

Making CS and classical EE meet: unification by formalization

Raymond Boute — INTEC, Ghent University

EE380 — Stanford, 2004/10/13

Overview

0. Motivation: problem, main cause, solution ("formalization to the rescue")
1. Design of a unifying formalism (= language + formal rules)
2. Illustration I — Origin of the basic ideas: signals and systems
3. Illustration II — A typical generic functional: design and unifying power
4. Illustration III — Typical predicative calculations: exploring program dynamics
5. Conclusions — Unifying Electrical and Computer engineering

Next topic

0. Motivation: problem, main cause, solution ("formalization to the rescue")

- Problem: a traditional rift between classical engineering and CS
- Main cause: style breach between well and poorly formalized mathematics
- Solution: proper formalization **UT FACIANT OPUS SIGNA**
"Letting the symbols do the work": formalization as a boon, not a burden

1. Design of a unifying formalism (= language + formal rules)
2. Illustration I — Origin of the basic ideas: signals and systems
3. Illustration II — A typical generic functional: design and unifying power
4. Illustration III — Typical predicative calculations: exploring program dynamics
5. Conclusions — Unifying Electrical and Computer engineering

0 Motivation: problem, main cause, solution

0.0 Problem: a traditional rift between classical engineering and CS

Professional engineers can often be distinguished from other designers by the engineers' ability to use mathematical models to describe and analyze their products.

(David L. Parnas, "Predicate Logic for Software Engineering")

- Observation: difference in practice
 - In classical engineering (electrical, mechanical, civil): established *de facto*
 - In software “engineering”: mathematical models rarely used (occasionally in critical systems under the name “Formal Methods”)
C. Michael Holloway: “software designers aspire to be(come) engineers”
- Difference reflected in design methods and support tools
 - Electronics engineers readily use, e.g., Matlab, Simulink (*textbook math*)
 - Software designers use acronym-ridden “soft” tools (with mathphobic notation), rarely provers or model checkers (problem: no common math)

0.1 Cause: style breach between well and poorly formalized mathematics

Consider the degree of formality in “everyday mathematics” calculations

- Well-developed in long-standing areas of mathematics (algebra, analysis, etc.)

From: R. Bracewell / transforms

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

From: R. Blahut / data compacting

$$\begin{aligned} &\frac{1}{n} \sum_{\mathbf{x}} p^n(\mathbf{x}|\theta) l_n(\mathbf{x}) \\ &\leq \frac{1}{n} \sum_{\mathbf{x}} p^n(\mathbf{x}|\theta) [1 - \log q^n(\mathbf{x})] \\ &= \frac{1}{n} + \frac{1}{n} L(\mathbf{p}^n; \mathbf{q}^n) + H_n(\theta) \\ &= \frac{1}{n} + \frac{1}{n} d(\mathbf{p}^n, \mathcal{G}) + H_n(\theta) \\ &\leq \frac{2}{n} + H_n(\theta) \end{aligned}$$

- Poorly developed in logical parts. This causes a serious style breach.

“The notation of elementary school arithmetic, which nowadays everyone takes for granted, took centuries to develop. There was an intermediate stage called *syncopation*, using abbreviations for the words for addition, square, root, *etc.* For example Rafael Bombelli (*c.* 1560) would write

R. c. L. 2 p. di m. 11 L for our $3\sqrt{2 + 11i}$.

Many professional mathematicians to this day use the quantifiers ($\forall, \exists$) in a similar fashion,

$\exists \delta > 0$ s.t. $|f(x) - f(x_0)| < \epsilon$ if $|x - x_0| < \delta$, for all $\epsilon > 0$,

in spite of the efforts of [Frege, Peano, Russell] [...]. Even now, mathematics students are expected to learn complicated $(\epsilon\text{-}\delta)$ -proofs in analysis with no help in understanding the logical structure of the arguments. Examiners fully deserve the garbage that they get in return.”

(P. Taylor, “Practical Foundations of Mathematics”)

- Increasingly worse as we get closer to the necessities in Computing Science (calculating with logic expressions, set expressions etc.) (Examples to follow)

Calculational style to the rescue (illustration; calculation rules introduced later)

Proposition 2.1. for any function $f : \mathbb{R} \rightarrow \mathbb{R}$, any subset S of $\mathcal{D} f$ and any a adherent to S ,

- (i) $\exists (L : \mathbb{R} . L \text{ islim}_f a) \Rightarrow \exists (L : \mathbb{R} . L \text{ islim}_{f \upharpoonright S} a)$,
- (ii) $\forall L : \mathbb{R} . \forall M : \mathbb{R} . L \text{ islim}_f a \wedge M \text{ islim}_{f \upharpoonright S} a \Rightarrow L = M$.

Proof for (ii): Letting $b R \delta$ abbreviate $\forall x : S . |x - a| < \delta \Rightarrow |f x - b| < \epsilon$,

$$\begin{aligned}
 & L \text{ islim}_f a \wedge M \text{ islim}_{f \upharpoonright S} a \\
 \Rightarrow & \langle \text{Hint in prf. (i)} \rangle L \text{ islim}_{f \upharpoonright S} a \wedge M \text{ islim}_{f \upharpoonright S} a \\
 \equiv & \langle \text{Def. islim, hyp.} \rangle \forall (\epsilon : \mathbb{R}_{>0} . \exists \delta : \mathbb{R}_{>0} . L R \delta) \wedge \forall (\epsilon : \mathbb{R}_{>0} . \exists \delta : \mathbb{R}_{>0} . M R \delta) \\
 \equiv & \langle \text{Distribut. } \forall / \wedge \rangle \forall \epsilon : \mathbb{R}_{>0} . \exists (\delta : \mathbb{R}_{>0} . L R \delta) \wedge \exists (\delta : \mathbb{R}_{>0} . M R \delta) \\
 \equiv & \langle \text{Distribut. } \wedge / \exists \rangle \forall \epsilon : \mathbb{R}_{>0} . \exists \delta : \mathbb{R}_{>0} . \exists \delta' : \mathbb{R}_{>0} . L R \delta \wedge M R \delta' \\
 \Rightarrow & \langle \text{Closeness lem.} \rangle \forall \epsilon : \mathbb{R}_{>0} . \exists \delta : \mathbb{R}_{>0} . \exists \delta' : \mathbb{R}_{>0} . a \in \text{Ad } S \Rightarrow |L - M| < 2 \cdot \epsilon \\
 \equiv & \langle \text{Hyp. } a \in \text{Ad } S \rangle \forall \epsilon : \mathbb{R}_{>0} . \exists \delta : \mathbb{R}_{>0} . \exists \delta' : \mathbb{R}_{>0} . |L - M| < 2 \cdot \epsilon \\
 \equiv & \langle \text{Const. pred. } \exists \rangle \forall \epsilon : \mathbb{R}_{>0} . |L - M| < 2 \cdot \epsilon \\
 \equiv & \langle \text{Vanishing lem.} \rangle L - M = 0 \\
 \equiv & \langle \text{Leibniz, inv. } + \rangle L = M
 \end{aligned}$$

0.2 Solution: proper formalization

- Formal approach: not just “using math”, but doing it *formally*
 - “formal” = manipulating expressions on the basis of their *form*
 - “informal” = manipulating expressions on the basis of their *meaning*
- Dispelling poor reputation of formal mathematics
 - Idea “difficult, tedious” deserved only where badly done (traditional logic)
 - Formality tacitly much appreciated where successful (algebra, calculus)
 - Practical application in critical systems (well-known issue)
 - Even more important: **UT FACIANT OPUS SIGNA**
(Maxim of the conferences on *Mathematics of Program Construction*)

Provides help in *thinking*: deriving guidance from the *shape* of formulas
→ additional kind of / added dimension to intuition, tool for discovery!

- “All that remains” is showing how it is done
(making things simple required considerable thinking and effort)

Next topic

0. Motivation: problem, main cause, solution ("formalization to the rescue")

1. Design of a unifying formalism (= language + formal rules)

- Language rationale: the need for defect-free notation
- Language design: the four constructs of Functional Mathematics
- Rules rationale: calculational reasoning, also in logic
- Rule design: functions, generic functionals, predicates and quantifiers

2. Illustration I — Origin of the basic ideas: signals and systems

3. Illustration II — A typical generic functional: design and unifying power

4. Illustration III — Typical predicative calculations: exploring program dynamics

5. Conclusions — Unifying Electrical and Computer engineering

1 Design of a unifying formalism (= language + rules)

1.0 Language rationale: the need for defect-free notation

Why not always use “standard” mathematical conventions? Reason: defects!

Examples A: defects in often-used conventions in common mathematics

- Ellipsis, i.e., “omission dots” (...) as in $a_0 + a_1 + \cdots + a_n$

Common use violates Leibniz’s principle (substitution of equals for equals)

Example: $a_i = i^2$ and $n = 7$ yields $0 + 1 + \cdots + 49$ (probably not intended!)

- Summation sign $\sum$ not as well-understood as often assumed.

Example: error in *Mathematica*: $\sum_{i=1}^n \sum_{j=i}^m 1 = \frac{n \cdot (2 \cdot m - n + 1)}{2}$

Taking $n := 3$ and $m := 1$ yields 0 instead of the correct sum 1.

- Confusing function application with the function itself

Example: $y(t) = x(t) * h(t)$ where $*$ is convolution.

Causes incorrect instantiation, e.g., $y(t - \tau) = x(t - \tau) * h(t - \tau)$

Examples B: ambiguities in conventions for sets

- Patterns typical in mathematical writing:
(assuming logical expression p , arbitrary expression p)

Patterns	$\{x \in X \mid p\}$	and	$\{e \mid x \in X\}$
Examples	$\{m \in \mathbb{Z} \mid m < n\}$	and	$\{n \cdot m \mid m \in \mathbb{Z}\}$

The usual tacit convention is that $\in$ binds x . This **seems** innocuous, **BUT**

- Ambiguity is revealed in case p or e is itself of the form $y \in Y$.
Example: let $Even := \{2 \cdot m \mid m \in \mathbb{Z}\}$ (set of even numbers) in

Patterns	$\{x \in X \mid p\}$	and	$\{e \mid x \in X\}$
Examples	$\{n \in \mathbb{Z} \mid n \in Even\}$	and	$\{n \in Even \mid n \in \mathbb{Z}\}$

Both examples match *both patterns*, thereby illustrating the ambiguity.

Worse: notational defects *prohibit even the formulation of formal calculation rules!*
Symptom: formal calculation with set expressions rare/nonexistent in the literature.

1.1 Language design: the four constructs of Functional Mathematics

Introductory remarks

- No “ad hoc” patching of defects, but resynthesize from systematic basis.
- Unifying concept: *function* (= *domain* + *mapping*)
- Language syntax : 4 constructs: identifier, application, abstraction, tupling

Warning: here come a few syntactic technicalities
— but they “repair” all notational defects in engineering mathematics!

0. **Identifier**: any symbol or string except a few keywords.

Identifiers are *introduced* (or *declared*) by *bindings*

- General form: $i : X \wedge p$, read “ i in X satisfying p ”

Here i is the (tuple of) identifier(s), X a set and p a proposition.

Optional: *filter* $\wedge p$ (or **with** p), e.g., $n : \mathbb{N}$ is same as $n : \mathbb{Z} \wedge n \geq 0$

- Identifiers come in two flavors.
 - *Variables*: in an *abstraction* of the form $binding . expression$
Discussed very soon.
 - *Constants*: declared by a *definition* of the form **def** $binding$
Examples follow. Existence and uniqueness are proof obligations.

Well-established symbols, such as $\mathbb{B}$, $\Rightarrow$, $\mathbb{R}$, $+$, serve as predefined constants.

1. Function application:

- Default form: $\boxed{f x}$ for function f and argument e
Other affix conventions: by dashes in the binding, e.g., $— \star —$ for infix.
- Role of parentheses: *never* used as operators; only for parsing.
Precedence rules for making parentheses optional are the usual ones.
If f is a function-valued function, $\boxed{F x y}$ stands for $(F x) y$
- Special application forms for any infix operator $\star$
 - *Partial application* is of the form $x \star$ or $\star y$, and is defined by

$$\boxed{(x \star) y = x \star y = (\star y) x}$$

- *Variadic application* is of the form $x \star y \star z$ etc., *always* defined by

$$\boxed{x \star y \star z = F(x, y, z)}$$

for a suitably defined *elastic extension* F of $\star$, i.e., $F(x, y) = x \star y$.

2. Abstraction:

- General form: $b.e$ where b is a binding $(v : X \wedge p)$ and e an expression.
Intuitive meaning (formalized later): $v : X \wedge p . e$ denotes a *function*
 - Domain = the set of v in X satisfying p ;
 - Mapping: maps v to e .
- Examples
 - (i) The function $n : \mathbb{Z} . 2 \cdot n$ doubles every integer.
 - (ii) If v not free in e (trivial case), we define $\bullet$ by $X \bullet e = v : X . e$
Illustration: $(\mathbb{Z} \bullet 3) 7 = 3$
- Syntactic sugar: $e \mid b$ stands for $b.e$ and $v : X \mid p$ stands for $v : X \wedge p . v$.
- Utilization example: abstractions help synthesizing familiar expressions
such as $\sum i : 0 .. n . q^i$ and $\{m \cdot n \mid m : \mathbb{Z}\}$ and $\{m : \mathbb{Z} \mid m < n\}$.

3. Tupling:

- General form: $\boxed{e, e', e''}$ (any length) for 1 dimension

Intuitive meaning: function with

- Domain: $\mathcal{D}(e, e', e'') = \{0, 1, 2\}$
- Mapping: $(e, e', e'') 0 = e$ and $(e, e', e'') 1 = e'$ and $(e, e', e'') 2 = e''$.
- Parentheses are *not* part of tupling: as optional in (m, n) as in $(m + n)$.
- The empty tuple is ε and for singleton tuples we define τ with $\tau e = 0 \mapsto e$.
Legend: here we used two special cases of $\bullet$:
 - we define ε by $\varepsilon := \emptyset \bullet e$ (any e) for the *empty function*;
 - we define $\mapsto$ by $d \mapsto e = \iota d \bullet e$ for *one-point functions*.
- Matrices are 2-dimensional tuples.

Relax! This concludes the syntactic technicalities.

Next we consider the interesting issues: the formal calculation rules.

1.2 Rules rationale: calculational reasoning, also in logic

- a. **Calculational reasoning:** Generalizes the usual chaining of calculation steps to

$$\begin{array}{l} e_0 \quad R_0 \langle \text{Justification}_0 \rangle \quad e_1 \\ \quad \quad R_1 \langle \text{Justification}_1 \rangle \quad e_2 \quad \text{etc.} \end{array}$$

where R_i, R_{i+1} are mutually transitive, e.g., $=, \leq$ (arithmetic), $\equiv, \Rightarrow$ (logic).

Typical justifications:

- Inference rule: for any theorem p , INSTANTIATION: from p , infer $p[e^v]$

Note: $[e^v]$ expresses substitution of e for v , e.g., $(x \cdot y)[_{3+z}^x = (3 + z) \cdot y$.

- *Equational* reasoning: RST and LEIBNIZ: from $e = e'$ infer $d[e^v] = d[e'^v]$

- b. **Proposition calculus** Propositional operators $\neg, \equiv, \Rightarrow, \wedge, \vee$; constants 0, 1

- The equality operator is $\equiv$ (associative!).
- Set calculus (basic operator $\in$) taken as a derived calculus, e.g.,
$$x \in X \cap Y \equiv x \in X \wedge x \in Y \quad \text{and} \quad x \in X \cup Y \equiv x \in X \vee x \in Y$$

1.3 Rule design: functions, generic functionals, predicates and quantifiers

a. General rules for functions

- *Equality* is defined (taking domains into account) via

Leibniz's principle $f = g \Rightarrow \mathcal{D} f = \mathcal{D} g \wedge (x \in \mathcal{D} f \cap \mathcal{D} g \Rightarrow f x = g x)$

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

- Abstraction encapsulates substitution. Formal axioms for $v : X \wedge p . e$ are:

Domain axiom: $d \in \mathcal{D}(v : X \wedge p . e) \equiv d \in X \wedge p[d]^v$

Mapping axiom: $d \in \mathcal{D}(v : X \wedge p . e) \Rightarrow (v : X \wedge p . e) d = e[d]^v$

Equality is characterized via function equality (exercise).

b. Generic functionals

- Goals:
 - (i) Removing restrictions in common functionals from mathematics.
Example: composition $f \circ g$; common definition requires $\mathcal{R} g \subseteq \mathcal{D} f$
 - (ii) Making often-used implicit functionals from engineering explicit.
Example: $(x \hat{+} y) t = x t + y t$ rather than $(x + y) t = x t + y t$
 - (iii) Supporting the point-free-style (i.e., without variables/dummies)
 $square = times \circ duplicate$ next to $square\ x = times\ (x, x) = x \cdot x$.
- Design principle: defining the domain of the result function in such a way that the image definition does not involve out-of-domain applications.

To be continued soon!

c. Predicates and quantifiers

- **Goal:** calculating with quantifiers as smoothly as with derivatives/integrals. *Practical* use requires a large collection of calculation rules.
- **Definition:** a *predicate* is a boolean-valued function. Here $\mathbb{B} = \{0, 1\}$.
Convention: metavariables P, Q stand for arbitrary predicates.
- **Quantifiers:** axioms and forms of expression
 - Basic axioms: *quantifiers* $\forall, \exists$, are predicates on predicates defined by

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

- Forms of expression: point-free as shown but also other forms.
Taking for P an **abstraction** yields familiar forms like $\forall x : \mathbb{R}. x^2 \geq 0$.
Taking for P a **pair** p, q of boolean expressions yields $\forall (p, q) \equiv p \wedge q$.
So $\forall$ is an elastic extension of $\wedge$, and we define $p \wedge q \wedge r \equiv \forall (p, q, r)$

To be continued soon!

Next topic

0. Motivation: problem, main cause, solution ("formalization to the rescue")
1. Design of a unifying formalism (= language + formal rules)
2. Illustration I — Origin of the basic ideas: signals and systems
 - Step i (origin): signals in control and communications engineering
 - Step ii (generalization): generic functionals for point-free expression
 - Step iii (application): a practical functional predicate calculus
3. Illustration II — A typical generic functional: design and unifying power
4. Illustration III — Typical predicative calculations: exploring program dynamics
5. Conclusions — Unifying Electrical and Computer engineering

2 Illustration I — Origin of the basic ideas: signals & systems

2.0 Step i (origin): signals in control and communications engineering

Note: Signals as functions of time: $\text{Signal space } \mathbb{T} \rightarrow A$ Signal $s : \mathbb{T} \rightarrow A$

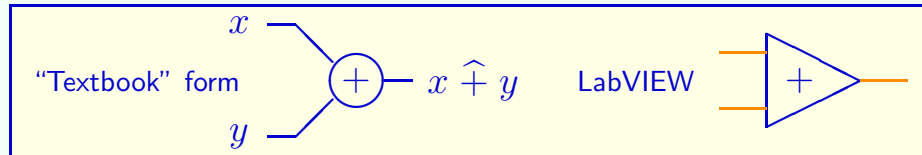
a. Practical need for point-free formulations

- Point-free formulations traditionally seen as relevant to pure theory only. Yet: Any (general) practical formalism needs **both** point-wise and point-free style.
- Modelling signal flow systems: by functionals from input to output signals.
Illustrates first “ad hoc” **functionals**, made generic afterwards as explained.
Extra feature: LabVIEW (a graphical language), as an opportunity for
 - Presenting a language with uncommon yet interesting semantics
 - Using it as one of the application examples of our approach

- Basic building blocks

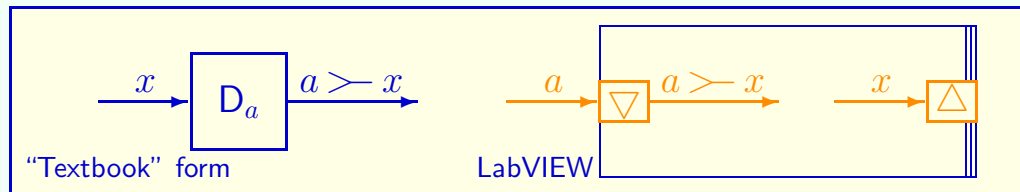
- Memoryless devices realizing arithmetic operations

- * 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)

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



- Time is not structural

Hence transformational design = elimination of the time variable

b. A transformational design example

- From specification to realization

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

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

- Calculational transformation

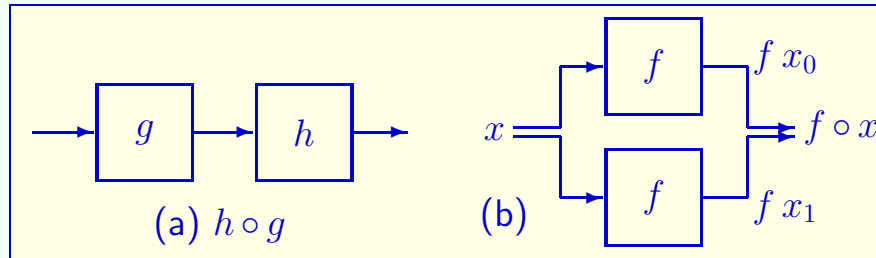
$$\begin{aligned} f\ n &= \langle \text{Def. } f \rangle \quad (n = 0) ? a \upharpoonright g(f\ (n - 1)) \\ &= \langle \text{Def. } \circ \rangle \quad (n = 0) ? a \upharpoonright (g \circ f)\ (n - 1) \\ &= \langle \text{Def. } D \rangle \quad D_a(g \circ f)\ n \\ &= \langle \text{Def. } \overline{=}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

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

- Functionals introduced (types designed afterwards by generification)

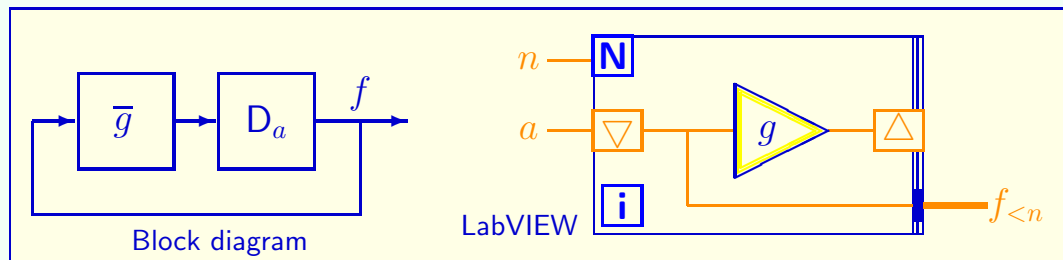
- Function composition ($\circ$), defined by $(f \circ g)\ x = f\ (g\ x)$
- Direct extension, 1 argument ($\overline{=}$), defined by $\overline{g}\ x = g \circ x$

- 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 $f = (D_a \circ \overline{g}) f$



2.1 Step ii (generalization): generic functionals for point-free expression

- a. Design principle: *no* restrictions on the argument functions: “out of domain” applications avoided by judiciously defining the domain of the result function.
- b. Illustration (in the order of the 3 goals): for any functions f, g , predicate P):

(i) Removing restrictions in functionals. Example: composition ($\circ$)

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

(ii) Making useful functionals explicit. Example: direct extension ($\hat{\star}$)

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

(iii) Eliminating/introducing dummies: “filtering” ($\downarrow$)

$$f \downarrow P = x : \mathcal{D} f \cap \mathcal{D} P \wedge P x . f x \quad \text{Shorthand: } f_P$$

A particularization: the familiar *restriction* ($\restriction$): $f \restriction X = f \downarrow (X \bullet 1)$.

Another example: compatibility ($\odot$): $f \odot g \equiv f \restriction \mathcal{D} g = g \restriction \mathcal{D} f$

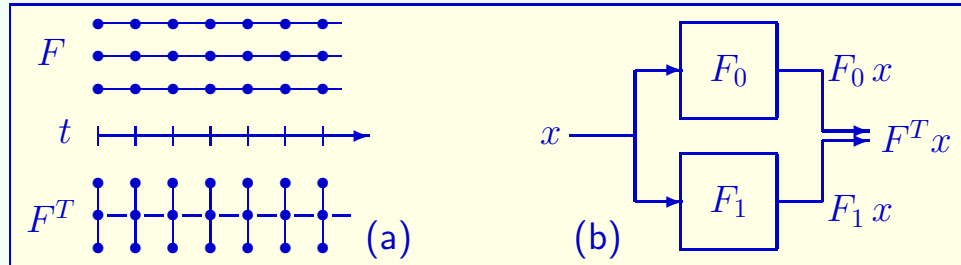
c. Two more examples of generic functional design

- **Transposition** (---^T) (already seen: composition, direct extension)

– Purpose: swapping the arguments of a functional: $F^T y x = F x y$

– Structural interpretations:

(a) From a family of signals to a tuple-valued signal; (b) Signal fanout



– Generic version (one variant): $F^T = y : \bigcap (\mathcal{D} \circ F) . x : \mathcal{D} F . F x y$

- **Function merge** ($\cup$), defined here in 2 parts to fit the line width:

$$\begin{aligned} x \in \mathcal{D}(f \cup g) &\equiv x \in \mathcal{D} f \cup \mathcal{D} g \wedge (x \in \mathcal{D} f \cap \mathcal{D} g \Rightarrow f x = g x) \\ x \in \mathcal{D}(f \cup g) &\Rightarrow (f \cup g) x = (x \in \mathcal{D} f) ? f x \upharpoonright g x \end{aligned}$$

2.2 Step iii (application): a practical functional predicate calculus

Goal: calculating with quantifiers as fluently as with derivatives and integrals

a. Quantifiers as predicates on predicates (reminder)

- Recall: constant function definer ($\bullet$): $X \bullet e = x : X . e$ with fresh x .
- Defining quantifiers $\forall$ and $\exists$: for any predicate P ,

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

Taking for P an abstraction yields familiar forms like $\forall x : \mathbb{R} . x \geq 0$.

Example of a typical derivation of an algebraic property (calculation rule)

$$\begin{aligned} & \forall P \wedge \forall Q \\ \equiv & \langle \text{Def. } \forall \rangle \quad P = \mathcal{D} P \bullet 1 \wedge Q = \mathcal{D} Q \bullet 1 \\ \Rightarrow & \langle \text{Leibniz} \rangle \quad \forall (P \hat{\wedge} Q) \equiv \forall (\mathcal{D} P \bullet 1 \hat{\wedge} \mathcal{D} Q \bullet 1) \\ \equiv & \langle \text{Def. } \hat{\wedge} \rangle \quad \forall (P \hat{\wedge} Q) \equiv \forall x : \mathcal{D} P \cap \mathcal{D} Q . (\mathcal{D} P \bullet 1) x \wedge (\mathcal{D} Q \bullet 1) x \\ \equiv & \langle \text{Def. } \bullet \rangle \quad \forall (P \hat{\wedge} Q) \equiv \forall x : \mathcal{D} P \cap \mathcal{D} Q . 1 \wedge 1 \\ \equiv & \langle \forall (X \bullet 1) \rangle \quad \forall (P \hat{\wedge} Q) \end{aligned}$$

b. For practical use, there is a large collection of such algebraic calculation rules.

Example: relating $\forall/\exists$ by *duality* (or *generalized De Morgan's law*)

$$\neg \forall P \equiv \exists (\neg P) \text{ or, in pointwise form, } \neg (\forall v : X . p) \equiv \exists v : X . \neg p$$

Distributivity rules (each has a dual, not stated here):

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

Pointwise example: $\exists (v : X . p) \Rightarrow q \equiv \forall (v : X . p \Rightarrow q)$ provided $v \notin \varphi q$.

As in algebra, the nomenclature is very helpful for familiarization and use.

Distributivity $\vee/\forall$ generalizes $q \vee (r \wedge s) \equiv (q \vee r) \wedge (q \vee s)$

L(eft)-distrib. $\Rightarrow/\forall$ generalizes $q \Rightarrow (r \wedge s) \equiv (q \Rightarrow r) \wedge (q \Rightarrow s)$

R(ight)-distr. $\Rightarrow/\exists$ generalizes $(r \vee s) \Rightarrow q \equiv (r \Rightarrow q) \wedge (s \Rightarrow q)$

P(seudo)-dist. $\wedge/\forall$ generalizes $q \wedge (r \wedge s) \equiv (q \wedge r) \wedge (q \wedge s)$

Derived rules (continued)

Some additional laws

Name	Point-free form
Distrib. $\forall/\wedge$	$\forall (P \hat{\wedge} Q) \Leftarrow \forall P \wedge \forall Q$
One-point rule	$\forall P_{=e} \equiv e \in \mathcal{D} P \Rightarrow P e$
Trading $\forall$	$\forall P_Q \equiv \forall (Q \hat{\Rightarrow} P)$
Transp./Swap	$\forall (\forall \circ R) \equiv \forall (\forall \circ R^\top)$

Note: $\mathcal{D} P = \mathcal{D} Q \Rightarrow \forall (P \hat{\wedge} Q) \Rightarrow \forall P \wedge \forall Q$.

Name	Pointwise form
Distrib. $\forall/\wedge$	$\forall (v : X . p \wedge q) \Leftarrow \forall (v : X . p) \wedge \forall (v : X . q)$
One-point rule	$\forall (v : X . v = e \Rightarrow p) \equiv e \in X \Rightarrow p[e^v]$
Trading $\forall$	$\forall (v : X \wedge q . p) \equiv \forall (v : X . q \Rightarrow p)$
Transp./Swap	$\forall (v : X . \forall w : T . p) \equiv \forall (w : T . \forall v : X . p)$

c. Wrapping up the rule package for function(al)s

- **Definition:** we define the *function range* operator $\mathcal{R}$ by

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

- **Consequence:** $\forall P \Rightarrow \forall (P \circ f)$ and $\mathcal{D} P \subseteq \mathcal{R} f \Rightarrow (\forall (P \circ f) \equiv \forall P)$

Pointwise form: $\forall (y : \mathcal{R} f . p) \equiv \forall (x : \mathcal{D} f . p[\frac{y}{f x}])$ (“dummy change”).

- **An important application:** set comprehension

Basis: we define $\{—\}$ as *fully interchangeable* with $\mathcal{R}$.

Consequence: defect-free set notation:

- Expressions like $\{2, 3, 5\}$ and $\{2 \cdot m \mid m : \mathbb{Z}\}$ have familiar form & meaning
- All desired calculation rules follow from predicate calculus via $\mathcal{R}$.
- In particular, we can prove $e \in \{v : X \mid p\} \equiv e \in X \wedge p[\frac{v}{e}]$ (exercise).

Next topic

0. Motivation: problem, main cause, solution ("formalization to the rescue")
1. Design of a unifying formalism (= language + formal rules)
2. Illustration I — Origin of the basic ideas: signals and systems
3. Illustration II — A typical generic functional: design and unifying power
 - Step i (origin): tolerances in engineering extended to functions
 - Step ii (generalization): generalized functional Cartesian product
 - Step iii (applications): various topics in computing
4. Illustration III — Typical predicative calculations: exploring program dynamics
5. Conclusions — Unifying Electrical and Computer engineering

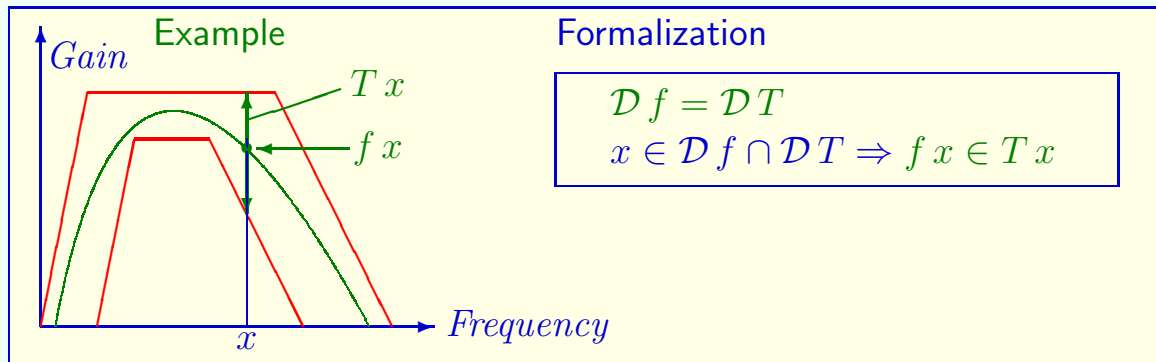
3 Illustration II — A typical generic functional

3.0 Step i (origin): tolerances in engineering extended to functions

- a. Tolerances for scalars: used routinely for all classical engineering artefacts
- b. Tolerances for functions: formalizing a convention in communications:

A *tolerance function* T specifies for every domain value x the set Tx of allowable function values. Note: $\mathcal{D}T$ also taken as the domain specification.

Example: radio frequency filter characteristic and its formalization



3.1 Step ii (generalization): generalized functional Cartesian product

a. *Generalized Functional Cartesian Product* $\times$: for *any* family T of sets,

Definition: $f \in \times T \equiv \mathcal{D} f = \mathcal{D} T \wedge \forall x : \mathcal{D} f \cap \mathcal{D} T . f x \in T x$
 equivalently: $\times T = \{f : \mathcal{D} T \rightarrow \bigcup T \mid \forall x : \mathcal{D} f \cap \mathcal{D} T . f x \in T x\}$

b. *Some properties* illustrating why $\times$ is our “workhorse” for types

Cartesian product:	$A \times B = \times(A, B)$
Function type:	$A \rightarrow B = \times(A \bullet B)$
Point-free form	$\times T = \{f : \mathcal{D} T \rightarrow \bigcup T \mid \forall (f \hat{\in} T)\}$
Explicit inverse	$\times^- S = x : \bigcup (f : S . \mathcal{D} f) . \{f x \mid f : S\}$
Function <i>equality</i> :	$f = g \equiv f \in \times(\iota \circ g)$
Dependent type	$\times(a : A . B_a) = \{f : A \rightarrow \bigcup (a : A . B_a) \mid \forall a : A . f a \in B_a\}$

Useful shorthand: $A \ni a \rightarrow B_a$ for $\times a : A . B_a$, as in: $A \ni a \rightarrow B_a \ni b \rightarrow C_{a,b}$

3.2 Step iii (applications): various topics in computing

a. Aggregate data types (all aggregates are functions!) Some typical cases:

- List types: $A^n = \times (\square n \bullet A)$ and $A^* = \bigcup n : \mathbb{N}. A^n$ and so on
- Record types: defining $\text{Record } F = \times (\bigcup F)$ for $F : \text{Fam}(\text{Fam}\mathcal{T})$

Example: if we let $\text{Person} := \text{Record } (name \mapsto \mathbb{A}^*, age \mapsto \mathbb{N})$

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

b. Overloading and polymorphism

- Aspects to be covered: disambiguation and refined typing
- Two main operators: (for family F of function types to be combined)
 - Parametrized (Church style): simply $\times F$
 - Unparametrized (Curry style): function type merge

$\text{def } \otimes : \text{Fam}(\mathcal{P}\mathcal{F}) \rightarrow \mathcal{P}\mathcal{F} \text{ with } \otimes F = \{\bigcup f \mid f : \times F \wedge \odot f\}$

c. Relational databases

- Formal description: by declarations (here explained by example)

```
def CID := Record (code ↦ Code, name ↦  $\mathbb{A}^*$ , inst ↦ Staff, prrq ↦ Code*)
```

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
...	...	...	

- Query: all usual query-operators are subsumed by generic functionals

- The usual *selection*-operator (σ) by $\sigma(S, P) = S \downarrow P$.
- The usual *projection*-operator (π) by $\pi(S, F) = \{r \upharpoonright F \mid r : S\}$.
- The usual *join*-operator ($\bowtie$) by $S \bowtie T = S \otimes T$.

Observation: this is the polymorphism-operator.

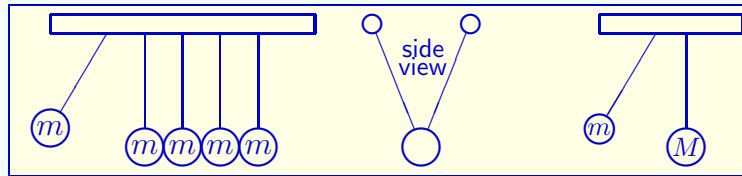
In pointwise form: $S \bowtie T = \{s \cup t \mid (s, t) : S \times T \wedge s \odot t\}$.

Next topic

0. Motivation: problem, main cause, solution ("formalization to the rescue")
1. Design of a unifying formalism (= language + formal rules)
2. Illustration I — Origin of the basic ideas: signals and systems
3. Illustration II — A typical generic functional: design and unifying power
4. Illustration III — Typical predicative calculations: exploring program dynamics
 - Step i (analogy): dynamics of colliding balls
 - Step ii: expressing program dynamics by program equations
 - Step iii: calculational deriving various "axiomatic" semantics
5. Conclusions — Unifying Electrical and Computer engineering

4 Illustration III — Typical calculations: in program dynamics

4.0 Step i (analogy): dynamics of colliding balls ("Newton's Cradle")



State $s := v, V$ (velocities); $\backslash s$ before and s' after collision. Lossless collision:

$$\begin{aligned} R(\backslash s, s') &\equiv m \cdot \backslash v + M \cdot \backslash V = m \cdot v' + M \cdot V' && \text{— momentum} \\ &\wedge m \cdot \backslash v^2 + M \cdot \backslash V^2 = m \cdot v'^2 + M \cdot V'^2 && \text{— energy } (\cdot 2) \end{aligned}$$

Letting $a := M/m$, assuming $v' \neq \backslash v$ and $V' \neq \backslash V$ (discarding the trivial case):

$$R(\backslash s, s') \equiv v' = -\frac{a-1}{a+1} \cdot \backslash v + \frac{2 \cdot a}{a+1} \cdot \backslash V \wedge V' = \frac{2}{a+1} \cdot \backslash v + \frac{a-1}{a+1} \cdot \backslash V$$

Crucial point: mathematics is not used as just a “compact language”; rather: the calculations yield insights that are hard to obtain by intuition.

4.1 Step ii: expressing program dynamics by program equations

Program equations for a simple language (Dijkstra's guarded commands)

State change expressed by $R: C \rightarrow \mathbb{S}^2 \rightarrow \mathbb{B}$, termination by $T: C \rightarrow \mathbb{S} \rightarrow \mathbb{B}$.

Syntax: command c	State change $R\ c\ (s, s')$
$v := e$	$s' = s[e^v]$
skip	$s' = s$
abort	0
$c' ; c''$	$\exists t. R\ c'\ (s, t) \wedge R\ c''\ (t, s')$
if $\square i: I. b_i \rightarrow c'_i$ fi	$\exists i: I. b_i \wedge R\ c'_i\ (s, s')$
Syntax: command c	Termination $T\ c\ s$
$v := e$	1
skip	1
abort	0
$c' ; c''$	$T\ c'\ s \wedge \forall t. R\ c'\ (s, t) \Rightarrow T\ c''\ t$
if $\square i: I. b_i \rightarrow c'_i$ fi	$\exists b \wedge \forall i: I. b_i \Rightarrow T\ c'_i\ s$

Iteration command c is do $b \rightarrow c'$ od; dynamics if $\neg b \rightarrow \text{skip} \ \square \ b \rightarrow (c'; c)$ fi

4.2 Step iii: calculationaly deriving various “axiomatic” semantics

- a. **Abbreviations:** in the sequel, we shall
 - often write $s.e$ for $s:\mathbb{S}.e$ (since the domain is always $\mathbb{S}$);
 - often use either s, s' or $\text{'s}, s$ instead of $\text{'s}, s'$ (just dummies!).
- b. **Ante-/postcondition semantics** expressed via equations (no “special logics”)

Let $\text{pred}_X = X \rightarrow \mathbb{B}$ for any set X , so $\text{pred}_{\mathbb{S}}$ is the set of state predicates.

Anteconditions A (“before”) and postconditions P (“after”) are of this type.

We define Hoare triples by functions of type $\boxed{\text{pred}_{\mathbb{S}} \times C \times \text{pred}_{\mathbb{S}} \rightarrow \mathbb{B}}$

We express termination for given antecondition by $\boxed{\text{Term} : C \rightarrow \text{pred}_{\mathbb{S}} \rightarrow \mathbb{B}}$

$\{A\} c \{P\} \equiv \forall \text{'s}. \forall s'. A \text{'s} \wedge R c(\text{'s}, s') \Rightarrow P s'$	“partial correctness”
$[A] c [P] \equiv \{A\} c \{P\} \wedge \text{Term} c A$	“total correctness”
$\text{Term} c A \equiv \forall s. A s \Rightarrow T c s$	“termination”

Intuitive justification: given antecondition A , all that is known about the relation between 's and s' is $A \text{'s}$ and $R c(\text{'s}, s')$. So this must imply $P s'$.

- c. Calculate all properties of interest *Just predicate calculus, no special logics!*

Example: weakest antecondition semantics (Dijkstra style). Definitions:

- *Weakest liberal antecondition*: weakest A satisfying $\{A\} c \{P\}$
- *Weakest antecondition*: weakest A satisfying $[A] c [P]$

Computational derivation of an expression for such anteconditions: push A out

$$\begin{aligned}
& [A] c [P] \\
\equiv & \langle \text{Def. } [A] c [P] \rangle \quad \{A\} c \{P\} \wedge \text{Term } c A \\
\equiv & \langle \text{Def. } \{A\} c \{P\} \rangle \quad \forall (s. \forall s'. A s \wedge R c(s, s') \Rightarrow P s') \wedge \text{Term } c A \\
\equiv & \langle \text{Def. Term } c A \rangle \quad \forall (s. \forall s'. A s \wedge R c(s, s') \Rightarrow P s') \wedge \forall (s. A \Rightarrow T c s) \\
\equiv & \langle \text{Distr. } \forall / \wedge \rangle \quad \forall s. \forall (s'. A s \wedge R c(s, s') \Rightarrow P s') \wedge (A s \Rightarrow T c s) \\
\equiv & \langle \text{Shunt } \wedge / \Rightarrow \rangle \quad \forall s. \forall (s'. A s \Rightarrow R c(s, s') \Rightarrow P s') \wedge (A s \Rightarrow T c s) \\
\equiv & \langle \text{Ldist. } \Rightarrow / \forall \rangle \quad \forall s. (A s \Rightarrow \forall s'. R c(s, s') \Rightarrow P s') \wedge (A s \Rightarrow T c s) \\
\equiv & \langle \text{Ldist. } \Rightarrow / \wedge \rangle \quad \forall s. A s \Rightarrow \forall (s'. R c(s, s') \Rightarrow P s') \wedge T c s
\end{aligned}$$

So $[A] c [P] \equiv \forall s. A s \Rightarrow \forall (s'. R c(s, s') \Rightarrow P s') \wedge T c s$. Hence we define

def $wla : C \rightarrow \text{pred}_{\mathbb{S}} \rightarrow \text{pred}_{\mathbb{S}}$ **with** $wla c P s \equiv \forall s'. R c(s, s') \Rightarrow P s'$
def $wa : C \rightarrow \text{pred}_{\mathbb{S}} \rightarrow \text{pred}_{\mathbb{S}}$ **with** $wa c P s \equiv wla c P s \wedge T c s$

d. Results and more analogies

- From the preceding, we obtain by functional predicate calculus:

$$\begin{aligned}
 \text{wa } \llbracket v := e \rrbracket P s &\equiv P(s[e^v]) \\
 \text{wa } \llbracket c' ; c'' \rrbracket &\equiv \text{wa } c' \circ \text{wa } c'' \\
 \text{wa } \llbracket \text{if } \llbracket i : I . b_i \rightarrow c'_i \text{ fi} \rrbracket P s &\equiv \exists b \wedge \forall i : I . b_i \Rightarrow \text{wa } c'_i P s \\
 \text{wa } \llbracket \text{do } b \rightarrow c' \text{ od} \rrbracket P s &\equiv \exists n : \mathbb{N} . w^n (\neg b \wedge P s) \text{ defining } w \text{ by} \\
 w q &\equiv (\neg b \wedge P s) \vee (b \wedge \text{wa } c' (s \bullet q) s)
 \end{aligned}$$

Warning: due to a syntactic shortcut, s = tuple of all program variables.

- Remark: practical rules for loops (invariants, bound functions) similarly
- Analogies: Green functions (for linear device d), Fourier transforms

$$\begin{aligned}
 \text{wla } c P s &\equiv \forall s' : \mathbb{S} . R c(s, s') \Rightarrow P s' \\
 \text{Rsp } d f x &= \mathcal{I} x' : \mathbb{R} . G d(x, x') \cdot f x' \quad (\text{linear } d) \\
 \text{Rsp } d f t &= \mathcal{I} t' : \mathbb{R} . h d(t - t') \cdot f t' \quad (\text{for LTI } d) \\
 \mathcal{F} f \omega &= \mathcal{I} t : \mathbb{R} . \exp(-j \cdot \omega \cdot t) \cdot f t
 \end{aligned}$$

Final topic

0. Motivation: problem, main cause, solution ("formalization to the rescue")
1. Design of a unifying formalism (= language + formal rules)
2. Illustration I — Origin of the basic ideas: signals and systems
3. Illustration II — A typical generic functional: design and unifying power
4. Illustration III — Typical predicative calculations: exploring program dynamics
5. Conclusions — Unifying Electrical and Computer engineering

5 **Conclusions — Unifying Engineering Disciplines**

- What we have shown
 - A formalism with a very simple language and powerful formal rules
 - Notational and methodological unification of CS and classical engineering
 - Unification also encompassing a large part of mathematics.
- Ramifications
 - Scientific: obvious
 - Educational: unified basis for ECE (Electrical and Computer Engineering)
- Problems to be recognized
 - Students find logic difficult (cause: de-emphasis on proofs in education)
 - Conservatism of colleagues possibly larger problem (even censorship).
- Conclusion Long-term advantages outweigh temporary “mathphobic” trends.