

Formal Calculation as a Tool for Discovery

Overview

0. Motivation: dichotomies and opportunities: UT FACIANT OPUS SIGNA
1. The formalism, part A: a simple and problem-free language
2. The formalism, part B: clear and problem-free formal rules
3. Illustrations in classical mathematics (analysis, discrete math)
Crucial lesson: handling equality consistently correctly (à la Leibniz)
4. Illustrations in mathematics related to Computing Science
 - Discovering instead of postulating theories (example: program semantics)
 - Reverse engineering and unification of theories (example: various calculi)
 - Improving formulation / discovering stronger results (e.g., temporal logic)
5. Conclusion — Unifying classical and computing-related math (EE and CS)

WARNING

Most formal calculations shown in this presentation

- Use concepts and rules that are gradually introduced during a full course
— and hence cannot be properly covered in 50 minutes
- Are meant to convey the LOOK and FEEL of the calculations
— like showing a printed circuit board in a talk on hardware

Hence they are best viewed accordingly:

- Concentrate on the LOOK and FEEL of the calculations
- Refer to the referenced papers and tutorials for the details

Next topic

0. Motivation: dichotomies and opportunities

- Dichotomy: a traditional rift between classical and computer engineering
- Main cause: style breach between well and poorly formalized mathematics
- Opportunities: formal expression & formal calculation as tools for discovery

UT FACIANT OPUS SIGNA

“Letting the symbols do the work”: formalization as a boon, not a burden

1. The formalism, part A: a simple and problem-free language
2. The formalism, part B: clear and problem-free formal rules
3. Illustrations in classical mathematics (analysis, discrete math)
4. Illustrations in mathematics related to Computing Science
5. Conclusion — Unifying classical and computing-related math (EE and CS)

0 Introduction: dichotomies and opportunities

0.0 Dichotomy: a traditional rift between classical and computer engineering

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")

a. 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"

b. 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 (doubtful intellectual content), 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

- a. 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

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

Reminder: concentrate on the look and feel of the calculations, not the details.

- b. 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.” [amen!]

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

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

0.2 Opportunities: proper formal expression and calculation

- a. 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*
- b. 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!

c. Attitude of typical mathematicians regarding formality is very ambivalent

- Applaud it and exploit it “shamelessly” in areas where it is well-designed
⇒ good for them!
- Prejudicial in areas where formality is poorly developed:
 - Very apologetic/tolerant of ill-conceived notations and conventions
 - Blame the results of these deficiencies on the principle of formality⇒ shame on them!

Important observations

- Abuse of notation is never innocuous, but always reflects conceptual flaws
- Poor notations/conventions hamper guidance by the shape of expressions

Proper attitude: “Test all things, hold fast [to] what is good” (who’s quoted?)

d. “All that remains” is showing the power of formalization made right and simple

Yet: making things simple required considerable thinking and effort.

Next topic

0. Motivation: dichotomies and opportunities: UT FACIANT OPUS SIGNA
1. The formalism, part A: a simple and problem-free language
 - Rationale: the need for defect-free notation
 - Funmath language design
2. The formalism, part B: clear and problem-free formal rules
3. Illustrations in classical mathematics (analysis, discrete math)
4. Illustrations in mathematics related to Computing Science
5. Conclusion — Unifying classical and computing-related math (EE and CS)

1 The formalism, part A: a simple and problem-free language

1.0 Rationale: the need for defect-free notation

Examples of defects in mathematical conventions

Examples A: defects in often-used conventions relevant to systems theory

- **Ellipsis**, i.e., dots (...) as in $a_0 + a_1 + \dots + a_n$, e.g., $0^2 + 1^2 + \dots + n^2$
Common use violates Leibniz's principle (substitution of equals for equals)
Example: **Substituting** $n = 7$ yields $0 + 1 + \dots + 49$ (probably not intended!)
- **Summation sign** \sum not as well-understood as often assumed.
Example: error in *Mathematica*: $\sum_{i=1}^n \sum_{j=i}^m 1 = \frac{n \cdot (2 \cdot m - n + 1)}{2}$
Substituting $n := 3$ and $m := 1$ yields $1 = 0$.
- **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 (“set expressions”, “ZF”)

- Patterns typical in mathematical writing:
assuming boolean (or “logical”) expression p , arbitrary expression e ,

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

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

- Worse: such defects *prohibit even the formulation of calculation rules!*
Formal calculation with “set expressions” rare/nonexistent in the literature.

Underlying cause: overloading relational operator \in for the binding of a dummy.
This poor convention is ubiquitous (not only for sets), as in $\forall x \in \mathbb{R}. x^2 \geq 0$.

1.1 Funmath language design

Basis: *function* (= *domain* + *mapping*)

Language syntax : 4 constructs: identifier, application, abstraction, tupling

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: $f x$ 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: $b.e$ 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 for 1 dimension: $\boxed{e, e', e''}$ (any length)

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 $(m + n)$.
- Empty tuple: ε . For singleton tuples, we define τ by $\tau e = 0 \mapsto e$. Legend:
 - ε is defined by $\varepsilon := \emptyset \bullet e$ (any e) for the *empty function*;
 - \mapsto is defined by $d \mapsto e = \iota d \bullet e$ for *one-point functions*.

Final remarks (hereby we conclude the syntactic technicalities)

- This design suffices for essentially ALL engineering mathematics notation!
- It supports two styles:
 - “Conservative”: resembling “standard” notation (yet “safe”)
 - “Full language”: requiring the above explanation or at least some alertness

Next topic

0. Motivation: dichotomies and opportunities: UT FACIANT OPUS SIGNA

1. The formalism, part A: a simple and problem-free language

2. The formalism, part B: clear and problem-free formal rules

- General style for calculational and equational reasoning
- Preparatory: rules for proposition logic and sets
- Central: functions and generic functionals (“concrete category theory”)
- Central: functional predicate calculus and quantifier

3. Illustrations in classical mathematics (analysis, discrete math)

4. Illustrations in mathematics related to Computing Science

5. Conclusion — Unifying classical and computing-related math (EE and CS)

2 The formalism, part B: clear and problem-free formal rules

2.0 General style for calculational and equational reasoning

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

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

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

- b. **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,

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

- c. **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 Preparatory: rules for calculating with propositions and sets

- a. **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 $e = e' \Rightarrow d[e^v] = d[e'^v]$.
- b. **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 $X = Y$.

2.2 Central: functions and generic functionals (“concrete category theory”)

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

(bootstrap for predicate calculus, which in turn yields elegant formulation)

- Abstraction encapsulates substitution. Formal axioms:

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

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

Note: mapping axiom is beta conversion as in lambda calculus.

Equality is characterized via function equality (exercise).

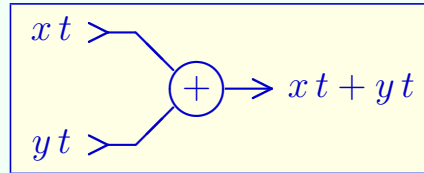
b. Generic functionals

- Concrete 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 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 (i), goal (ii) and new designs (discussed next).

- **IMPORTANT STYLE GOAL:** support point-wise and point-free mathematical expression and smooth transition between both styles

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

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

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

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 ,

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

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

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

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

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

- *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* (\upharpoonright): $f \upharpoonright X = f \downarrow (X \bullet 1)$.

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 merge* (\cup) is defined here in 2 parts to fit within the page width:

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

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

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

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

2.3 Central: functional predicate calculus and quantifiers

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.

a. Axioms and forms of expression

- Basic axioms: *quantifiers* (\forall, \exists) are *predicates on predicates* defined by

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

(recall the definition: a *predicate* is a $\{0, 1\}$ -valued function)

- 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** (a few typical examples; the “full package” is quite extensive)

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

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

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)-distr. \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)-distr. \wedge/\forall generalizes $q \wedge (r \wedge s) \equiv (q \wedge r) \wedge (q \wedge s)$

c. Derived rules (continued)

Some typical 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. Using this predicate calculus to complete the rule package for function(al)s

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

- **Application: set comprehension (completing the rule package for sets)**

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

Consequence: defect-free “set notation” (ZF-style):

- Expressions like $\{2, 3, 5\}$ and $\{2 \cdot m \mid m : \mathbb{Z}\}$: 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).

Note (“conservative” style): for an operator f , write $\mathcal{R} f$ rather than $\{f\}$

Next topic

0. Motivation: dichotomies and opportunities: UT FACIANT OPUS SIGNA

1. The formalism, part A: a simple and problem-free language

2. The formalism, part B: clear and problem-free formal rules

3. Illustrations in classical mathematics (analysis, discrete math)

- Analysis examples: calculation replacing syncopation; transform methods
- Discrete mathematics example: properly designing the \sum -operator
- Crucial lesson: handling equality consistently correctly (à la Leibniz)

4. Illustrations in mathematics related to Computing Science

5. Conclusion — Unifying classical and computing-related math (EE and CS)

3 Illustrations in classical mathematics (analysis, discrete math)

3.0 Analysis example: calculation replacing syncopation

```
def ad : (ℝ → ℬ) → (ℝ → ℬ) with ad P v ≡ ∀ ε : ℝ>0. ∃ x : ℝP. |x - v| < ε
def open : (ℝ → ℬ) → ℬ with
  open P ≡ ∀ v : ℝP. ∃ ε : ℝ>0. ∀ x : ℝ. |x - v| < ε ⇒ P x
def closed : (ℝ → ℬ) → ℬ with closed P ≡ open (¬ P)
```

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

closed P

```
≡ ⟨Definit. closed⟩ open (¬ P)
≡ ⟨Definit. open⟩ ∀ v : ℝ¬P. ∃ ε : ℝ>0. ∀ x : ℝ. |x - v| < ε ⇒ ¬ P x
≡ ⟨Trading sub ∀⟩ ∀ v : ℝ. ¬ P v ⇒ ∃ ε : ℝ>0. ∀ x : ℝ. |x - v| < ε ⇒ ¬ P x
≡ ⟨Contrapositive⟩ ∀ v : ℝ. ¬ ∃ (ε : ℝ>0. ∀ x : ℝ. P x ⇒ ¬ (|x - v| < ε)) ⇒ P v
≡ ⟨Duality, twice⟩ ∀ v : ℝ. ∀ (ε : ℝ>0. ∃ x : ℝ. P x ∧ |x - v| < ε) ⇒ P v
≡ ⟨Definition ad⟩ ∀ v : ℝ. ad P v ⇒ P v
≡ ⟨P v ⇒ ad P v⟩ ∀ v : ℝ. ad P v ≡ P v (proving P v ⇒ ad P v is near-trivial)
```

3.1 Analysis example: transform methods

- a. **Emphasis:** formally correct use of functionals

Avoiding common defective notations like $\mathcal{F}\{f(t)\}$ 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.

- b. **Example:** 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

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

c. Calculation example: the inverse Laplace transform

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

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

Calculation of an explicit expression: For t as specified,

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

Reminder: concentrate on the look and feel of the calculations, not the details.

3.2 Discrete mathematics example: properly designing the \sum -operator

- Recall the Mathematica formula

$$\sum_{i=1}^n \sum_{j=i}^m 1 = \frac{n \cdot (2 \cdot m - n + 1)}{2}$$

Setting $n := 3$ and $m := 1$ yields $1 = 0$. Current version gives just a warning.

- A proper design for \sum : elastic operator with 3 axioms: for any a , any numeric c and any number-valued functions f and g with nonintersecting finite domains,

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

From these basic rules, all rules needed in practice are derived.

E.g., *trading rule* $\sum f_P = \sum (P \hat{\wedge} f)$, similar to $\exists Q_P = \exists (P \hat{\wedge} Q)$.

We define $\sum_{k=m}^n e$ to stand for $\sum k : m .. n . e$ (integer m and n , numeric e).

- Formal calculation (with “trading” in the star role) yields the correct formula

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

3.3 Crucial lesson: handling equality consistently correctly (à la Leibniz)

a. Principle: equality is fundamental, so keep it sacrosanct (no cartoons!)

- Reflexivity, symmetry, transitivity (as any equivalence);

- What distinguishes $\boxed{=}$ is Leibniz's principle, viz., $\boxed{e = e' \Rightarrow d_e^v = d_{e'}^v}$

b. Typical violations

(i) In a typical (otherwise excellent) discrete mathematics text:

Let f and g be functions $\mathbf{N} \rightarrow \mathbf{R}$. We say that f is *Big Oh* of g and write $f = \mathcal{O}(g)$ if there is an integer n_0 and a positive real number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$.

[...]

If f and g are functions $\mathbf{N} \rightarrow \mathbf{R}$, we say that f has *smaller order* than g and write $f \prec g$ if and only if $f = \mathcal{O}(g)$ but $g \neq \mathcal{O}(f)$. If $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$, then we say that f and g have the *same order* and write $f \asymp g$.

(ii) In a typical (otherwise excellent) algebra text:

Let S be a set of numbers and a adherent to S . Let f be a function defined on S . We say that the **limit of $f(x)$ as x approaches a exists**, if there exists a number L having the following property. Given ϵ , there exists a number $\delta > 0$ such that for all $x \in S$ satisfying $|x - a| < \delta$ we have $|f(x) - L| < \epsilon$. If that is the case, we write

$$\lim_{\substack{x \rightarrow a \\ x \in S}} f(x) = L .$$

c. Underlying conceptual flaw in first part of (i) and in (ii):

sloppy confusion between equalities and relations

Typical deficiencies mentioned for (ii):

- Using notation with $=$ to start with presupposes existence and uniqueness
- In such a flawed setting, formal proof of uniqueness is trivial (and wrong)

d. Proper formulations (equally concise, yet free of misleading connotations)

(i) Use a relation: write $L \text{ islim}_f a$, read as “ L is a limit for f at a ”, with

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

General definition discipline in Funmath for such cases (“trilogy principle”)

- define an appropriate relation (definition of the form $y R x \equiv \dots$);
- prove uniqueness, if applicable (i.e., $y R x \wedge y' R x \Rightarrow x = y$);
- define operator (domain defined to characterize existence).

(ii) Use a relation: write $f \text{ is}\mathcal{O} g$ (or $f \preceq g$), read as “ f is *Big Oh* of g ”, with

$$f \text{ is}\mathcal{O} g \equiv \exists c : \mathbb{R}_{>0} . \exists m : \mathbb{N} . \forall n : \mathbb{N}_{\geq m} . |f(n)| \leq c \cdot |g(n)| .$$

Also, define $f \prec g \equiv f \preceq g \wedge \neg (g \preceq f)$ and $f \asymp g \equiv f \preceq g \wedge g \preceq f$.

\Rightarrow Inherit the entire machinery of relation calculus “for free”.

No complicated theories about “one-way equations” needed to get things right

Next topic

0. Motivation: dichotomies and opportunities: UT FACIANT OPUS SIGNA
1. The formalism, part A: a simple and problem-free language
2. The formalism, part B: clear and problem-free formal rules
3. Illustrations in classical mathematics (analysis, discrete math)
4. Illustrations in mathematics related to Computing Science
 - Discovering instead of postulating theories (example: program semantics)
 - Reverse engineering and unification of theories (example: various calculi)
 - Improving formulation / discovering stronger results (e.g., temporal logic)
5. Conclusion — Unifying classical and computing-related math (EE and CS)

4 Illustrations in mathematics related to Computing Science

4.0 From data structures to data bases and query languages **SKIP**

a. Aggregate data types (all aggregates are functions!) Some typical cases¹:

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

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

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

¹All based on our “workhorse for function types”, the Functional Generalized Cartesian Product: for any set-valued function T (called “tolerance function”),

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

b. Relational databases

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

```
def CID := Record (code ↦ Code, name ↦ 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 operators: all the usual ones are subsumed by generic functionals

– The usual *selection*-operator (σ) by $\sigma(S, P) = S \downarrow P$.

– The usual *projection*-operator (π) by $\pi(S, F) = \{r \upharpoonright F \mid r : S\}$.

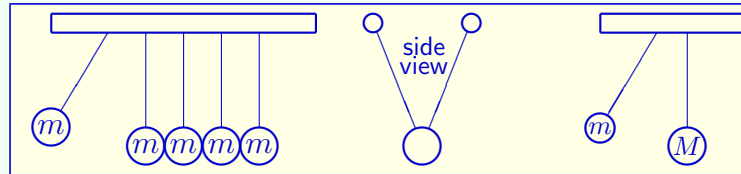
– The usual *join*-operator (\bowtie) by $S \bowtie T = S \otimes T$.

Definition (generic overload operator): $S \otimes T = \{s \cup t \mid (s, t) : S \times T \wedge s \odot t\}$

Observation: \otimes (and hence \bowtie) is associative, although \cup is not.

4.1 Discovering instead of postulating theories (e.g., program semantics)

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



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

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

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

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

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

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 $T_- : C \rightarrow \mathbb{S} \rightarrow \mathbb{B}$.

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

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
$c' ; c''$	$\exists t \cdot R_{c'}(s, t) \wedge R_{c''}(t, s')$	$T_{c'} s \wedge \forall t \cdot R_{c'}(s, t) \Rightarrow T_{c''} t$
if $\square i : I \cdot b_i \rightarrow c'_i$ fi	$\exists i : I \cdot b_i \wedge R_{c'_i}(s, s')$	$\exists b \wedge \forall i : I \cdot b_i \Rightarrow T_{c'_i} s$
do $b \rightarrow c'$ od	if $\neg b \rightarrow$ skip $\square b \rightarrow (c' ; c)$ fi	

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

Example: ante/post semantics (Hoare) but with predicates of type $\mathbb{S} \rightarrow \mathbb{B}$

$\{A\} c \{P\} \equiv \forall (s, s') \cdot (\mathbb{S}^2 \downarrow R_c) \cdot A s \Rightarrow P s'$	"partial correctness"
$[A] c [P] \equiv \{A\} c \{P\} \wedge Term_c A$	"total correctness"
$Term_c A \equiv \forall s \cdot A s \Rightarrow T_c s$	"termination"

d. Discover all properties by calculation *Predicate calculus, no special logics!*

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

– *Weakest liberal antecondition*: weakest A satisfying $\{A\} c \{P\}$ (given P)

– *Weakest antecondition*: weakest A satisfying $[A] c [P]$ (given P)

Calculating an expression for such an antecondition: push A to the left

$$\begin{aligned}
 \{A\} c \{P\} &\equiv \langle \text{Def. } \{A\} c \{P\} \rangle \quad \forall (s, s') : (\mathbb{S}^2 \downarrow R_c) . A s \Rightarrow P s' \\
 &\equiv \langle \text{Trading under } \forall \rangle \quad \forall (s, s') : \mathbb{S}^2 . R_c(s, s') \Rightarrow A s \Rightarrow P s' \\
 &\equiv \langle \text{Nest } \forall; \text{ shunt } \Rightarrow \rangle \quad \forall s . \forall s' . A s \Rightarrow R_c(s, s') \Rightarrow P s' \\
 &\equiv \langle \text{Left distr. } \Rightarrow / \forall \rangle \quad \forall s . A s \Rightarrow \forall s' . R_c(s, s') \Rightarrow P s' \\
 &\equiv \langle \text{Intro. } Wla \text{ below} \rangle \quad \forall s . A s \Rightarrow Wla_c P s \quad (*) \\
 [A] c [P] &\equiv \langle \text{Def. } [A] c [P] \rangle \quad \{A\} c \{P\} \wedge Term_c \\
 &\equiv \langle (*); \text{ def. } Term \rangle \quad \forall (s . A s \Rightarrow Wla_c P s) \wedge \forall (s . A s \Rightarrow T_c s) \\
 &\equiv \langle \text{Distrib. } \wedge / \forall \rangle \quad \forall s . (A s \Rightarrow Wla_c P s) \wedge (A s \Rightarrow T_c s) \\
 &\equiv \langle \text{Left distr. } \Rightarrow / \wedge \rangle \quad \forall s . A s \Rightarrow Wla_c P s \wedge T_c s
 \end{aligned}$$

Uniqueness being easy to show, this justifies defining

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** $Wa_c P s \equiv Wla_c P s \wedge T_c s$

Remark: observe the similarity in proof style with the earlier analysis example

closed P

\equiv $\langle \text{Def. closed} \rangle$ $\text{open}(\neg P)$

\equiv $\langle \text{Def. open} \rangle$ $\forall v: \mathbb{R}_{\neq P}. \exists \epsilon: \mathbb{R}_{>0}. \forall x: \mathbb{R}. |x - v| < \epsilon \Rightarrow \neg P x$

\equiv $\langle \text{Trading } \forall \rangle$ $\forall v: \mathbb{R}. \neg P v \Rightarrow \exists \epsilon: \mathbb{R}_{>0}. \forall x: \mathbb{R}. |x - v| < \epsilon \Rightarrow \neg P x$

\equiv $\langle \text{Contrapos.} \rangle$ $\forall v: \mathbb{R}. \neg \exists (\epsilon: \mathbb{R}_{>0}. \forall x: \mathbb{R}. P x \Rightarrow \neg(|x - v| < \epsilon)) \Rightarrow P v$

\equiv $\langle \text{Dual, } 2\times \rangle$ $\forall v: \mathbb{R}. \forall (\epsilon: \mathbb{R}_{>0}. \exists x: \mathbb{R}. P x \wedge |x - v| < \epsilon) \Rightarrow P v$

\equiv $\langle \text{Defin. ad} \rangle$ $\forall v: \mathbb{R}. \text{ad } P v \Rightarrow P v$

\equiv $\langle \text{Lemma} \rangle$ $\forall v: \mathbb{R}. \text{ad } P v \equiv P v$ (Lemma: $P v \Rightarrow \text{ad } P v$; easy)

Reminder: concentrate on the look and feel of the calculations, not the details.

e. Results and more analogies

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

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

Warning: propositional formulation; s = tuple of all program variables.

- Remark: practical rules for loops (invariants, bound functions) similarly
- Analogies: Green functions (for component c), Fourier transforms

$$\begin{aligned}
 \text{Wla}_c P s &\equiv \forall s': \mathbb{S} . \text{R}_c (s, s') \Rightarrow P s' && \text{(for command } c) \\
 \text{Slp}_c A s &\equiv \exists \backslash s: \mathbb{S} . \text{R}_c (\backslash s, s) \wedge A \backslash s && \text{(for command } c) \\
 \text{Rsp}_c f x &= \mathcal{I} x': \mathbb{R} . \text{G}_c (x, x') \cdot f x' && \text{(for linear component } c) \\
 \text{Rsp}_c f t &= \mathcal{I} t': \mathbb{R} . \text{h}_c (t - t') \cdot f t' && \text{(for LTI component } c) \\
 \mathcal{F} f \omega &= \mathcal{I} t: \mathbb{R} . \exp(-j \cdot \omega \cdot t) \cdot f t
 \end{aligned}$$

4.2 Reverse engineering and unification of theories (e.g., various calculi)

Chosen topic: find equations for Rutger Dijkstra's *Computation Calculus* (CC)

a. Preliminary definitions (assuming given state space \mathbb{S})

- *Computations* are elements of $\mathcal{C} := \mathbb{S}^+ \cup \mathbb{S}^\infty$

Computation predicates are of type $\text{CP} := \mathcal{C} \rightarrow \mathbb{B}$ (specifications, behaviors)

- *Joining* $\text{---} \sqcap \text{---} : \text{CP}^2 \rightarrow \text{CP}$ as usual for predicates: $(P \sqcap Q) x \equiv P x \wedge Q x$

Composition $\text{---}; \text{---} : \text{CP}^2 \rightarrow \text{CP}$ is defined for any $C', C'' : \text{CP}^2$ and $\gamma : \mathcal{C}$ by

$$(C'; C'') \gamma \equiv (\# \gamma = \infty \wedge C' \gamma) \vee \exists n : \mathcal{D} \gamma . C' (\gamma \upharpoonright (n+1)) \wedge C'' (\gamma \downharpoonright n)$$

where $x \upharpoonright n = x_{<n}$ and $x \downharpoonright n = \sigma^n x$ for any $x : A^\omega$ and $n : \square (\# x + 1)$

- **Examples:** (some important computation predicates)

– $\boxed{\text{T} := \mathcal{C} \bullet 1}$ and $\boxed{\text{F} := \mathcal{C} \bullet 0}$ and $\boxed{\mathbf{1} := \gamma : \mathcal{C} . \# \gamma = 1}$ ($\mathbf{1}$ is unit of $;$).

– The *eternity* predicate $\boxed{\text{E} := \text{T}; \text{F}}$ and the *bounded* predicate $\boxed{\text{B} := \neg \text{E}}$.

The properties $\text{E} \gamma \equiv \# \gamma = \infty$ and $\text{B} \gamma \equiv \# \gamma \neq \infty$ justify the names.

b. **States** States are represented as sequences of length 1

- E.g., $\gamma_\alpha := \tau \gamma_0$ (initial state) and $\gamma_\omega := \tau \gamma_{\#\gamma-1}$ if $\#\gamma \neq \infty$ (final state).
- *State predicates* are predicates of type $\text{SP} := \{P : \text{CP} \mid P \sqsubseteq \mathbf{1}\}$.
- **Illustration:** functional predicate calculus shows: for any $P : \text{SP}$ and $C : \text{CP}$,
 - (i) $P ; C = P ; \mathbf{T} \sqcap C$ (iii) $(P ; \mathbf{T})\gamma \equiv P \gamma_\alpha$
 - (ii) $C ; P = C \sqcap \mathbf{T} ; P$ (iv) $(\mathbf{T} ; P)\gamma \equiv \#\gamma \neq \infty \Rightarrow P \gamma_\omega$

c. **Calculation example** Reverse engineering a termination condition

- CC defines Hoare semantics as follows: for any A en P in SP en C in CP,

$$\boxed{\text{(a) } \{A\} C \{P\} \equiv A ; C \sqsubseteq \mathbf{T} ; P \quad \text{(b) } [A] C [P] \equiv A ; C \sqsubseteq \mathbf{B} ; P} \quad (1)$$

- Discovering T such that $[A] C [P] \equiv T \wedge \{A\} C \{P\}$.

$$\begin{aligned} [A] C [P] &\equiv \langle \text{Def. (1.b)} \rangle A ; C \sqsubseteq \mathbf{B} ; P \\ &\equiv \langle \text{Prop. (ii)} \rangle A ; C \sqsubseteq \mathbf{B} \sqcap \mathbf{T} ; P \\ &\equiv \langle \text{Ldist. } \sqsubseteq / \sqcap \rangle A ; C \sqsubseteq \mathbf{B} \wedge A ; C \sqsubseteq \mathbf{T} ; P \\ &\equiv \langle \text{Def. (1.a)} \rangle A ; C \sqsubseteq \mathbf{B} \wedge \{A\} C \{P\} \end{aligned}$$

d. Reverse engineering to find *systems equations*

Goal: abstract variants of program equations capturing CC, i.e.,
calculating $R_- : CP \rightarrow \mathbb{S}^2 \rightarrow \mathbb{B}$ en $T_- : CP \rightarrow \mathbb{S} \rightarrow \mathbb{B}$ such that

$$\begin{aligned} \{A\} C \{P\} &\equiv \forall (s, s') : \mathbb{S}^2 . R_C (s, s') \Rightarrow A(\tau s) \Rightarrow P(\tau s') \\ A; C \sqsubseteq B &\equiv \forall s : \mathbb{S} . A(\tau s) \Rightarrow T_C s \end{aligned}$$

Calculating $T_C s$ (recall: concentrate on the look and feel, not the details):

$$\begin{aligned} A; C \sqsubseteq B &\equiv \langle \text{Prop. (i)} \rangle A; T \sqcap C \sqsubseteq B \\ &\equiv \langle \text{Defin. } \sqsubseteq \rangle \forall \gamma : \mathcal{C} . (A; T \sqcap C) \gamma \Rightarrow B \gamma \\ &\equiv \langle \text{Defin. } \sqcap \rangle \forall \gamma : \mathcal{C} . (A; T) \gamma \wedge C \gamma \Rightarrow B \gamma \\ &\equiv \langle \text{Prop. (iii)} \rangle \forall \gamma : \mathcal{C} . A \gamma_\alpha \wedge C \gamma \Rightarrow B \gamma \\ &\equiv \langle \text{Shunt } \wedge \rangle \forall \gamma : \mathcal{C} . A \gamma_\alpha \Rightarrow C \gamma \Rightarrow B \gamma \\ &\equiv \langle \text{Defin. } \gamma_\alpha \rangle \forall \gamma : \mathcal{C} . A(\tau \gamma_0) \Rightarrow C \gamma \Rightarrow B \gamma \\ &\equiv \langle \text{1-pt. rule} \rangle \forall \gamma : \mathcal{C} . \forall s : \mathbb{S} . s = \gamma_0 \Rightarrow A(\tau s) \Rightarrow C \gamma \Rightarrow B \gamma \\ &\equiv \langle \text{Shunt } \Rightarrow \rangle \forall \gamma : \mathcal{C} . \forall s : \mathbb{S} . A(\tau s) \Rightarrow C \gamma \Rightarrow s = \gamma_0 \Rightarrow B \gamma \\ &\equiv \langle \text{Swap } \forall \rangle \forall s : \mathbb{S} . \forall \gamma : \mathcal{C} . A(\tau s) \Rightarrow C \gamma \Rightarrow s = \gamma_0 \Rightarrow B \gamma \\ &\equiv \langle \text{Ldst. } \Rightarrow / \forall \rangle \forall s : \mathbb{S} . A(\tau s) \Rightarrow \forall \gamma : \mathcal{C} . C \gamma \Rightarrow s = \gamma_0 \Rightarrow B \gamma \\ &\equiv \langle \text{Trading} \rangle \forall s : \mathbb{S} . A(\tau s) \Rightarrow \forall \gamma : \mathcal{C}_C . s = \gamma_0 \Rightarrow B \gamma \end{aligned}$$

Similarly, we find $R_C (s, s') \equiv \exists \gamma : \mathcal{C}_C . \gamma_0 = s \wedge \# \gamma \neq \infty \wedge \gamma_{\# \gamma - 1} = s'$.

4.3 Improving formulation / discovering stronger results

a. Example: temporal operators in TLA⁺ (L. Lamport, *Specifying Systems*)

The definition of *temporal quantifiers* uses an auxiliary function \natural

Purpose: remove duplicates in sequences, e.g., $\natural(4, 4, 5, 5, 5, 3, \dots) = 4, 5, 3, \dots$

- Lamport's definition (for infinite sequences) verbatim (from page 316)

$$\begin{aligned} \natural\sigma &\triangleq \text{LET } f[n \in \text{Nat}] \triangleq \text{IF } n = 0 \text{ THEN } 0 \\ &\quad \text{ELSE IF } \sigma[n] = \sigma[n-1] \\ &\quad \quad \text{THEN } f[n-1] \\ &\quad \quad \text{ELSE } f[n-1] + 1 \\ S &\triangleq \{f[n] : n \in \text{Nat}\} \\ \text{IN } [n \in S &\mapsto \sigma[\text{CHOOSE } i \in \text{Nat} : f[i] = n]] \end{aligned}$$

- Funmath definition (for finite and infinite sequences) with function merge \cup

$$\text{def } \natural : \mathbf{S}^\omega \rightarrow \mathbf{S}^\omega \text{ with } \natural\beta = \cup n : \mathcal{D}\beta . \sum (k : \square n . \beta(k+1) \neq \beta k) \mapsto \beta n$$

Proving equivalence (for infinite sequences) is a typical exam question (with hints).

b. Example: smoother proofs discovering stronger results

Context: temporal formulas difficult to design/interpret. Hence: use of *patterns*.
Typical pattern from Lamport's *Specifying Systems*: expression for *weak fairness*

$$\text{WF}_v(A) \equiv \Box(\Box(\text{ENABLED } \langle A \rangle_v) \Rightarrow \Diamond \langle A \rangle_v)$$

Problem: merging fairness condx: when $\text{WF}_v(A \vee B) \Rightarrow \text{WF}_v(A) \wedge \text{WF}_v(B)$?

- Lamport's solution: defining

$$\begin{aligned} DR1 &\triangleq \Box(\text{ENABLED } \langle A \rangle_v \Rightarrow \Box \neg \text{ENABLED } \langle B \rangle_v \vee \Diamond \langle A \rangle_v) \\ DR2 &\triangleq \Box(\text{ENABLED } \langle B \rangle_v \Rightarrow \Box \neg \text{ENABLED } \langle A \rangle_v \vee \Diamond \langle B \rangle_v) \end{aligned}$$

and proving

$$DR1 \wedge DR2 \Rightarrow (\text{WF}_v(A) \wedge \text{WF}_v(B) \equiv \text{WF}_v(A \vee B))$$

Proof: mutual implication, the 2 resulting parts shunted by contradiction.

Total: 2,5 pages (102–105), many subtheorems, no discovery regarding “why?”

- Funmath solution: discovering conditions by calculation

- Preamble: exploring simple algebraic properties of operators involved, using their definitions and predicate calculus, e.g., (with evident shorthands),
 (i) $\mathbb{E}(A \vee B) \equiv \mathbb{E}A \vee \mathbb{E}B$; (ii) $\langle A \vee B \rangle_v \equiv \langle A \rangle_v \vee \langle B \rangle_v$.
- Rules for \square and \diamond are obtained similarly, such as monotonicity over \Rightarrow and distributivity laws of \square over \wedge (written \square/\wedge), \diamond/\vee and $\square\diamond/\vee$.
 E.g., $wA \equiv \square(\square\mathbb{E}A \Rightarrow \diamond A)$ yields (iii) $wA \equiv \diamond\square\mathbb{E}A \Rightarrow \square\diamond A$.
- Discovering the condition for $w(A \vee B) \Rightarrow wA \wedge wB$ by calculating

$$\begin{aligned}
 & w(A \vee B) \Rightarrow wA \\
 & \equiv \langle \text{Form (iii) for } w \rangle (\diamond\square\mathbb{E}(A \vee B) \Rightarrow \square\diamond(A \vee B)) \Rightarrow \diamond\square\mathbb{E}A \Rightarrow \square\diamond A \\
 & \equiv \langle \text{Shunting for } \Rightarrow \rangle \diamond\square\mathbb{E}A \Rightarrow (\diamond\square\mathbb{E}(A \vee B) \Rightarrow \square\diamond(A \vee B)) \Rightarrow \square\diamond A \\
 & \equiv \langle \text{(i); dist. } \square\diamond/\vee \rangle \diamond\square\mathbb{E}A \Rightarrow (\diamond\square(\mathbb{E}A \vee \mathbb{E}B) \Rightarrow \square\diamond A \vee \square\diamond B) \Rightarrow \square\diamond A \\
 & \equiv \langle \text{O } \varphi \Rightarrow \text{O } (\varphi \vee \psi) \rangle \diamond\square\mathbb{E}A \Rightarrow \square\diamond A \vee \square\diamond B \Rightarrow \square\diamond A \\
 & \equiv \langle q \vee p \Rightarrow q \equiv p \Rightarrow q \rangle \diamond\square\mathbb{E}A \Rightarrow \square\diamond B \Rightarrow \square\diamond A
 \end{aligned}$$

So if we define $z(A, B) := \diamond\square\mathbb{E}A \Rightarrow \square\diamond B \Rightarrow \square\diamond A$, clearly
 $z(A, B) \wedge z(A, B) \equiv w(A \vee B) \Rightarrow wA \wedge wB$ (Note: \equiv , not just \Rightarrow)

Next topic

0. Motivation: dichotomies and opportunities: UT FACIANT OPUS SIGNA
1. The formalism, part A: a simple and problem-free language
2. The formalism, part B: clear and problem-free formal rules
3. Illustrations in classical mathematics (analysis, discrete math)
4. Illustrations in mathematics related to Computing Science
5. Conclusion — Unifying classical and computing-related math (EE and CS)

5 **Conclusion — Electrical and Computer Engineering Unified**

- What we have shown
 - A formalism with a very simple language and powerful formal rules
 - Notational & methodological unification of CS and other engineering
 - Unification also encompassing a large part of mathematics (cont./discr.)
- Ramifications
 - Scientific: obvious (unified framework, highlighting analogies etc.)
 - 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).

- A major opportunity offered by mathematical software

- Observe: classical math tools (Maple etc.) easy to use, widely accepted
Ease of use and acceptance much lower for logical and discrete math tools
- Cause: long-standing, cleaned-up, uniform conventions vs. the opposite

Quoting a referee:

“Mathematicians spend a large part of their formative years on learning to cope with imperfections in existing mathematical conventions”

This statement was clearly written in a mood of resignation, but:

- * Should we lose any time and energy in nonproductive “friction”?
 - * Think of the future generations!
- Opportunity: keep the good parts, **get rid of defects**

- **Conclusion** Long-term advantages outweigh temporary “mathphobic” trends.

Some reference material

- Recent papers (2005–2006)

- Raymond Boute, “Functional Declarative Language Design and Predicate Calculus: a Practical Approach”, *ACM TOPLAS*, Vol. 27, No. 5, pp. 988–1047 (Sept. 2005)
- Raymond Boute, “Calculational semantics: deriving programming theories from equations by functional predicate calculus”, to appear in *ACM TOPLAS*, Vol. 28 (2006)
- Raymond Boute, Andreas Schäfer, “The Timer Cascade: Functional Modelling and Real Time Calculi”, in: Dang Van Hung, Martin Wirsing, eds., *Proc. 2nd International Colloquium on Theoretical Aspects of Computing (ICTAC’05)*, Springer LNCS 3722

- Seminars and tutorials (2003–2005)

- R. Boute, “Formal Calculation Unifying Engineering Theories Beyond Software”, *Summer School Marktoberdorf 2004/08/13*
(http://asimod.in.tum.de/2004/SLIDES/slides/mod04_boute_all.pdf)
- R. Boute, “Formal Methods unifying Computing Science and Systems Theory”, *Berkeley CHESS Seminar 2004/10/12* (<http://chess.eecs.berkeley.edu/seminar.htm>)
- R. Boute, “Making CS and classical EE meet: unification by formalization”, *Stanford CSL Colloquium* (<http://www.stanford.edu/class/ee380/Abstracts/041013.html>)
- Tutorials (retrievable via Google by conference acronym given below)
SEFM2003 (Brisbane), ICTAC2004 (Guiyang), WCC2004 (Toulouse), FM05 (Newcastle)