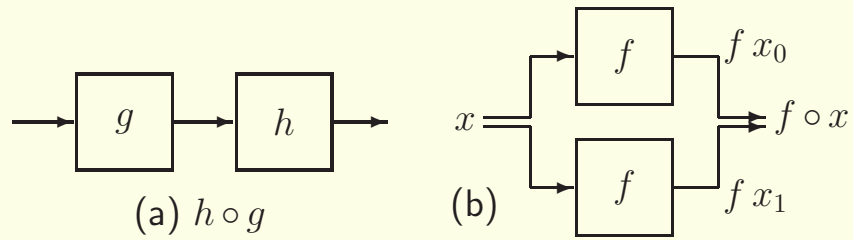
- **Functionals introduced** (types **designed** later during the generification)

- Function composition:  $\circ$ , defined by  $(f \circ g)\ x = f(g\ x)$

- Direct extension (1 argument):  $\overline{=}$ , defined by  $\overline{g}\ x = g \circ x$

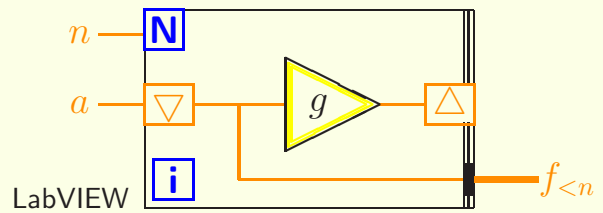
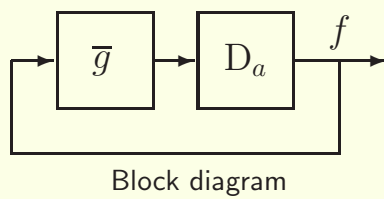
- **Structural interpretations** of composition and the fixpoint equation

- Structural interpretations of composition: (a) cascading; (b) replication



Example property:  $\overline{h \circ g} = \overline{h} \circ \overline{g}$  (proof: exercise)

- Immediate structural solution for the fixpoint equation  $f = (D_a \circ \overline{g}) f$



- **A third operator: transposition** (already seen: composition, extension)

- **Purpose:** swapping the arguments of a higher-order function

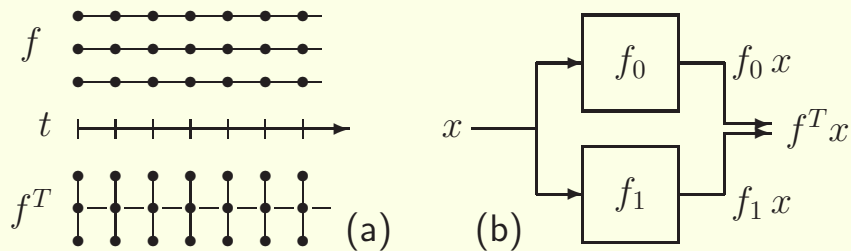
$$f^T y x = f x y$$

- Nomenclature obviously borrowed from matrix theory

- **Structural interpretations:**

- From a family of signals to a tuple-valued signal,

- Signal fanout



- Subsumes the zip operator from functional programming

$$\text{zip}[[a,b,c],[a',b',c']] = [[a,a'],[b,b'],[c,c']]$$

assuming lists are functions, and up to currying.

- **Calculating with transposition, composition and direct extension**

- **Duality composition – transposition:** assuming  $x$  not free in  $M$ ,

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

Also:  $f \circ (x, y, z) = f x, f y, f z$  and  $(f, g, h)^T x = f x, g x, h x$

- **Generalizing direct extension** to an arbitrary number of arguments:

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

(hints added orally) hence  $f \hat{\star} f' = (\star) \circ (f, f')^T$  by extensionality.

Therefore we define the generalized direct extension operator  $\overset{\leq}{\star}$  by

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

for any function  $g$  and any family  $h$  of functions.

## 2 Making the functionals generic

### 2.1 Conventions for functions

- **Function** = domain ( $\mathcal{D} f$ ) and mapping (unique  $f x$  for every  $x$  in  $\mathcal{D} f$ ).
- **Function equality** = equality of the domains and the mappings. Formally:
  - **Leibniz's principle**:  $f = g \Rightarrow \mathcal{D} f = \mathcal{D} g \wedge f x = g x$  or, with “guards”,

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

Equivalently:

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

- **Converse (function extensionality)**: using a fresh dummy  $x$ ,

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

- **Style of definition** (awaiting quantifiers)

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

Conventions:  $x$  a variable,  $X$  a set,  $p_x$  and  $q_{f,x}$  propositions.

The subscripts are comments specifying which of  $x$  and  $f$  may occur free.

**Example:** the *constant function specifier*  $\bullet$ : for any set  $X$  and any  $e$ ,

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

- **Denoting functions by typed abstractions**

- **Principle:** recall the style of definition by two 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  $x \in \mathcal{D} f \Rightarrow q_{f,x}$

If  $q$  has the explicit form  $f x = e$ , then we (can) denote function  $f$  by

$$x : X \wedge p . e$$

The so-called *filter*  $\wedge p$  is optional, and  $x : X . e$  stands for  $x : X \wedge 1 . e$ .

- **Remark:** supports “common” (point-wise) notations like  $\forall x : \mathbb{R} . x^2 \geq 0$ .

- **Axioms** (a typed lambda calculus): for the domain and the mapping

$$d \in \mathcal{D} (x : X \wedge p . e) \equiv d \in X \wedge p_d^x$$

$$d \in \mathcal{D} (x : X \wedge p . e) \Rightarrow (x : X \wedge p . e) d = e_d^x \quad (7)$$

- **Examples:**

- \*  $X \bullet e = x : X . e$  for fresh  $x$  (not free in  $e$ )

- \*  $(n : \mathbb{N} . 2 \cdot n) = (n : \mathbb{Z} \wedge n \geq 0 . 2 \cdot n)$

- **Remark:** the colon ( $:$ ) yields **unambiguous** binding, unlike overloading  $\in$ .

- **Intermezzo: notational ramifications of “functional mathematics”**

– Entire notational framework needs/uses only **four** constructs:

identifier	function application	abstraction	tuple denotation
$n, succ, +, \forall$	$f\ n, x + y, \forall x : X . p$	$x : X \wedge p . e$	$a, b, c$

Properties:

- \* Synthesizes common conventions, minus ambiguities/inconsistencies
  - \* Adds forms of expression (e.g., point-free) and **precise calculation rules**
- **Case example: the function range operator  $\mathcal{R}$**
- \* Common practice: overloaded use of  $\in$  for binding, e.g.,  $\forall x \in X . p$   
 Problem:  $\{m \in \mathbb{N} \mid m < n\}$  and  $\{2 \cdot m \mid m \in \mathbb{N}\}$  may **seem** harmless, **but**: what about  $\{x \in X \mid x \in Y\}$ ? Is it  $X \cap Y$  or a **subset of  $\mathbb{B}$** ?
  - \* Solution: *range* operator with axiom  $y \in \mathcal{R} f \equiv \exists x : \mathcal{D} f . y = f x$ .
  - \* Equivalent symbol:  $\{—\}$ . Application to tuples and abstractions yields familiar form and meaning. **Examples:**  $\{a, b, c\}$  and  $\{n : \mathbb{Z} . 2 \cdot n\}$
  - \* Using  $x : X \mid p$  as syntactic sugar for  $x : X \wedge p . x$  does the same for expressions like  $\{m : \mathbb{N} \mid m < n\}$ .

## 2.2 Design criteria and method for generic functionals

- **Reason for making functionals generic:** in functional mathematics, they become shared by many more kinds of objects than usual.
- **Shortcomings of traditional operators:** restrictions on the arguments, e.g.,
  - the usual  $f \circ g$  requires  $\mathcal{R} g \subseteq \mathcal{D} f$ , in which case  $\mathcal{D} (f \circ g) = \mathcal{D} g$
  - the usual  $f^{-}$  requires  $f$  injective, in which case  $\mathcal{D} f^{-} = \mathcal{R} f$
- **Approach used here:**
  - No restrictions on the argument function(s)
  - Instead, refine domain of the result function (say,  $f$ ) via its domain axiom  $x \in \mathcal{D} f \equiv x \in X \wedge p$  ensuring that, in the mapping axiom  $x \in \mathcal{D} f \Rightarrow q$ ,  $q$  does not contain out-of-domain applications in case  $x \in \mathcal{D} f$  (**guarded**)
  - Conservational, i.e.,
    - For previously known functionals, if the traditional restriction on the argument is satisfied anyway, the generalization yields the “usual” case.
    - For new functionals (extension, transposition, merge etc.): design freedom

## 2.3 Functionals designed generically

- **Filtering ( $\downarrow$ )** Function filtering generalizes  $\eta$ -conversion  $f = x : \mathcal{D} f . f x$ : for any function  $f$  and predicate  $P$ ,

$$f \downarrow P = x : \mathcal{D} f \cap \mathcal{D} P \wedge P x . f x \quad (8)$$

Set filtering:  $x \in X \downarrow P \equiv x \in X \cap P \wedge P x$ .

Shorthand:  $a_b$  for  $a \downarrow b$ , yielding convenient abbreviations like  $f_{<n}$  and  $\mathbb{R}_{\geq 0}$ .

- **Function restriction ( $\upharpoonright$ )**: the usual domain restriction:

$$f \upharpoonright X = f \downarrow (X \bullet 1). \quad (9)$$

- **Composition ( $\circ$ )** generalizes traditional composition: for any functions  $f$  and  $g$  (without restriction),

$$\begin{aligned} x \in \mathcal{D}(f \circ g) &\equiv x \in \mathcal{D} g \wedge g x \in \mathcal{D} f \\ x \in \mathcal{D}(f \circ g) &\Rightarrow (f \circ g) x = f(g x). \end{aligned} \quad (10)$$

Conservational: if the traditional requirement  $\mathcal{R} g \subseteq \mathcal{D} f$  is satisfied, then  $\mathcal{D}(f \circ g) = \mathcal{D} g$ .

- **Inversion ( $\text{---}^-$ )** For any function  $f$ ,

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

For  $\text{Bdom}$  (*bijection domain*) and  $\text{Bran}$  (*bijection range*):

$$\text{Bdom } f = \{x : \mathcal{D} f \mid \forall x' : \mathcal{D} f . f x' = f x \Rightarrow x' = x\} \quad (12)$$

$$\text{Bran } f = \{f x \mid x : \text{Bdom } f\}. \quad (13)$$

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

- **Dispatching ( $\&$ ) and parallel ( $\parallel$ )** For any functions  $f$  and  $g$ ,

$$\mathcal{D}(f \& g) = \mathcal{D} f \cap \mathcal{D} g \quad x \in \mathcal{D}(f \& g) \Rightarrow (f \& g) x = f x, g x \quad (14)$$

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

- **(Duplex) direct extension ( $\hat{=}$ )** For any infix operator  $\star$ , functions  $f, g$ ,

$$\begin{aligned} x \in \mathcal{D}(f \hat{\star} g) &\equiv x \in \mathcal{D}f \cap \mathcal{D}g \wedge (fx, gx) \in \mathcal{D}(\star) \\ x \in \mathcal{D}(f \hat{\star} g) &\Rightarrow (f \hat{\star} g)x = fx \star gx. \end{aligned} \quad (16)$$

Equivalently,  $f \hat{\star} g = x : \mathcal{D}f \cap \mathcal{D}g \wedge (fx, gx) \in \mathcal{D}(\star) . fx \star gx$ .

If  $x \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow (fx, gx) \in \mathcal{D}(\star)$  then  $f \hat{\star} g = x : \mathcal{D}f \cap \mathcal{D}g . fx \star gx$ .

**Example: equality:**  $(f \hat{=} g) = x : \mathcal{D}f \cap \mathcal{D}g . fx = gx$

**Half direct extension:** for any function  $f$  and any  $x$ ,

$$f \overleftarrow{\star} x = f \hat{\star} \mathcal{D}f^\bullet x \quad \text{and} \quad x \overrightarrow{\star} f = \mathcal{D}f^\bullet x \hat{\star} f.$$

**Simplex direct extension ( $\overline{=}$ ):** recall  $\overline{f}g = f \circ g$ .

- **Function override ( $\otimes$  and  $\oslash$ )** For any functions  $f$  and  $g$ ,  
 $g \otimes f = f \oslash g$  and

$$\begin{aligned} \mathcal{D}(f \oslash g) &= \mathcal{D}f \cup \mathcal{D}g \\ x \in \mathcal{D}(f \oslash g) &\Rightarrow (f \oslash g)x = x \in \mathcal{D}f ? fx \dagger gx \end{aligned} \quad (17)$$

- **Function merge ( $\cup$ )** For any functions  $f$  and  $g$ ,

$$\begin{aligned} x \in \mathcal{D}(f \cup g) &\equiv x \in \mathcal{D}f \cup \mathcal{D}g \wedge (x \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow fx = gx) \\ x \in \mathcal{D}(f \cup g) &\Rightarrow (f \cup g)x = x \in \mathcal{D}f ? fx \dagger gx. \end{aligned} \quad (18)$$

- **Relational functionals: compatibility ( $\odot$ ), subfunction ( $\subseteq$ )**

$$\begin{aligned} f \odot g &\equiv f \upharpoonright \mathcal{D}g = g \upharpoonright \mathcal{D}f \\ f \subseteq g &\equiv f = g \upharpoonright \mathcal{D}f. \end{aligned}$$

**Examples** of typical algebraic properties:

$$f \subseteq g \equiv \mathcal{D}f \subseteq \mathcal{D}g \wedge f \odot g \text{ and } f \odot g \Rightarrow f \oslash g = f \cup g = f \otimes g$$

$\subseteq$  is a partial order (reflexive, antisymmetric, transitive)

$$\text{For equality: } f \odot g \equiv \forall (f \cong g) \text{ and } f = g \equiv \mathcal{D}f = \mathcal{D}g \wedge f \odot g.$$

## 2.4 Elastic extensions for generic operators

- **Elastic operators in general**

- **Principle:** functionals replacing the common ad hoc abstractors, e.g.,

$$\forall x : X \quad \sum_{i=m}^n \quad \lim_{x \rightarrow a} .$$

Together with function abstraction: yield familiar expressions

$$\forall x : X . P x \quad \sum_{i:m..n} . x_i \quad \lim (x : \mathbb{R} . f x) a$$

For less casual “users”, they also yield point-free forms such as

$$\forall P \quad \sum x \quad \lim f a$$

and direct applicability to sequences (tuples, lists etc.)

$$\forall (x, y, z) = x \wedge y \wedge z \quad \sum (x, y, z) = x + y + z$$

Example:  $\forall x : \mathbb{R} . x^2 \geq 0$  obtains familiar form and meaning,  
but also a novel decomposition:  $\forall$  and  $x : \mathbb{R} . x^2 \geq 0$ , are both *functions*.

- **General importance** Same functionals for point-free and point-wise style.

- **Some observations**

- **Examples** of definitions: quantifiers  $\forall P \equiv P = \mathcal{D} P \bullet 1$  and summation

$$\Sigma \varepsilon = 0 \quad \Sigma (a \mapsto c) = c \quad \Sigma (f \cup g) = \Sigma f + \Sigma g \quad (19)$$

for any  $a$ , any numeric  $c$  and any number-valued functions  $f$  and  $g$  with finite nonintersecting (but otherwise arbitrary) domains.

Note: (19) shows  $\Sigma$  to be a *merge homomorphism*, a generalization of the well-known  $h(x ++ y) = hx \oplus hy$  for data structures  $x$  and  $y$ .

- **Subsumes the “Eindhoven notation”**  $Q x : P.x : f.x$ 
  - \* More general underlying different principle and design
  - \* Supports also the point-free style
  - \* No algebraic restrictions (associativity, commutativity etc.)  
R. Backhouse also noted that certain restrictions are unnecessary; elastic operators go further
  - \* Not restricted to discrete math (useful in mathematical analysis).

- **Variadic operators, elastic extensions and variadic shorthand**

- **Variadic operator**: elastic operator with meaningful restriction to sequences

- **Elastic extension** of an infix operator  $\star$ :

- \* **Definition**: any elastic operator  $F$  satisfying  $F(x, y) = x \star y$

- \* **Examples**:  $\forall(x, y) \equiv x \wedge y$  and  $\Sigma(x, y) = x + y$

- \* **Important**: not unique, leaving room for judicious design

- **Variadic shorthand**:

- \* **Definition**: argument/operator alternation  $x \star y \star z$  denoting  $F(x, y, z)$

- \* **Examples**:  $x \wedge y \wedge z \equiv \forall(x, y, z)$  and  $x + y + z = \Sigma(x, y, z)$

More interesting: with predicates *con* (*constant*) and *inj* (*injective*),

$$x = y = z \equiv \text{con}(x, y, z)$$

$$x \neq y \neq z \equiv \text{inj}(x, y, z).$$

Gives  $x \neq y \neq z$  useful meaning (impossible with “old” conventions).

- \* Choice of  $F$  for commonly used  $\star$ : **conservative**. **Example**: for **associative**  $\star$ , require that the restriction of  $F$  to lists is a *list homomorphism*, i.e.,  $F(x ++ y) = F x \star F y$  for any lists  $x$  and  $y$  in  $\mathcal{D} F$ .

- **Function transposition ( $\text{---}^T$ )** Recall the image definition  $f^T y x = f x y$ . Making  $\text{---}^T$  generic requires decision about  $\mathcal{D} f^T$  for *any* function family  $f$ .

- **Intersecting variant ( $\text{---}^T$ )** Motivation:  $\hat{g} f = g \circ f^T$  uses intersection. This is captured by taking  $\mathcal{D} f^T = \cap x : \mathcal{D} f . \mathcal{D}(f x)$  or  $\mathcal{D} f^T = \cap (\mathcal{D} \circ f)$ . Hence  $f^T = y : \cap (\mathcal{D} \circ f) . x : \mathcal{D} f . f x y$  or, with separate axioms

$$\begin{aligned} \mathcal{D} f^T &= \cap x : \mathcal{D} f . \mathcal{D}(f x) \\ y \in \mathcal{D} f^T &\Rightarrow \mathcal{D}(f^T y) = \mathcal{D} f \\ &\wedge (x \in \mathcal{D}(f^T y) \Rightarrow f^T y x = f x y) \end{aligned} \quad (20)$$

- **Uniting variant ( $\text{---}^U$ )**, taking  $\mathcal{D} f^U = \cup (\mathcal{D} \circ f)$ . Elaboration using the design criterion yields  $f^U = y : \cup (\mathcal{D} \circ f) . x : \mathcal{D} f \wedge y \in \mathcal{D}(f x) . f x y$  or

$$\begin{aligned} \mathcal{D} f^U &= \cup x : \mathcal{D} f . \mathcal{D}(f x) \\ y \in \mathcal{D} f^U &\Rightarrow \mathcal{D}(f^U y) = \{x : \mathcal{D} f \mid y \in \mathcal{D}(f x)\} \\ &\wedge (x \in \mathcal{D}(f^U y) \Rightarrow f^U y x = f x y) \end{aligned} \quad (21)$$

- **Variadic shorthand** Observation:  $(g \& h) x i = (g, h) i x$  for  $i : \{0, 1\}$ . Design decision:  $f \& g \& h = (f, g, h)^T$ .

- **Elastic parallel ( $\parallel$ )** For any function family  $F$  and function  $f$ ,

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

This can be seen as a typed variant of the well-known **S**-combinator.

- **Elastic merge** For any function family  $f$ ,

$$y \in \mathcal{D} (\cup f) \equiv \quad (23)$$

$$y \in \cup (\mathcal{D} \circ f) \wedge \forall (x, x') : (\mathcal{D} f)^2 . y \in \mathcal{D} (f x) \cap \mathcal{D} (f x') \Rightarrow f x y = f x' y$$

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

Interesting examples: for **any** function  $g$ ,

$$g = \cup x : \mathcal{D} g . x \mapsto g x \quad \text{and} \quad g^- = \cup x : \mathcal{D} g . g x \mapsto x$$

Illustrates how the generic design criterion leads to fine operator intermeshing.

- **Elastic compatibility** For any function family  $f$

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

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

## 2.5 A generic functional refining function types

- **Coarse typing: the function arrow** defined by

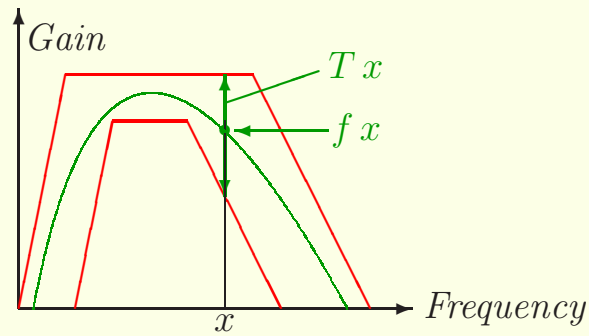
$$f \in X \rightarrow Y \equiv \mathcal{D}f = X \wedge \mathcal{R}f \subseteq Y$$

- **The function approximation paradigm for range refinement**

- **Purpose:** formalizing *tolerances* for *functions*
- **Principle:** *tolerance function*  $T$  specifies for every domain value  $x$  the set  $Tx$  of allowable values. Note:  $\mathcal{D}T$  serves as the domain specification.  
Formalized: a function  $f$  *meets* tolerance  $T$  iff

$$\mathcal{D}f = \mathcal{D}T \wedge (x \in \mathcal{D}f \cap \mathcal{D}T \Rightarrow f x \in T x).$$

- Pictorial representation (example: radio frequency filter characteristic).



$$\mathcal{D}f = \mathcal{D}T \wedge (x \in \mathcal{D}f \cap \mathcal{D}T \Rightarrow f x \in T x).$$

- Generalized Functional Cartesian Product  $\times$ : for any family  $T$  of sets,

$$f \in \times T \equiv \mathcal{D}f = \mathcal{D}T \wedge \forall x: \mathcal{D}f \cap \mathcal{D}T. f x \in T x. \quad (26)$$

Immediate properties of (26):

- Function equality  $f = g \equiv \mathcal{D}f = \mathcal{D}g \wedge \forall x: \mathcal{D}f \cap \mathcal{D}g. f x = g x$  yields  $f = g \equiv f \in \times(\iota \circ g)$  (exact approximation).
- (Semi-)pointfree form:  $\times T = \{f: \mathcal{D}T \rightarrow \cup T \mid \forall(f \hat{\in} T)\}$

– Commonly used conventions as particularizations

- \* The usual Cartesian product for a pair of sets  $A, B$  is defined by

$$(a, b) \in A \times B \equiv a \in A \wedge b \in B$$

Letting  $T := A, B$  in (26), calculation yields

$$\times(A, B) = A \times B$$

If  $A \neq \emptyset$  and  $B \neq \emptyset$ , then  $\times^-(A \times B) 0 = A$  and  $\times^-(A \times B) 1 = B$ .

- \* Dependent types: letting  $T := a : A . B_a$ ,

$$\times(a : A . B_a) = \{f : A \rightarrow \cup(a : A . B_a) \mid \forall a : A . f a \in B_a\}$$

Convenient shorthand:  $A \ni a \rightarrow B_a$  for  $\times a : A . B_a$  **Examples:**

- Clearer chained dependencies, e.g.,  $A \ni a \rightarrow B_a \ni b \rightarrow C_{a,b}$ .
- Types for generic operators. Let  $\mathcal{F}$  and  $\mathcal{T}$  be the function and type universes and  $\text{fam}$  defined by  $f \in \text{fam } Y \equiv \mathcal{R} f \subseteq Y$ ,

$$\begin{aligned} (\circ) &\in \mathcal{F}^2 \ni (f, g) \rightarrow \{x : \mathcal{D} g \mid g x \in \mathcal{D} f\} \rightarrow \mathcal{R} f \\ (T) &\in \text{fam } \mathcal{F} \ni f \rightarrow \cap(\mathcal{D} \circ f) \rightarrow \mathcal{D} f \ni x \rightarrow \mathcal{R}(f x). \end{aligned}$$

## 3 Applications in programming

### 3.1 Functional programming

Principle: **defining objects as functions makes them inherit all generic operators**.  
Moreover, by doing so, **nothing is lost** (expression styles, calculation rules).

- **Application to sequences** (as functions) with basic *prefix* operator  $\succ$

$$a \succ x = i : \square (\# x + 1) . (i = 0) ? a \dagger x (i - 1) \quad (27)$$

- Supports existing reasonings, e.g., induction: for any predicate  $P : A^* \rightarrow \mathbb{B}$ ,

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

Similarly for infinite structures and coinduction.

- Interesting applications of composition and transposition Examples:

- \*  $(0, 3, 5, 7) \circ (2, 3, 1) = 5, 7, 3$  and  $(0, 3, 5, 7) \circ (2, 3, 5) = 5, 7$ .
- \*  $f \circ (x, y) = f x, f y$  subsumes  $\text{map } f \ [x, y] = [f x, f y]$ .
- \*  $((a, b, c), (a', b', c'))^T = ((a, a'), (b, b'), (c, c'))$  subsumes  
 $\text{zip } [[a, b, c], [a', b', c']] = [[a, a'], [b, b'], [c, c']]$

- **Transposition in functional languages**

Definition usually possible for the (very) special case  $A \rightarrow (B \rightarrow C)$ .

Image  $f^T$  of  $f: A \rightarrow (B \rightarrow C)$  has type  $B \rightarrow (A \rightarrow C)$  and property  $(f^T)^T = f$ .

- **Application to sequences** (continuation)

- Sequences have inverses Example:  $(3, 3, 7)^{-} 7 = 2$

- Some operators over sequences have inverses e.g.,  $\succ^{-} (a \succ x) = a, x$   
Application example:  $head\ x = \succ^{-} x\ 0$  for any nonempty sequence  $x$ .

- Pattern matching as equational definition using function inverses

Recursive definitions like  $f (a \succ x) = h (a, f\ x)$ : just covered.

Generalization: definition format  $f (g (a, x)) = e (a, x)$ .

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  $g$ .

- **Direct extension:** many application opportunities.

Example (sequences): direct extension as a pairwise map:

`direx (*) [[a,b,c],[a',b',c']] = [a * a', b * b', c * c']` or:

`direx (*) = (map (*)) . zip`. More generally:  $(\hat{\star}) = \overline{(\star)} \circ (\&)$ .

## 3.2 Aggregate data types and structures

- **Pascal-like records** (ubiquitous in programs) **How making them functional?**
  - **Well-known approach:** selector functions matching the field labels. (e.g., Haskell) Problem: records themselves as arguments, not functions.
  - **Preferred alternative:** generalized functional cartesian product  $\times$ : records as *functions*, domain: set of field labels from an *enumeration type*. **E.g.**,

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

Then  $person : Person$  satisfies  $person\ name \in \mathbb{A}^*$  and  $person\ age \in \mathbb{N}$ .

- **Syntactic sugar:**  $\text{Record} : \text{fam } (\text{fam } \mathcal{T}) \rightarrow \mathcal{P } \mathcal{F}$  with  $\text{Record } F = \times (\cup F)$   
Now we can write  $Person := \text{Record } (name \mapsto \mathbb{A}^*, age \mapsto \mathbb{N})$ .

- **More about the funcart operator  $\times$**

- “Workhorse” for typing all structures unified by functional mathematics.

- \* Recall  $A \rightarrow B = \times(A \bullet B)$  and  $A \times B = \times(A, B)$ .

- \* For set  $A$  and  $n: \mathbb{N} \cup \iota\infty$ , define  $A \uparrow n = \square n \rightarrow A$  (shorthand:  $A^n$ )

- So  $A \uparrow n = \times(\square n \bullet A)$ , the  $n$ -fold Cartesian product.

- \*  $A^* = \cup n: \mathbb{N}. A^n$  completes the functional unification of aggregates

- **Is a genuine functional, not an ad hoc abstractor**

Yields many useful algebraic properties. Most noteworthy: **inverse**.

- \* Choice axiom  $\times T \neq \emptyset \equiv \forall x: \mathcal{D}T. Tx \neq \emptyset$  characterizes  $\text{Bdom } \times$

- \* If  $\times T \neq \emptyset$ , then  $\times^- (\times T) = T$

- \* **Example:** if  $A \neq \emptyset$  and  $B \neq \emptyset$ , then  $\times^- (A \times B) = A, B$ , hence  $\times^- (A \times B) 0 = A$  and  $\times^- (A \times B) 1 = B$ .

- \* **Explicit image definition:** for any nonempty  $S$  in the range of  $\times$ ,

$$\times^- S = x: \text{Dom } S. \{f x \mid f: S\} \quad (28)$$

where  $\text{Dom } S$  is the common domain of the functions in  $S$  extracted, e.g., by  $\text{Dom } S = \cap f: S. \mathcal{D}f = \cap (\mathcal{D} \upharpoonright S)$ .

- **Other structures** are also defined as functions.
  - **Example: trees** are functions whose domains are *branching structures*, i.e., sets of sequences describing the path from the root to a leaf.
    - \* Covers any branch labeling, e.g., for binary tree: subset of  $\mathbb{B}^*$ .
    - \* Classes of trees: specified by restrictions on the branching structures.
    - \* The  $\times$  operator can even specify types for leaves individually.
  - Aggregates (as functions) inherit elastic operators for suitable arguments.
    - Example:**  $\sum s$  sums the “elements” of any number-valued structure  $s$ .
  - Generic functionals are inherited whatever the image type.

- **Important for structures: direct extension**
  - Dijkstra considers all operators implicitly extended to “structures”
    - \* Without elaborating the domain (is fixed: the program state space).
    - \* Even for equality, which becomes a pointwise instead of an “overall” characteristic.
  - LabVIEW building blocks are similarly extended (called *polymorphism*).
  - Our view
    - \* Implicit extensions are convenient in a particular area of discourse
    - \* Broader application range needs finer tuning offered by an explicit generic operator ( $\bar{\phantom{x}}$ ,  $\hat{\phantom{x}}$  or  $\langle \phantom{x} \rangle$ ).

### 3.3 Overloading and polymorphism

- **Terminology**

- **Overloading**: using identifier for designating “different” objects.  
**Polymorphism**: different argument types, formally same image definition.
- In Haskell: called *ad hoc* and *ad hoc* polymorphism respectively.
- Considering general overloading also suffices for covering polymorphism.

- **Two main issues in overloading an operator:**

- **Disambiguation** of application to all possible arguments
- **Refined typing**: reflecting relation between argument and result type

Covered respectively by:

- Ensuring that different functions denoted by the operator are *compatible* in the sense of the generic ©-operator.
- A suitable type operator whose design is discussed next.

- **Option: overloading by explicit parametrization** Trivial with  $\times$ .  
Example: *binary addition* function adding two binary words of equal length.

**def** *binadd*<sub>-</sub> :  $\times n : \mathbb{N} . (\mathbb{B}^n)^2 \rightarrow \mathbb{B}^{n+1}$  **with** *binadd*<sub>n</sub> (*x*, *y*) = ...

Only the type is relevant. Note: *binadd*<sub>n</sub>  $\in (\mathbb{B}^n)^2 \rightarrow \mathbb{B}^{n+1}$  for any  $n : \mathbb{N}$ .

- **Option: overloading without auxiliary parameter**

– **Requirement:** operator  $\otimes$  with properties exemplified for *binadd* by

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

\* Note that  $n : \mathbb{N} . (\mathbb{B}^n)^2 \rightarrow \mathbb{B}^{n+1}$  is a family of function types.

\* Domain of *binadd* is  $\cup n : \mathbb{N} . (\mathbb{B}^n)^2$ , and the type

$$\cup (n : \mathbb{N} . (\mathbb{B}^n)^2) \ni (x, y) \rightarrow \mathbb{B}^{\#x+1}$$

\* **This information must be obtainable from**  $\otimes n : \mathbb{N} . (\mathbb{B}^n)^2 \rightarrow \mathbb{B}^{n+1}$ .

– An interesting design

- \* Consider *binadd* as a *merge* of functions of type  $(\mathbb{B}^n)^2 \rightarrow \mathbb{B}^{n+1}$  (one for each  $n$ ).

The family of functions merged is taken from  $\times n : \mathbb{N}. (\mathbb{B}^n)^2 \rightarrow \mathbb{B}^{n+1}$

Requirement: *compatibility*

(trivially satisfied in this example because of nonintersecting domains).

- \* Generalization to arbitrary function type family: yields the desired *function type merge* ( $\otimes$ ) operator

$$\mathbf{def} \otimes : \mathbf{fam} (\mathcal{P} \mathcal{F}) \rightarrow \mathcal{P} \mathcal{F} \mathbf{with} \otimes F = \{ \cup f \mid f : (\times F) \textcircled{c} \} \quad (29)$$

Note: (29) is not the original form [Van den Beuken], the latter used a non-generic merge to enforce compatibility implicitly.

- \* Applications for other purposes than polymorphism shown later.

- **Use in a (declarative) language context** Incremental definition:
  - **Regular definitions:** `def  $x : X$  with  $p_x$` , allowing only one definition for  $x$ .
  - **Overloaded operator definitions:** `defo` drops this condition but **requires** that, for any collection of definitions of the form

$$\mathbf{defo} \ f : F_i \ \mathbf{with} \ P_i \ f$$

(for  $i$  in some index set  $I$  at the meta-level, but the same  $f$ ),  
the derived collection of definitions

$$\mathbf{def} \ g_i : F_i \ \mathbf{with} \ P_i \ g_i$$

is *compatible*, i.e.,  $\textcircled{C} \ i : I . g_i$

(often satisfied trivially by nonintersecting domains).

Then the given collection defines an operator

$$f : \bigotimes_{i : I} F_i \ \mathbf{with} \ f = \bigcup_{i : I} g_i$$

- **A rough analogy with Haskell**

Illustrated by an **example** from *A Gentle Introduction to Haskell 98*:

```
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) rendered by

```
def Eq :  $\mathcal{T} \rightarrow \mathcal{P} \mathcal{F}$  with Eq X =  $X^2 \rightarrow \mathbb{B}$ 
defo —==— : Eq Integer with x == y  $\equiv$  x integerEq y
defo —==— : Eq Float with x == y  $\equiv$  x floatEq y.
```

The type of == is  $Eq\ Integer \otimes Eq\ Float$  and, observing that nonintersecting domains ensure compatibility,  $(==) = (integerEq) \cup (floatEq)$ .

An essential difference is that Haskell attaches operators to a class.

### 3.4 Functional predicate calculus

- **Motivation for choice of topic**

- Most basic mathematical tool in software engineering [Parnas].
- Practical use requires a complete “toolkit” of calculation rules [Gries]
- Full toolkit given in our course notes (website), here only structure:

- **Style of formulation**

- Point-free formulation style enabled by the use of generic functionals
- Traditional-looking forms by writing predicates as function abstractions

- **Axioms** The *quantifiers*  $\forall$  and  $\exists$  are predicates over predicates (say,  $P$ ):

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

Conceptually: the simplest possible definitions.

Calculationally: rich algebra of calculation rules,

all derived from **function equality**: Leibniz and Extensionality).

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

	Point-free	Point-wise
Duality	$\forall(\neg P) \equiv (\neg \exists) P$	$\forall(x: X . \neg p_x) \equiv \neg(\exists x: X . p_x)$
Meeting	$\forall P \wedge \forall Q \Rightarrow \forall(P \hat{\wedge} Q)$	$\forall(x: X . p_x) \wedge \forall(x: Y . q_x)$ $\Rightarrow \forall(x: X \cap Y . p_x \wedge q_x)$
Constant	$\forall(X \bullet p) \equiv X = \emptyset \vee p$	$\forall(x: X . p) \equiv X = \emptyset \vee p$

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

Obvious particular cases:  $\forall \varepsilon \equiv 1$  and  $\exists \varepsilon \equiv 0$

$\forall(X \bullet 1) \equiv 1$  and  $\forall(X \bullet 0) \equiv X = \emptyset$

$\exists(X \bullet 0) \equiv 0$  and  $\exists(X \bullet 1) \equiv X \neq \emptyset$ .

- **Important issue: typing and possibly empty domains**

- Neglected in most logic textbooks and (hence) cause of errors by students
- Especially important for the finite structures in software and algorithmics

- **Semidistributivity rules** (examples)

$$\begin{array}{ll}
\forall (p \bar{\wedge} P) \equiv (p \wedge \forall P) \vee \mathcal{D}P = \emptyset & \forall (x: X . p \wedge q_x) \equiv (p \wedge \forall x: X . q_x) \vee X = \emptyset \\
\forall (p \bar{\Rightarrow} P) \equiv p \Rightarrow \forall P & \forall (x: X . p \Rightarrow q_x) \equiv p \Rightarrow \forall (x: X . q_x) \\
\forall (P \bar{\Leftarrow} p) \equiv \exists P \Rightarrow p & \forall (x: X . p_x \Rightarrow q) \equiv \exists (x: X . p_x) \Rightarrow q
\end{array}$$

By duality:

$$\begin{array}{l}
\exists (p \bar{\wedge} P) \equiv p \wedge \exists P \\
\exists (p \bar{\Rightarrow} P) \equiv (p \Rightarrow \exists P) \wedge \mathcal{D}P \neq \emptyset \\
\exists (P \bar{\Leftarrow} p) \equiv (\forall P \Rightarrow p) \wedge \mathcal{D}P \neq \emptyset.
\end{array}$$

Note: also covered by a generalized variant of Shannon's theorem.

- **Instantiation and generalization**

- Counterparts appear in logical textbooks as the axiom and inference rule for quantification
- here again are direct **consequences** of the equational axioms for quantifiers ( $\forall P \equiv P = \mathcal{D} P \bullet 1$  and  $\exists P \equiv P \neq \mathcal{D} P \bullet 0$ ) and function equality.

$$\begin{array}{ll} \forall P \Rightarrow x \in \mathcal{D} P \Rightarrow P x & \forall (x : X . p_x) \Rightarrow x \in X \Rightarrow p_x \\ q \Rightarrow x \in \mathcal{D} P \Rightarrow P x \vdash q \Rightarrow \forall P & q \Rightarrow x \in X \Rightarrow p_x \vdash q \Rightarrow \forall (x : X . p_x) \end{array}$$

Basis for rules seen in logic textbooks and others rules important for practice

- **Final examples**

- **Trading:**

$$\begin{array}{ll} \forall (P \downarrow Q) \equiv \forall (Q \Leftrightarrow P) & \forall (x : X \wedge q_x . p_x) \equiv \forall (x : X . q_x \Rightarrow p_x) \\ \exists (P \downarrow Q) \equiv \exists (Q \hat{\wedge} P) & \exists (x : X \wedge q_x . p_x) \equiv \exists (x : X . q_x \wedge p_x) \end{array}$$

- **Composition rule:** if  $\mathcal{D} P \subseteq \mathcal{R} f$  then  $\forall P \equiv \forall (P \circ f)$ ; pointwise variant:  
*dummy change:*  $\forall (x : \mathcal{D} P . P x) \equiv \forall (y : \mathcal{D} f . f y \in \mathcal{D} P \Rightarrow P (f y))$ .

### 3.5 Formal semantics (from conventional languages to LabVIEW)

- **Expressing abstract syntax** (Unification of Meyer’s ad hoc conventions)
  - For *aggregate constructs and list productions*: functional Record and  $*$ .  
This is  $\times$  actually:  $\text{Record } F = \times (\cup F)$  and  $A^* = \cup n : \mathbb{N} . \times (\square n \bullet A)$ .
  - For *choice productions needing disjoint union*: generic elastic  $\mid$ -operator  
For any family  $F$  of types,

$$\mid F = \cup x : \mathcal{D} F . \{x \mapsto y \mid y : F x\} \tag{31}$$

Idea: analogy with  $\cup F = \cup (x : \mathcal{D} F . F x) = \cup x : \mathcal{D} F . \{y \mid y : F x\}$ .

#### Remarks

- \* Variadic shorthand:  $A \mid B = \mid(A, B) = \{0 \mapsto a \mid a : A\} \cup \{1 \mapsto b \mid b : B\}$
- \* Using  $i \mapsto y$  rather than the common  $i, y$  yields more uniformity.  
Same three type operators can describe directory and file structures.
- \* For program semantics, disjoint union is often “overengineering”.

– Typical examples: (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.

A few items are left undefined here (easily inferred).

If disjoint union wanted: *Skip* | *Assignment* | *Compound* | etc.

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

**def** *p*: *Program* **with** *p* = *declarations*  $\mapsto$  *dl*  $\cup$  *body*  $\mapsto$  *instr*

- **Semantics** “Functionalizing” Meyer’s formulation using generic functionals.

**Example:** for static semantics, *validity of declaration lists* (no double declarations) and the *variable inventory* are expressed by

$$\begin{aligned} \mathbf{def} \ Vdcl : Dlist \rightarrow \mathbb{B} \ \mathbf{with} \ Vdcl \ dl = \mathbf{inj} \ (dl^T v) \\ \mathbf{def} \ Var : Dlist \rightarrow \mathcal{P} \ Variable \ \mathbf{with} \ Var \ dl = \mathcal{R} \ (dl^T v) \end{aligned}$$

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

$$\begin{aligned} \mathbf{def} \ typmap : Dlist_{Vdcl} \ni dl \rightarrow Var \ dl \rightarrow Tval \ \mathbf{with} \\ typmap \ dl = tval \circ (dl^T t) \circ (dl^T v)^{-} \end{aligned}$$

Equivalently,  $typmap \ dl = \cup d : \mathcal{R} \ dl . d \ v \mapsto tval \ (d \ t)$ .

A type map can be used as a context parameter for expressing validity of expressions and instructions, shown next.

- **Semantics** (continuation) How function *merge* ( $\sqcup$ ) obviates case expressions

**Example:** type (*Tex*) and type correctness (*Vexp*) of expressions. Assume

**def** *Expr* := *Constant*  $\sqcup$  *Variable*  $\sqcup$  *Applic*

**def** *Constant* := *IntCons*  $\sqcup$  *BoolCons*

**def** *Applic* := **Record** (*op*  $\mapsto$  *Oper*, *term*  $\mapsto$  *Expr*, *term'*  $\mapsto$  *Expr*)

Letting *Tmap* :=  $\sqcup dl : Dlist_{V_{ddl}} . typmap dl$  and *Tval* := {*it*, *bt*, *ut*}, define

**def** *Tex* : *Tmap*  $\rightarrow$  *Expr*  $\rightarrow$  *Tval* **with**

*Tex* *tm* = (*c* : *IntCons* . *it*)  $\sqcup$  (*c* : *BoolCons* . *bt*)

$\sqcup$  (*v* : *Variable* . *ut*)  $\otimes$  *tm*

$\sqcup$  (*a* : *Applic* . (*a op*  $\in$  *Arith\_op*) ? *it*  $\dagger$  *bt*)

**def** *Vexp* : *Tmap*  $\rightarrow$  *Expr*  $\rightarrow$   $\mathbb{B}$  **with**

*Vexp* *tm* = (*c* : *Constant* . 1)  $\sqcup$  (*v* : *Variable* . *v*  $\in$   $\mathcal{D}$  *tm*)

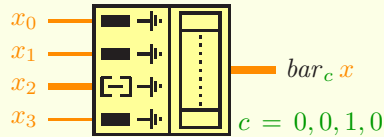
$\sqcup$  (*a* : *Applic* . *Vexp* *tm* (*a term*)  $\wedge$  *Vexp* *tm* (*a term'*)  $\wedge$

*Tex* *tm* (*a term*) = *Tex* *tm* (*a term'*)

= (*a op*  $\in$  *Bool\_op*) ? *bt*  $\dagger$  *it*)

- **Semantics of data flow languages**

- **Already shown:** Generic operators for calculations interconnects.
- **Semantics:** some time ago for Silage (textual), now LabVIEW (graphical)  
**Example:** LabVIEW block *Build Array* It is **generic**: configuration parametrizable by menu selection (number and kind of input: *element* or *array*).



We formalize the configuration by a list in  $\mathbb{B}^+$  (0 = *element*, 1 = *array*)

**Semantics example:**  $bar_{0,0,1,0}(a, b, (c, d), e) = a, b, c, d, e$

Type expression:  $\mathbb{B}^+ \ni c \rightarrow \times (i : \mathcal{D} c . (V, V^*) (c i)) \rightarrow V^*$  for base type  $V$

Image definition:  $bar_c x = ++ i : \mathcal{D} c . (\tau(x i), x i) (c i)$ . Point-free result:

$$\mathbf{def} \ bar\_ : \mathbb{B}^+ \ni c \rightarrow \otimes V : \mathcal{T} . \times ((V, V^*) \circ c) \rightarrow V^* \mathbf{with}$$

$$bar_c = ++ \circ \| ((\tau, id) \circ c).$$

### 3.6 Relational databases in functional style

- **Database system** = storing information + convenient user interface  
Presentation: offering precisely the information wanted as “virtual tables”.

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
...	...	...	

- **Relational database** presents the tables as relations.
  - **Traditional view**: rows as tuples (and tuples not seen as functions).  
Problem: access only by separate indexing function using numbers.  
Patch: “grafting” *attribute names* for column headings.  
Disadvantages: model not purely relational, operators on tables ad hoc.
  - **Functional view**; the table rows as *records* using  $\text{Record } F = \times (\cup F)$   
Advantage: embedding in general framework, inheriting algebraic properties and generic operators.

- **Relational databases as sets of functions using**  $\text{Record } F = \times (\cup F)$

**Example:** the table representing *General Course Information*

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
...	...	...	

can be declared as  $GCI : \mathcal{P} \text{ CID}$ , a set of *Course Information Descriptors* with  
 $\text{def } CID := \text{Record} (\text{code} \mapsto \text{Code}, \text{name} \mapsto \mathbb{A}^*, \text{inst} \mapsto \text{Staff}, \text{prrq} \mapsto \text{Code}^*).$

- **Access to a database:** done 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  $GCI$

The second requires *joining* table  $GCI$  with a time table.

All require *selecting* relevant rows.

- **Formalizing queries**

Basic elements of any *query language* for handling virtual tables:

*selection*, *projection* and *natural join* [Gries].

Our generic functionals directly provide this functionality. Convention: let  $\mathcal{P} R$  be the type for an arbitrary table (set) of records (functions) of type  $R$ .

- **Selection** ( $\sigma$ ) selects in any table  $S : \mathcal{P} R$  those records satisfying  $P : R \rightarrow \mathbb{B}$ .  
Solution: set filtering  $\sigma(S, P) = S \downarrow P$ .  
**Example:**  $GCI \downarrow (r : CID . r \text{ code} = \text{CS300})$  selects the row pertaining to question (a), “Who is the instructor for CS300?”.
- **Projection** ( $\pi$ ) yields in any  $S : \mathcal{P} R$  columns with field names in a set  $F$ .  
Solution: function domain restriction  $\pi(S, F) = \{r \upharpoonright F \mid r : S\}$ .  
**Example:**  $\pi(GCI, \{\text{code}, \text{inst}\})$  selects the columns for question (a) and  $\pi(GCI \downarrow (r : CID . r \text{ code} = \text{CS300}), \iota \text{ inst})$  reflects the entire question.
- **Join** ( $\bowtie$ ) combines tables  $S$  and  $T$  by uniting the field name sets, rejecting records whose contents for common field names disagree. (next slide)

- **Join** ( $\bowtie$ ) combines tables  $S$  and  $T$  by uniting the field name sets, rejecting records whose contents for common field names disagree.

Solution:  $S \bowtie T = \{s \cup t \mid (s, t) : S \times T \wedge s \odot t\}$  (**function type merge!**)

**Example:**  $GCI \bowtie CS$  combines table  $GCI$  with the *course schedule* table  $CS$  (e.g., the table below) in the desired manner for answering questions

- (b) “At what time is K. Jason normally teaching a course?”
- (c) “Which courses is R. Barns teaching in the Spring Quarter?”

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
...	...	...	...	...

- **Algebraic remarks about function type merge** Note that  $S \bowtie T = S \otimes T$ . Compatibility property:  $f \odot g \wedge (f \cup g) \odot h \equiv f \odot g \wedge f \odot h \wedge g \odot h$  and similarly (by symmetry),  $(g \odot h) \wedge f \odot (g \cup h) \equiv g \odot h \wedge f \odot g \wedge f \odot h$ . This can be used to show  $\odot(f, g, h) \Rightarrow (f \cup g) \cup h = f \cup (g \cup h)$ . Hence, although  $\cup$  is *not* associative,  $\otimes$  (and hence  $\bowtie$ ) is associative.

### 3.7 Final considerations

- **Sobering thought:** even in “functional” programming languages, the most-often used data structures (lists, tuples, etc.) are *not* functions!  
Usual reason: restrictions imposed by the implementation

- **Observations:**

- Implementation principles and techniques evolve in time
- Functional view is “downward compatible” with existing definitions
- Hence this view can still be exploited in *reasoning* about programs!

- **Example** (suggested by Steven Johnson). An interleaving variant of zip is

$$\text{zap } [a : x] [b : y] = a : b : \text{zap } x y, \quad (32)$$

(intended for infinite sequences). Task: prove that

$$\text{map } f (\text{zap } x y) = \text{zap } (\text{map } f x) (\text{map } f y). \quad (33)$$

This is usually done by co-induction. Alternative: proving it as an instance of a more general property of *generic functionals*.

- **Elaboration**

Recall (32), i.e.,  $zap (a \succ x) (b \succ y) = a \succ b \succ zap x y$

To prove: (33), i.e.,  $map f (zap x y) = zap (map f x) (map f y)$

Note: infinite sequences “are” functions with domain  $\mathbb{N}$

- “Quickie” proof for (33) By induction on  $\mathbb{N}$ , (32) becomes

$$zap x y (2 \cdot n) = x n \quad \text{and} \quad zap x y (2 \cdot n + 1) = y n$$

So (33) follows by the laws for  $\circ$  since  $map f x = f \circ x = \overline{f} x$ .

Yet: uses a domain variable and case distinction (even, odd argument).

- **Proof from general properties** Via  $zap x y n = (x \frac{n}{2}, y \frac{n-1}{2}) (n \dagger 2)$ ,

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

where  $\alpha : \{2 \cdot n \mid n : \mathbb{N}\} \rightarrow \mathbb{N}$  with  $\alpha n = n/2$  and  $\beta : \{2 \cdot n + 1 \mid n : \mathbb{N}\} \rightarrow \mathbb{N}$  with  $\beta n = (n - 1)/2$ , whereas  $\gamma : \mathbb{N} \rightarrow \mathbb{B}$  with  $\gamma n = n \dagger 2$ .

On the next slide, we prove (33) from (34) using the following property of the generic functionals: for any function  $f$  and function family  $F$

$$\overline{\overline{f}} (\parallel F^U) = \parallel (\overline{\overline{f}} F)^U. \tag{35}$$

- **Details of the proof:** given (34), i.e.  $zap\ x\ y = \|(x \circ \alpha, y \circ \beta)^U \gamma$ , we prove (33), i.e.,  $map\ f\ (zap\ x\ y) = zap\ (map\ f\ x)\ (map\ f\ y)$ , using (35), i.e.,  $\overline{f}\ (\| F^U) = \| (\overline{f}\ F)^U$  as follows

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

**Remark:** more important than the proof of theorem (33) itself is establishing theorem (35) as a functional generalization thereof.

Also, the fact that defining sequences as functions allows handling “sequences with holes” (such as  $x \circ \alpha$  and  $y \circ \beta$ ) effortlessly may also prove useful.

## 4 Conclusion

- Small collection of functionals directly useful in a wide range of applications, from continuous mathematics to programming
- Generic nature and (hence) wide coverage depend on two elements:
  - Unifying view on objects by (re)defining them as functions
  - Judicious specification of the domains for the 'results' of by the functionals.

Additional benefits: point-free style, useful algebraic rules for calculation

- **Examples** shown for 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 shown here (yet interesting): examples in non-discrete mathematics.
- **Valuable side-effect for organizing human knowledge:** exploiting similarities between disparate fields, reducing conceptual and notational overhead, and making the transition easier by capturing analogies formally.