

Tutorial — FM 2005

Formal Methods as a Unifying Basis for Electrical and Computer Engineering

Raymond Boute, INTEC — Ghent University

Tuesday, 2005-07-19

09:00–09:10	0. Introduction: purpose and approach
	Lecture I. The formalism: Functional Mathematics
09:10–09:30	1. Part A: the language
09:30–10:30	2. Part B: the calculation rules
10:30–11:00	(Half-hour break)
	Lecture II. Illustrations: unifying engineering mathematics
11:00–11:45	3. Part A: classical engineering and continuous mathematics
11:45–12:30	4. Part B: computer engineering and discrete mathematics

0 Introduction: purpose and approach

0.0 Motivation: bridging the 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: rift in the use of mathematical modelling in everyday practice
 - In classical engineering (electrical, ...): mathematical modelling regular
 - In software "engineering": mathematical modelling rarely used (occasionally in critical systems under the name "Formal Methods")
C. Michael Holloway: "software designers aspire to be(come) engineers"
- Differences 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 The methodology rift mirrors a style breach throughout mathematics

Consider the degree of systematic symbolic calculation in “everyday mathematics”

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

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 $\sqrt[3]{2 + 11i}$.

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 (ϵ - δ)-proofs in analysis with no help in understanding the logical structure of the arguments. Examiners fully deserve the garbage that they get in return.”

(Paul 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)

0.2 Approach: exploiting the advantages of formalization

- “Formal approach” means: 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 HW/SW 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!)

1 The formalism, part A: the language

1.0 Language rationale: the need for defect-free notation

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

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

- **Ellipsis**, i.e., “omission dots” (\dots) as in $a_0 + a_1 + \dots + a_n$
Common use violates Leibniz’s principle (substitution of equals for equals)
Example: $a_i = i^2$ and $n = 7$ yields $0 + 1 + \dots + 49$ (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 e)

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 Formalism design: Functional Mathematics (Funmath)

- Unifying formalism for continuous and discrete mathematics
 - *Formalism* = *language* (notation) + *formal rules*
 - Unifying concept: *function* (= *domain* + *mapping*)
Functions as first-class objects and basis for unification
- The language (characteristics)
 - No “ad hoc” patching of defects, but restart from systematic basis.
 - Simple structure: 4 constructs: identifier, application, abstraction, tupling
Synthesizing common notations, without their defects
Synthesizing new useful forms of expression, in particular: “point-free”,
e.g.: $square = times \circ duplicate$ versus $square\ x = x\ times\ x$
- Formal rules (main characteristic): *calculational*

Warning: here come a few syntactic technicalities

— but they “repair” all notational defects in engineering mathematics!

1.2 The four constructs

Overview: 0. identifier, 1. application, 2. abstraction, 3. tupling

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

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

- General form: $\boxed{i : X \wedge p}$, 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., $\boxed{n : \mathbb{N}}$ is same as $\boxed{n : \mathbb{Z} \wedge n \geq 0}$

Identifiers from i should not appear in expression X .

- Identifiers come in two flavors.

- *Variables*: in an *abstraction* of the form $\boxed{\text{binding} . \text{expression}}$

Discussed very soon.

- *Constants*: declared by a *definition* of the form $\boxed{\text{def binding}}$

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: $f x$ 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 (overruling/emphasizing affix conventions/precedence).
Precedence rules for making parentheses optional are the usual ones.
If f is a function-valued function, $f x y$ stands for $(f x) y$
- Special application forms for any infix operator \star

– *Partial application* is of the form $a \star$ or $\star b$, and is defined by

$$(a \star) b = a \star b = (\star b) a$$

– *Variadic application* is of the form $a \star b \star c$ etc., *always* defined by

$$a \star b \star c = F(a, b, c)$$

for a suitably defined *elastic extension* F of \star .

2. Abstraction:

- General form: $\boxed{b.e}$ where
 - b is a binding and
 - e an expression, extending after “.” as far as parentheses permit.Intuitive meaning: $v : X \wedge p . e$ denotes a *function*
 - Domain = the set of v in X satisfying p ;
 - Mapping: maps v to e (as in lambda calculus)
- 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 $\boxed{X \bullet e = v : X . e}$
Illustration: $(\mathbb{Z} \bullet 3) 7 = 3$
- Syntactic sugar: $\boxed{e \mid b}$ stands for $b . e$ and $\boxed{v : X \mid p}$ stands for $v : X \wedge p . v$.
- We shall see how abstractions help synthesizing familiar expressions such as $\boxed{\sum i : 0 .. n . q^i}$ and $\boxed{\{m \cdot n \mid m : \mathbb{Z}\}}$ and $\boxed{\{m : \mathbb{Z} \mid m < n\}}$.

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.

2 The formalism, part B: the calculation rules

2.0 Rules for equational and calculational reasoning

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

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

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

- **General inference rule:** For any theorem p ,

$$\text{INSTANTIATION: from } p, \text{ infer } p[e^v]$$

Note: $[e^v]$ or $[v := e]$ expresses substitution of e for v , for instance,

$$(x + y = y + x)[x, y := 3, z + 1] \text{ stands for } 3 + (z + 1) = (z + 1) + 3.$$

- **Equational reasoning:** basic rules are reflexivity, symmetry, transitivity and

$$\text{LEIBNIZ'S PRINCIPLE: from } e = e', \text{ infer } d[e^v] = d[e'^v]$$

2.1 Two styles: pointwise (with variables) and point-free (w/o variables)

a. **Lambda calculus** as an archetype of the pointwise style. A reminder:

- Language: $term ::= variable \mid \underline{(term\ term)} \mid \underline{(\lambda variable_term)}$

Examples: $x \quad (xy) \quad (\lambda x.x) \quad (\lambda x.(\lambda y.(x(yz))))$

Plus conventions making many parentheses optional, e.g., LMN for $(LM)N$

- Rules: equality (symmetry, transitivity and Leibniz's principle) plus:

Axiom, beta conversion: $(\lambda v.M)N = M \underset{N}{[v]}$

Axiom, alpha conversion: $(\lambda v.M) = (\lambda w.M \underset{w}{[v]})$ provided $w \notin \varphi M$

Rule ζ (extensionality): $\frac{Mv = Nv}{M = N}$ provided $v \notin \varphi(M, N)$

b. **Combinator terms** as an archetype of the point-free style. A reminder:

- Language: $term ::= \underline{K} \mid \underline{S} \mid \underline{(term\ term)}$

- Rules: equality (with variant for Leibniz), and

axioms $\underline{KLM = L}$ and $\underline{SPQR = PR(QR)}$ plus extensionality.

2.2 Rules for calculating with propositions and sets

- **Proposition calculus** Usual propositional operators $\neg, \equiv, \Rightarrow, \wedge, \vee$. Notes:
 - For practical use, an extensive set of rules is needed (see e.g. Gries)
 - Note: \equiv is associative, \Rightarrow is not. We read $p \Rightarrow q \Rightarrow r$ as $p \Rightarrow (q \Rightarrow r)$.
 - Binary algebra is embedded in arithmetic. Logic constants are 0 and 1.
 - Leibniz's principle can be rewritten $e = e' \Rightarrow d[e^v] = d[e'^v]$.
- **Calculating with sets** The basic operator is \in .
 - The rules are derived ones (set calculus from proposition calculus), e.g.,

Set intersection \cap is defined by $x \in X \cap Y \equiv x \in X \wedge x \in Y$
Cartesian product \times is defined by $x, y \in X \times Y \equiv x \in X \wedge y \in Y$
After defining $\{—\}$, we can prove $y \in \{x : X \mid p\} \equiv y \in X \wedge p[x]$

- *Set equality* is defined via

Leibniz's principle: $X = Y \Rightarrow (x \in X \equiv x \in Y)$, and the converse:
Extensionality: from $x \in X \equiv x \in Y$ (with new x), infer $X = Y$.

2.3 Binary algebra and calculating with conditionals

- a. **Principle:** binary algebra as a restriction of *minimax algebra*, i.e., *least upper bound* (\vee) and *greatest lower bound* (\wedge) operators over $\mathbb{R}' := \mathbb{R} \cup \{-\infty, +\infty\}$.

Definition: $a \vee b \leq c \equiv a \leq c \wedge b \leq c$ and $c \leq a \wedge b \equiv c \leq a \wedge c \leq b$

Restriction to \mathbb{B} is illustrated by listing $f_i(x, y)$ for $i: 1..15$ (functions $\mathbb{B}^2 \rightarrow \mathbb{B}$).

x, y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0,0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0,1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1,0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1,1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
\mathbb{B}		\forall	$<$		$>$		\neq	\wedge	\wedge	\equiv	\gg	\Rightarrow	\ll	\Leftarrow	\vee	
\mathbb{R}'			$<$		$>$		\neq		\wedge	$=$	\gg	\leq	\ll	\geq	\vee	

- b. **Conditionals** $c?e' \dagger e = (e, e')c$ (“if c then e' else e ”) Properties:

- $(c?f \dagger g)x = c?fx \dagger gx$ and $f(c?x \dagger y) = c?fx \dagger fy$
- $c?b \dagger b' \equiv (c \Rightarrow b) \wedge (\neg c \Rightarrow b')$ for boolean b, b'
- $z = (c?x \dagger y) \equiv (c \Rightarrow z = x) \wedge (\neg c \Rightarrow z = y)$

2.4 Rules for calculating with functions and generic functionals

a. General rules for functions

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

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

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

- Abstraction encapsulates substitution. Formal axioms:

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

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

Equality is characterized via function equality (exercise).

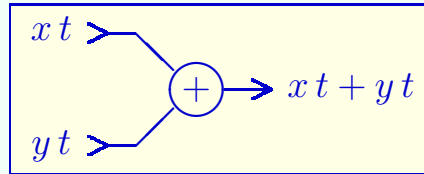
b. Generic functionals

- Goals:

- (a) Removing restrictions in common functionals from mathematics.

- Example: composition $f \circ g$; common definition requires $\mathcal{R}g \subseteq \mathcal{D}f$

- (b) Making often-used implicit functionals from systems theory explicit.



Usual notations: $(x + y) t = x t + y t$ (overloading $+$)

or: $(x \oplus y) t = x t + y t$ (special symbol)

- Design principle: defining the domain of the result function in such a way that the image definition does not involve out-of-domain applications.

This applies to goal (a), goal (b) and new designs (discussed next).

- Design illustrating goal (a): *composition* (\circ)

For any functions f, g ,

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

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

- Design illustrating goal (b): *(Duplex) direct extension* ($\hat{}$)

For any functions \star (infix), f, g ,

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

Example: given $f : \mathbb{N} \rightarrow \mathbb{R}$ and $g : \mathbb{Z} \rightarrow \mathbb{C}$ we get $\mathcal{D}(f \hat{+} g) = \mathbb{N}$.

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

$$f \overleftarrow{\star} e = f \hat{\star} (\mathcal{D}f \bullet e) \quad \text{and} \quad e \overrightarrow{\star} f = (\mathcal{D}f \bullet e) \hat{\star} f$$

Typical algebraic property: $x \overrightarrow{\star} f = (x \star) \circ f$

Simplex direct extension ($\overline{}$) is defined by

$$\overline{f}g = f \circ g$$

c. Generic functionals (continued:): some other important generic functionals

- *Function merge* (\cup) is defined in 2 parts to fit the line:

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

- *Filtering* (\downarrow) introduces/eliminates arguments: (here P is a predicate)

$$f \downarrow P = x : \mathcal{D}f \cap \mathcal{D}P \wedge P x . f x$$

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

We extend \downarrow to sets: $x \in (X \downarrow P) \equiv x \in X \cap \mathcal{D}P \wedge P x$.

Writing a_b for $a \downarrow b$ and using partial application, this yields formal rules for useful shorthands like $f_{<n}$ and $\mathbb{Z}_{>0}$.

- *Function compatibility* (\odot) is a relation on functions:

$$f \odot g \equiv f \lrcorner \mathcal{D}g = g \lrcorner \mathcal{D}f$$

Algebraic property: $f = g \equiv \mathcal{D}f = \mathcal{D}g \wedge f \odot g$.

2.5 Rules for calculating with predicates and quantifiers

Preliminary remarks

- Goal: formally calculating with quantifiers as fluently as with derivatives/integrals.
- *Practical* use requires a large collection of calculation rules.
- Here only give the axioms and most important derived rules.

Axioms and calculation rules

a. 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

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

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 $p \wedge q \wedge r \equiv \forall(p, q, r)$

b. Derived rules

Relating \forall/\exists by *duality* (or *generalized De Morgan's law*)

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

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

Name of the rule	Point-free form	Letting $P := v : S . p$ with $v \notin \varphi q$
Distributivity \vee/\forall	$q \vee \forall P \equiv \forall (q \vec{\vee} P)$	$q \vee \forall (v : S . p) \equiv \forall (v : S . q \vee p)$
L(eft)-distrib. \Rightarrow/\forall	$q \Rightarrow \forall P \equiv \forall (q \vec{\Rightarrow} P)$	$q \Rightarrow \forall (v : S . p) \equiv \forall (v : S . q \Rightarrow p)$
R(ight)-distr. \Rightarrow/\exists	$\exists P \Rightarrow q \equiv \forall (P \vec{\Leftarrow} q)$	$\exists (v : S . p) \Rightarrow q \equiv \forall (v : S . p \Rightarrow q)$
P(seudo)-dist. \wedge/\forall	$q \wedge \forall P \equiv \forall (q \vec{\wedge} P)$	$q \wedge \forall (v : S . p) \equiv \forall (v : S . q \wedge p)$

Note: \wedge/\forall assumes $\mathcal{D}P \neq \emptyset$. The general form is $(p \wedge \forall P) \vee \mathcal{D}P = \emptyset \equiv \forall (p \vec{\wedge} P)$

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 $q \wedge (r \wedge s) \equiv (q \wedge r) \wedge (q \wedge s)$

c. Derived rules (continued)

Some additional laws

Name	Point-free form	Letting $P := v : S . p$ with $v \notin \varphi q$
Distrib. \forall/\wedge	$\forall(P \widehat{\wedge} Q) \equiv \forall P \wedge \forall Q$	$\forall(v : S . p \wedge q) \equiv \forall(v : S . p) \wedge \forall(v : S . q)$
One-point rule	$\forall P_{=e} \equiv e \in \mathcal{D}P \Rightarrow P e$	$\forall(v : S . v = e \Rightarrow p) \equiv e \in S \Rightarrow p _e^v$
Trading \forall	$\forall P_Q \equiv \forall(Q \widehat{\Rightarrow} P)$	$\forall(v : S \wedge q . p) \equiv \forall(v : S . q \Rightarrow p)$
Transp./Swap	$\forall(\forall \circ R) = \forall(\forall \circ R^T)$	$\forall(v : S . \forall w : T . p) \equiv \forall(w : T . \forall v : S . p)$

Note: \forall/\wedge assumes $\mathcal{D}P = \mathcal{D}Q$. Without this condition, $\forall P \wedge \forall Q \Rightarrow \forall(P \widehat{\wedge} Q)$.

Just one derivation example:

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

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

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

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

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

- **Set comprehension**

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

Consequence: defect-free set notation:

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

2.6 Unifying induction principles via predicate calculus

- a. Some definitions: for any relation $\prec: X^2 \rightarrow \mathbb{B}$, any $S: \mathcal{P} X$ and any $x: X$,

Minimal element: $x \text{ ismin}_{\prec} S \equiv x \in S \wedge \forall y: X. y \prec x \Rightarrow y \notin S$

Least element: $x \text{ isleast}_{\prec} S \equiv x \in S \wedge \forall y: S. x \prec y$

- b. Well-foundedness and supporting induction

- Defining *Well-Foundedness*: every nonempty subset has a minimal element

$$\text{WF}(\prec) \equiv \forall S: \mathcal{P} X. S \neq \emptyset \Rightarrow \exists x: X. x \text{ ismin}_{\prec} S$$

- Definition, *Supporting Induction*:

$$\text{SI}(\prec) \equiv \forall P: \text{pred}_X. \forall (x: X. \forall (y: X_{\prec x}. P y) \Rightarrow P x) \Rightarrow \forall x: X. P x$$

- Equivalence theorem [Gries, Dijkstra, ...]

$$\text{THEOREM, EQUIVALENCE OF WF AND SI: } \text{WF}(\prec) \equiv \text{SI}(\prec)$$

PROOF: next image (in functional predicate calculus)

WF (\prec)
 $\equiv \langle \text{Definition WF} \rangle \text{ and } S \neq \emptyset \equiv \exists x: S. 1$
 $\forall S: \mathcal{P} X. \exists (x: S. 1) \Rightarrow \exists (x: X. x \text{ ismin}_{\prec} S)$
 $\equiv \langle S = X \cap S, \text{ trading} \rangle$
 $\forall S: \mathcal{P} X. \exists (x: X. x \in S) \Rightarrow \exists (x: X. x \text{ ismin}_{\prec} S)$
 $\equiv \langle \text{Definition } \text{ismin} \rangle$
 $\forall S: \mathcal{P} X. \exists (x: X. x \in S) \Rightarrow \exists (x: X. x \in S \wedge \forall y: X. y \prec x \Rightarrow y \notin S)$
 $\equiv \langle p \Rightarrow q \equiv \neg q \Rightarrow \neg p \rangle$
 $\forall S: \mathcal{P} X. \neg (\exists x: X. x \in S \wedge \forall y: X. y \prec x \Rightarrow y \notin S) \Rightarrow \neg (\exists x: X. x \in S)$
 $\equiv \langle \text{Duality } \forall/\exists, \text{ De Morgan} \rangle$
 $\forall S: \mathcal{P} X. \forall (x: X. x \notin S \vee \neg (\forall y: X. y \prec x \Rightarrow y \notin S)) \Rightarrow \forall x: X. x \notin S$
 $\equiv \langle \vee \text{ to } \Rightarrow, \text{ i.e., } a \vee \neg b \equiv b \Rightarrow a \rangle$
 $\forall S: \mathcal{P} X. \forall (x: X. \forall (y: X. y \prec x \Rightarrow y \notin S) \Rightarrow x \notin S) \Rightarrow \forall x: X. x \notin S$
 $\equiv \langle \text{Dummy change using } f: \text{pred}_X \rightarrow \mathcal{P} X \text{ with } x \in f x \equiv \neg (P x) \rangle$
 $\forall P: X \rightarrow B. \forall (x: X. \forall (y: X. y \prec x \Rightarrow P y) \Rightarrow P x) \Rightarrow \forall x: X. P x$
 $\equiv \langle \text{Trading, definition SI} \rangle$
 SI (\prec)

c. Important particular instances of well-founded induction

- Induction over \mathbb{N} (predicates $P : \mathbb{N} \rightarrow \mathbb{B}$) An axiom for natural numbers:

Every nonempty subset of \mathbb{N} has a *minimal* element under $<$.

Equivalently, every nonempty subset of \mathbb{N} has a *least* element under \leq .

Strong induction over \mathbb{N} : define \prec in SI by $m \prec n \equiv m < n$

$$\forall (n : \mathbb{N} . P n) \equiv \forall (n : \mathbb{N} . \forall (m : \mathbb{N} . m < n \Rightarrow P m) \Rightarrow P n)$$

Weak induction over \mathbb{N} : define \prec in SI by $m \prec n \equiv m + 1 = n$

$$\forall (n : \mathbb{N} . P n) \equiv P 0 \wedge \forall (n : \mathbb{N} . P n \Rightarrow P (n + 1)).$$

- Structural induction over lists in A^* (predicates $P : A^* \rightarrow \mathbb{B}$)

List prefix is well-founded and yields

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

Suffices for proving most properties about functional programs with lists.

d. Illustration: proving a property of the Fibonacci numbers

Given $\text{def fib}_- : \mathbb{N} \rightarrow \mathbb{N}$ with $\text{fib}_0 = 0 \wedge \text{fib}_1 = 1 \wedge \text{fib}_{n+2} = \text{fib}_{n+1} + \text{fib}_n$

To prove $\forall m : \mathbb{N}. \forall n : \mathbb{N}. \text{fib}_{m+n+1} = \text{fib}_{m+1} \cdot \text{fib}_{n+1} + \text{fib}_m \cdot \text{fib}_n$ we define

$P : \mathbb{N} \rightarrow \mathbb{B}$ with $Pn \equiv \forall m : \mathbb{N}. \text{fib}_{m+n+1} = \text{fib}_{m+1} \cdot \text{fib}_{n+1} + \text{fib}_m \cdot \text{fib}_n$

and prove $\forall P$ by induction, i.e., $P0 \wedge \forall (n : \mathbb{N}). Pn \Rightarrow P(n+1)$.

(0) Proving $P0$, i.e., $\forall m : \mathbb{N}. \text{fib}_{m+1} = \text{fib}_{m+1} \cdot \text{fib}_1 + \text{fib}_m \cdot \text{fib}_0$ is trivial.

(1) Proving $\forall (n : \mathbb{N}). Pn \Rightarrow P(n+1)$: for given n , we assume Pn (IH) and

prove $P(n+1)$, i.e., $\forall m : \mathbb{N}. \text{fib}_{m+(n+1)+1} = \text{fib}_{m+1} \cdot \text{fib}_{(n+1)+1} + \text{fib}_m \cdot \text{fib}_{n+1}$

as follows: for arbitrary $m : \mathbb{N}$, we calculate $\text{fib}_{m+(n+1)+1}$.

$$\begin{aligned}
 \text{fib}_{m+(n+1)+1} &= \langle \text{Assoc. } + \rangle \text{fib}_{(m+1)+n+1} \\
 &= \langle \text{Instant. IH} \rangle \text{fib}_{m+2} \cdot \text{fib}_{n+1} + \text{fib}_{m+1} \cdot \text{fib}_n \\
 &= \langle \text{Def. fib} \rangle (\text{fib}_{m+1} + \text{fib}_m) \cdot \text{fib}_{n+1} + \text{fib}_{m+1} \cdot \text{fib}_n \\
 &= \langle \text{Arithmetic} \rangle \text{fib}_{m+1} \cdot (\text{fib}_{n+1} + \text{fib}_n) + \text{fib}_m \cdot \text{fib}_{n+1} \\
 &= \langle \text{Def. fib} \rangle \text{fib}_{m+1} \cdot \text{fib}_{n+2} + \text{fib}_m \cdot \text{fib}_{n+1}
 \end{aligned}$$

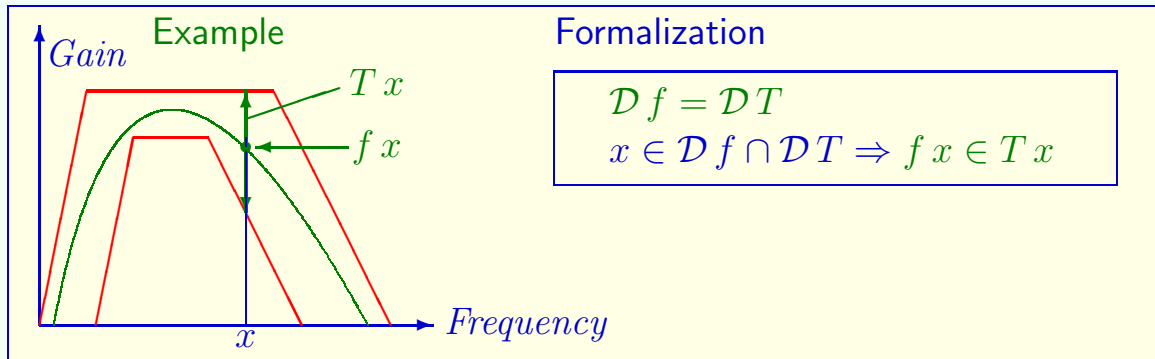
3 Illustrations A: classical engineering & continuous math

3.0 The Generalized Functional Cartesian Product: functional tolerance

- Tolerances for scalars: used routinely for all classical engineering artefacts
- 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



The Generalized Functional Cartesian product (continued)

a. *Defining the FunCart operator* \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:	$A \times B = \times(A, B)$ (for any sets A and B)
Function type:	$A \rightarrow B = \times(A \bullet B)$ (idem)
Point-free form	$\times T = \{f: \mathcal{D} T \rightarrow \bigcup T \mid \forall (f \widehat{\in} T)\}$
Explicit inverse	$\times^{-1} S = x: \bigcup (f: S. \mathcal{D} f). \{f x \mid f: S\}$
Function equality:	$f = g \equiv f \in \times(\iota \circ g)$
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: $A \ni a \rightarrow B_a \ni b \rightarrow C_{a,b}$

3.1 Analysis: calculation replacing syncopation — first example

Basic topology: adherence, open and closed “sets” (here: predicates; more elegant!)

def $\text{ad} : (\mathbb{R} \rightarrow \mathbb{B}) \rightarrow (\mathbb{R} \rightarrow \mathbb{B})$ **with** $\text{ad } P v \equiv \forall \epsilon : \mathbb{R}_{>0} . \exists x : \mathbb{R}_P . |x - v| < \epsilon$
def $\text{open} : (\mathbb{R} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$ **with**
 $\text{open } P \equiv \forall v : \mathbb{R}_P . \exists \epsilon : \mathbb{R}_{>0} . \forall x : \mathbb{R} . |x - v| < \epsilon \Rightarrow P x$
def $\text{closed} : (\mathbb{R} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$ **with** $\text{closed } P \equiv \text{open } (\neg P)$

Example: proving the *closure property* $\boxed{\text{closed } P \equiv \text{ad } P = P}$.

$\text{closed } P$

\equiv $\langle \text{Definit. closed} \rangle \text{ open } (\neg P)$
 \equiv $\langle \text{Definit. open} \rangle \forall v : \mathbb{R}_{\neg P} . \exists \epsilon : \mathbb{R}_{>0} . \forall x : \mathbb{R} . |x - v| < \epsilon \Rightarrow \neg P x$
 \equiv $\langle \text{Trading sub } \forall \rangle \forall v : \mathbb{R} . \neg P v \Rightarrow \exists \epsilon : \mathbb{R}_{>0} . \forall x : \mathbb{R} . |x - v| < \epsilon \Rightarrow \neg P x$
 \equiv $\langle \text{Contraposit.} \rangle \forall v : \mathbb{R} . \neg \exists (\epsilon : \mathbb{R}_{>0} . \forall x : \mathbb{R} . P x \Rightarrow \neg (|x - v| < \epsilon)) \Rightarrow P v$
 \equiv $\langle \text{Duality, twice} \rangle \forall v : \mathbb{R} . \forall (\epsilon : \mathbb{R}_{>0} . \exists x : \mathbb{R} . P x \wedge |x - v| < \epsilon) \Rightarrow P v$
 \equiv $\langle \text{Definition ad} \rangle \forall v : \mathbb{R} . \text{ad } P v \Rightarrow P v$
 \equiv $\langle P v \Rightarrow \text{ad } P v \rangle \forall v : \mathbb{R} . \text{ad } P v \equiv P v$ (proving $P v \Rightarrow \text{ad } P v$ is simple)

Analysis: calculation replacing syncopation — second example

Define $L \text{ islim}_f a \equiv \forall \epsilon: \mathbb{R}_{>0}. \exists \delta: \mathbb{R}_{>0}. \forall x: \mathcal{D} f. |x - a| < \delta \Rightarrow |f x - L| < \epsilon$.

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 $\forall x: S. |x - a| < \delta \Rightarrow |f x - b| < \epsilon$,

$L \text{ islim}_f a \wedge M \text{ islim}_{f \upharpoonright_S} a$

$\Rightarrow \langle \text{Hint in prf. (i)} \rangle L \text{ islim}_{f \upharpoonright_S} a \wedge M \text{ islim}_{f \upharpoonright_S} a$

$\equiv \langle \text{Def. islim, hyp.} \rangle \forall (\epsilon: \mathbb{R}_{>0}. \exists \delta: \mathbb{R}_{>0}. L R \delta) \wedge \forall (\epsilon: \mathbb{R}_{>0}. \exists \delta: \mathbb{R}_{>0}. M R \delta)$

$\equiv \langle \text{Distribut. } \forall / \wedge \rangle \forall \epsilon: \mathbb{R}_{>0}. \exists (\delta: \mathbb{R}_{>0}. L R \delta) \wedge \exists (\delta: \mathbb{R}_{>0}. M R \delta)$

$\equiv \langle \text{Distribut. } \wedge / \exists \rangle \forall \epsilon: \mathbb{R}_{>0}. \exists \delta: \mathbb{R}_{>0}. \exists \delta': \mathbb{R}_{>0}. L R \delta \wedge M R \delta'$

$\Rightarrow \langle \text{Closeness lem.} \rangle \forall \epsilon: \mathbb{R}_{>0}. \exists \delta: \mathbb{R}_{>0}. \exists \delta': \mathbb{R}_{>0}. a \in \text{Ad } S \Rightarrow |L - M| < 2 \cdot \epsilon$

$\equiv \langle \text{Hyp. } a \in \text{Ad } S \rangle \forall \epsilon: \mathbb{R}_{>0}. \exists \delta: \mathbb{R}_{>0}. \exists \delta': \mathbb{R}_{>0}. |L - M| < 2 \cdot \epsilon$

$\equiv \langle \text{Const. pred. } \exists \rangle \forall \epsilon: \mathbb{R}_{>0}. |L - M| < 2 \cdot \epsilon$

$\equiv \langle \text{Vanishing lem.} \rangle L - M = 0$

$\equiv \langle \text{Leibniz, inv. } + \rangle L = M$

3.2 Computational use of functionals in systems theory

- a. Need for serious cleanups Defects in common conventions listed (in 2000) by

E. Lee, P. Varaiya, *Introducing signals and systems: the Berkeley approach*

<http://ptolemy.eecs.berkeley.edu/publications/papers/00/spe1/spe1.pdf>

- Using the function argument (in attempting to) characterize the domain
Example: $x(n) = x(nT)$ to express sampling
- Expressions, functions and function applications “systematically” confused
Examples: $y(t) = T(x(t))$ for systems behavior
 $y(t) = x(t) * h(t)$ for convolution
- Consequences: no formal calculation rules, erroneous instantiation
Example: $y(t - \tau) = x(t - \tau) * h(t - \tau)$ (wrong from any viewpoint)
- We also add: wrong variable bindings (useless for calculation)
Example: $\mathcal{F}\{f(t)\} = \int_{-\infty}^{+\infty} f(t) \cdot e^{j \cdot \omega \cdot t} \cdot dt$ for the Fourier-transform

Note: in Funmath, all those defects were already eliminated in 1990

b. Using functionals in reasoning about properties of systems

- Define $\mathcal{S}_A = \mathbb{T} \rightarrow A$ for value space A and time domain \mathbb{T} .
A *signal* is a function of type \mathcal{S}_A and a *system* is of type $\mathcal{S}_A \rightarrow \mathcal{S}_A$.
- Typical possible characteristics for systems $s : \mathcal{S}_A \rightarrow \mathcal{S}_A$.
 - For additive \mathbb{T} , define $\sigma_- : \mathbb{T} \rightarrow \mathcal{S}_A \rightarrow \mathcal{S}_A$ with $\sigma_\tau x t = x(t + \tau)$.
Then system s is *time-invariant* iff $\forall \tau : \mathbb{T} . s \circ \sigma_\tau = \sigma_\tau \circ s$
 - Let $A = \mathbb{C}$; then s is *linear* iff $\forall z : \mathcal{S}_\mathbb{C} . \forall c : \mathbb{C} . s(c \cdot z) = c \cdot s z$
- Example: the response of an LTI systems to the parametrized exponential $E_- : \mathbb{C} \rightarrow \mathbb{T} \rightarrow \mathbb{C}$ with $E_c t = e^{c \cdot t}$ is $s E_c = s E_c 0 \cdot E_c$. Proof: calculate

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

Substituting $t := 0$ yields $s E_c \tau = s E_c 0 \cdot E_c \tau$.

c. Illustrating more advantages of proper conventions for functionals

- Example: avoiding $\mathcal{F}\{f(t)\}$ and writing $\mathcal{F}f\omega$ instead

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

Clear and unambiguous bindings allow formal calculation.

- Advantage: formalizing Laplace transforms via Fourier transforms.

Auxiliary function: $\ell_{\sigma} : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$ with $\ell_{\sigma} t = (t < 0) ? 0 \mid e^{-\sigma\cdot t}$

We define the Laplace-transform $\mathcal{L}f$ of a function f by:

$$\mathcal{L}f(\sigma + j\cdot\omega) = \mathcal{F}(\ell_{\sigma} \hat{\cdot} f)\omega$$

for real σ and ω , with σ such that $\ell_{\sigma} \hat{\cdot} f$ has a Fourier transform.

With $s := \sigma + j\cdot\omega$ we obtain (exercise)

$$\mathcal{L}f s = \int_0^{+\infty} f t \cdot e^{-s\cdot t} \cdot dt .$$

- Calculation example: the inverse Laplace transform

Specification of \mathcal{L}' : $\mathcal{L}'(\mathcal{L} f) t = f t$ for all $t \geq 0$

(weakened where $\ell_\sigma \hat{f}$ is discontinuous).

Calculation of an explicit expression: For t as specified,

$$\begin{aligned}
 \mathcal{L}'(\mathcal{L} f) t &= \langle \text{Specification} \rangle f t \\
 &= \langle a = 1 \cdot a \rangle e^{\sigma \cdot t} \cdot \ell_\sigma t \cdot f t \\
 &= \langle \text{Definition } \hat{\ } \rangle e^{\sigma \cdot t} \cdot (\ell_\sigma \hat{f}) t \\
 &= \langle \text{Weakened} \rangle e^{\sigma \cdot t} \cdot \mathcal{F}'(\mathcal{F}(\ell_\sigma \hat{f})) t \\
 &= \langle \text{Definition } \mathcal{F}' \rangle e^{\sigma \cdot t} \cdot \frac{1}{2 \cdot \pi} \cdot \int_{-\infty}^{+\infty} \mathcal{F}(\ell_\sigma \hat{f}) \omega \cdot e^{j \cdot \omega \cdot t} \cdot d\omega \\
 &= \langle \text{Definition } \mathcal{L} \rangle e^{\sigma \cdot t} \cdot \frac{1}{2 \cdot \pi} \cdot \int_{-\infty}^{+\infty} \mathcal{L} f(\sigma + j \cdot \omega) \cdot e^{j \cdot \omega \cdot t} \cdot d\omega \\
 &= \langle \text{Const. factor} \rangle \frac{1}{2 \cdot \pi} \cdot \int_{-\infty}^{+\infty} \mathcal{L} f(\sigma + j \cdot \omega) \cdot e^{(\sigma + j \cdot \omega) \cdot t} \cdot d\omega \\
 &= \langle s := \sigma + j \cdot \omega \rangle \frac{1}{2 \cdot \pi \cdot j} \cdot \int_{\sigma - j \cdot \infty}^{\sigma + j \cdot \infty} \mathcal{L} f s \cdot e^{s \cdot t} \cdot ds
 \end{aligned}$$

3.3 Overloading and polymorphism

a. Terminology (0) and main issues (1)

- (0) Overloading: same identifier designating “different” objects (functions).
Polymorphism: different argument types, formally same image definition.
- (1) Disambiguation: via argument type. Means: compatibility (©)
Refined typing: link argument/result type. Means: proper operator

b. Kinds of overloading/polymorphism

- By explicit parametrization Trivial with \times .

Example: *binary addition* function adding two binary words of equal length.

```
def binadd_ :  $\times n : \mathbb{N} . (\mathbb{B}^n)^2 \rightarrow \mathbb{B}^{n+1}$  with binaddn (x, y) = ...
```

- Without auxiliary parameter (to be designed next)

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) = ...
```

c. An interesting design satisfying the requirement

- Principle (explained via the example)

`binadd` as a *merge* of functions of type $(\mathbb{B}^n)^2 \rightarrow \mathbb{B}^{n+1}$ (various n).

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

Requirement: *compatibility* of merged functions (merging selectively)

- Generic functional: for 2 arbitrary function types (function sets) F, G :

$$F \otimes G = \{f \cup g \mid f, g : F \times G \wedge f \odot g\}$$

- Elastic extension to arbitrary function type families:

$$\mathbf{def} \otimes : \mathbf{Fam}(\mathcal{P} \mathcal{F}) \rightarrow \mathcal{P} \mathcal{F} \mathbf{ with } \otimes T = \{\cup f \mid f : (\times T) \odot\}$$

- Applications for other purposes than polymorphism shown later.

4 Illustrations B: computer engineering & discrete math

4.0 Formalizing aggregate data types

a. The funcart operator \times as the “workhorse” for function typing

- Recall $A \rightarrow B = \times(A \bullet B)$ and $A \times B = \times(A, B)$
- Array types: for set A and $n: \mathbb{N} \cup \iota\infty$, define

$$A \uparrow n = \square n \rightarrow A$$

General shorthand: a^b for $a \uparrow b$.

Note: A^n is the n -fold Cartesian product, since $A \uparrow n = \times(\square n \bullet A)$

- Stream types for infinite sequences: simply A^∞ Note: $A^\infty = \mathbb{N} \rightarrow A$.
- List types for finite sequences

$$A^* = \bigcup_{n: \mathbb{N}} A^n$$

- Sequence types for any sequences: $A^\omega = A^* \cup A^\infty$

b. Pascal-like records (ubiquitous in programs) How making them functional?

- Well-known approach: selector function for each field label.
(e.g., Haskell)

Problem: records themselves are 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:

$$\text{Record} : \text{Fam}(\text{Fam}\mathcal{T}) \rightarrow \mathcal{P}\mathcal{F} \quad \text{with} \quad \text{Record } F = \times (\cup F)$$

Now we can write

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

4.1 Relational databases in functional style

- a. 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
...	

- b. 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 Record $F = \times (\cup F)$
Advantage: embedding in general framework, inheriting algebraic properties and generic operators.

- c. Relational databases as sets of functions using 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
...	

is declared as $GCI : \mathcal{P} CID$, a set of *Course Information Descriptors* with

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

- d. Access to a database: done by suitably formulated *queries*, such as
- Who is the instructor for CS300?
 - At what time is K. Jason normally teaching a course?
 - 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.

e. Formalizing queries

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

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

Our generic functionals provide this functionality. Convention: record type R .

- **Selection (σ)** 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 for question (a), “Who is the instructor for CS300?”.

- **Projection (π)** yields in any $S : \mathcal{P} R$ columns with field names in a set F .

Solution: restriction $\pi(S, F) = \{r \upharpoonright F \mid r : S\}$.

Example: $\pi(GCI, \{\text{code}, \text{inst}\})$ selects the columns for question (a)

whereas $\pi(GCI \downarrow (r : CID . r \text{ code} = \text{CS300}), \iota \text{ inst})$ reflects all of (a).

- **Join (\bowtie)** combines tables S, 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., as 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 remark Note that $S \bowtie T = S \otimes T$ We can show

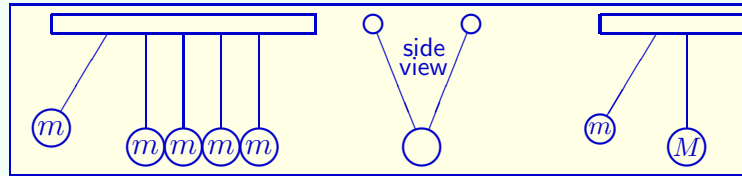
$$\odot (f, g, h) \Rightarrow (f \cup g) \cup h = f \cup (g \cup h)$$

Hence, although \cup is *not* associative, \otimes (and hence \bowtie) is associative.

Importance: this is all about generic functionals, no ad hoc database theories!

4.2 Computational semantics unifying theories of programming

a. An analogy: colliding balls ("Newton's Cradle")



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

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

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

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

Crucial point: mathematics is not used as just a "compact language" (layman's view); rather: the calculations yield insights that are hard to obtain by intuition.

b. 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	Behavior (program equations or equivalent program)	
	State change $R_c(s, s')$	Termination $T_c s$
$v := e$	$s' = s[e^v]$	1
skip	$s' = s$	1
abort	0	0
$c'; c''$	$\exists t. R_c(s, t) \wedge R_{c''}(t, s')$	$T_{c'} s \wedge \forall t. R_{c'}(s, t) \Rightarrow T_{c''} t$
if $\parallel i : I. b_i \rightarrow c'_i$ fi	$\exists i : I. b_i \wedge R_{c'_i}(s, s')$	$\exists b \wedge \forall i : I. b_i \Rightarrow T_{c'_i} s$
do $b \rightarrow c'$ od	if $\neg b \rightarrow \text{skip} \parallel b \rightarrow (c'; c)$ fi	

Abbreviation: $(s \bullet e) = s : \mathbb{S}. e$. Note: \mathbb{S} is the program state space.

c. Program theories expressed via the equations (no “special logics”)

Example: ante/post semantics (Hoare style) with predicates in $\mathbb{S} \rightarrow \mathbb{B}$

$\{A\} c \{P\}$	$\equiv \forall (s, s'). (\mathbb{S}^2 \downarrow R_c). A \ s \Rightarrow P \ s'$	“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”

d. Calculate all properties of interest *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 antecondx: push A out

$$\begin{aligned}
 & [A] c [P] \\
 \equiv & \langle \text{Def. } [] [] \rangle \{A\} c \{P\} \wedge \text{Term}_c A \\
 \equiv & \langle \text{Def. } \{ \} \{ \} \rangle \forall (s. \forall s'. A s \wedge R_c(s, s') \Rightarrow P s') \wedge \text{Term}_c A \\
 \equiv & \langle \text{Df. Term}_c A \rangle \forall (s. \forall s'. A s \wedge R_c(s, s') \Rightarrow P s') \wedge \forall (s. A \Rightarrow T_c s) \\
 \equiv & \langle \text{Distr. } \forall / \wedge \rangle \forall s. \forall (s'. A s \wedge R_c(s, s') \Rightarrow P s') \wedge (A s \Rightarrow T_c s) \\
 \equiv & \langle \text{Shunt } \wedge / \Rightarrow \rangle \forall s. \forall (s'. A s \Rightarrow R_c(s, s') \Rightarrow P s') \wedge (A s \Rightarrow T_c s) \\
 \equiv & \langle \text{Ldist. } \Rightarrow / \forall \rangle \forall s. (A s \Rightarrow \forall s'. R_c(s, s') \Rightarrow P s') \wedge (A s \Rightarrow T_c s) \\
 \equiv & \langle \text{Ldist. } \Rightarrow / \wedge \rangle \forall s. A s \Rightarrow \forall (s'. R_c(s, s') \Rightarrow P s') \wedge T_c s
 \end{aligned}$$

So $[A] c [P] \equiv \forall s. A s \Rightarrow \forall (s'. R_c(s, s') \Rightarrow P s') \wedge T_c s$. Hence define

$$\begin{aligned}
 & \text{def } \text{wla} : C \rightarrow \text{pred}_S \rightarrow \text{pred}_S \text{ with } \text{wla } c P s \equiv \forall s'. R_c(s, s') \Rightarrow P s' \\
 & \text{def } \text{wa} : C \rightarrow \text{pred}_S \rightarrow \text{pred}_S \text{ with } \text{wa } c P s \equiv \text{wla } c P s \wedge T_c s
 \end{aligned}$$

e. Results and more analogies

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

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

Syntactic shortcut used: $s = \text{tuple of all program variables.}$

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

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

4.3 Formal reasoning in automata theory (using functionals)

We consider systems $s : A^* \rightarrow B^*$

- a. **Sequentiality** Define \leq on A^* (or B^* etc.) by $x \leq y \equiv \exists z : A^* . y = x ++ z$.

System s is *non-anticipatory* or **sequential** iff $x \leq y \Rightarrow s x \leq s y$

Function $r : (A^*)^2 \rightarrow B^*$ is a **residual behavior** of s iff $s(x ++ y) = s x ++ r(x, y)$

THEOREM: s is sequential iff it has a residual behavior function.

Proof: we start from the sequentiality side.

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

We used the *function comprehension* axiom: for any relation $R : X \times Y \rightarrow \mathbb{B}$,

$$\forall (x : X . \exists y : Y . R(x, y)) \equiv \exists f : X \rightarrow Y . \forall x : X . R(x, f x)$$

b. **Derivatives and primitives** The preceding framework leads to the following.

- Observation: An rb function is unique (exercise).
- We define the *derivation* operator D on sequential systems by

$$D s \varepsilon = \varepsilon \quad \text{and} \quad D s (x \prec a) = s x ++ D s (x \prec a)$$

With the rb function r of s , $D s (x \prec a) = r (x, \tau a)$.

- *Primitivation* I is defined for any $g: A^* \rightarrow B^*$ by

$$I g \varepsilon = \varepsilon \quad \text{and} \quad I g (x \prec a) = I g x ++ g (x ++ a)$$

- Properties (note a striking analogy from analysis)

$$\begin{array}{l|l} s (x \prec a) = s x ++ D s (x \prec a) & s x = s \varepsilon ++ I (D s) x \\ f (x + h) \approx f x + D f x \cdot h & f x = f 0 + I (D f) x \end{array}$$

In the second row, D is derivation as in analysis, and $I g x = \int_0^x g y \cdot dy$.

- The *state space* is $\{y: A^* \cdot r (x, y) \mid x: A^*\}$.

4.4 Getting things right in discrete mathematics

- a. Errors in mathematical software (more in “discrete” than in “continuous”)

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

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

Cause of errors: \sum *not* well-understood, formal rules rare (\rightarrow syncopation)

- b. Getting things right by proper formalization and calculation

- Proper formal definition (Funmath): for any a , any number c and any number-valued functions f and g with finite nonintersecting domains:

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

Extension to infinite (but ordered) domains by the usual limit construction.

Classical notation $\sum_{i=m}^n f_i$ defined as shorthand for $\sum i : m .. n . f_i$.

- Formal calculation (with *trading* $\sum f_P = \sum (P \hat{\cdot} f)$ as the star) yields

$$\sum_{i=1}^n \sum_{j=i}^m 1 = (k \geq 1) ? \frac{k \cdot (2 \cdot m - k + 1)}{2} \dagger 0 \quad \text{where } k := m \wedge n$$

5 **Final considerations**

- 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
 - Convenient for “pencil and paper” use (mature for being automated)
- Ramifications
 - Scientific: obvious
 - Professional: proper basis (prerequisite?) for use of software tools
Role for “logic” tools similar to role of calculus for Maple etc.
Problem: some find logic hard (cause: de-emphasis on proofs in education)
Yet: here made easier and more general than in other logic formalisms
- Conclusion: long-term advantages outweigh transitory “mathphobic” trends.