Genericity by functionalization: defining data as functions[1]

Raymond Boute INTEC — Ghent University

Overview

0. **Principle**
1. **Origin** (induction: collecting useful functionals)
2. **Making the functionals generic** (design: generalizing the functionals)
3. **Applications in programming** (deduction: applying the new functionals)
   List of application areas, 2 examples discussed
4. **Issues to be further explored**

[1]Prepared for the 2002 Summer School and Workshop on Generic Programming.

0 Principle

- **Genericity**
  One program, applicable to various data types
- **Genericity by parametrization**
  Program adapted to the various data types
- **Genericity by functionalization**
  Data adapted by (re)definition as functions: uniform type interface concept

1 Origin: modelling continuous and discrete systems

1.0 Need for point-wise and point-free expressions

- **Central issue: eliminating the time variable**
- **Motivation**: may be quite different, e.g.,

  a. In Funmath (*Functional Mathematics*) [Boute, 1992]:
     transforming behavioural *specifications* into structural *realizations*

  b. In FRP (*Functional Reactive Programming*) [Hudak et al.]:
     avoiding time leaks and space leaks

  FRP chosen for comparison due to its familiarity to the Haskell community. Other formalisms: Silage (Hilfinger) for DSP, LabVIEW (National Instruments) for control etc.

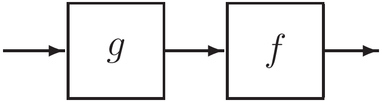
- **Results in diverging developments:** respectively,

  a. In Funmath: subsequent generalization to generic functionals

  b. In FRP: specialization to a DSL (domain-specific language)

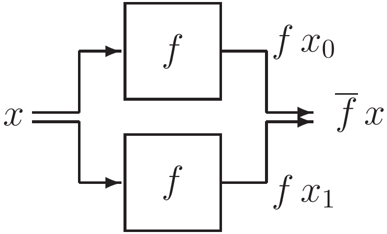
1.1 Example: potential genericity of function composition

- **Application A: expressing behaviour of memoryless devices**
  - Principle: extend static behaviours of type $A \rightarrow B$ to dynamic behaviours $\mathcal{S}_A \rightarrow \mathcal{S}_B$ for signals.
    The type for signals is $\mathcal{S}_X = \mathbb{T} \rightarrow X$ (for suitable time domain $\mathbb{T}$).
  - Specification (semantics): operator $^{-}$ defined by
    $\overline{f}\, s\, t = f\,(s\, t)$ for any $f : A \rightarrow B$ and $s : \mathcal{S}_A$
  - Realizations
    - In Funmath: *direct extension* operator $^{-}$ defined by $\overline{f}\ s = f \circ s$
      with $\circ$ defined as expected by $(f \circ g)\, x = f\,(g\, x)$
    - In FRP: operator `arr`, writing `arr f` for the signal behaviour
  - Remark: similar operator $\widehat{\ }$ with $(x \mathbin{\widehat{\star}} y)\, t = x\, t \mathbin{\widehat{\star}} y\, t$ for dyadic $\star$.

- **Application B:** structural cascade connection
  - Principle: 

  - Realization:
    - In Funmath: again using composition: $f \circ g$ for any functions
    - In FRP: operator >>>, writing `g >>> f` for signal behavours only
  - Remark: property $\overline{f \circ g} = \overline{f} \circ \overline{g}$ (proof: exercise), sugar $g\,;f = f \circ g$.

- **Application C:** function map
  Assuming *tuples as functions* in the sense that $(a, b)\,0 = a$ and $(a, b)\,1 = b$
  If $x = x_0, x_1$ then $f\,x_0, f\,x_1 = f \circ x = \overline{f}\,x$. Structural interpretation:



1.2 Example: potential genericity of function transposition

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

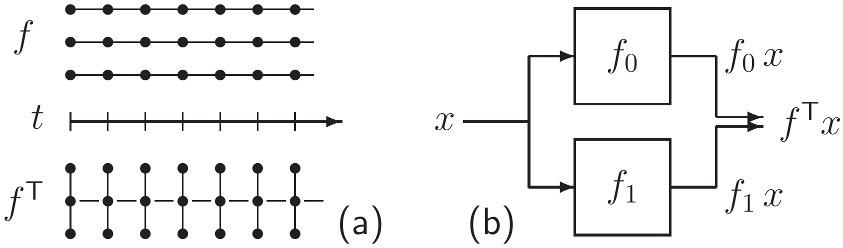
$$f^{\mathsf{T}} y\, x = f\, x\, y$$

  Nomenclature obviously borrowed from matrix theory (*up to curruing*)

- Structural interpretations:

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

  b. Signal fanout



2 Making the functionals generic

2.0 Principle: adding the "most general" type information

- **Conventions regarding functions**
  - Function = *domain* ($\mathcal{D}\,f$) and *mapping* (unique $f\,x$ for every x in $\mathcal{D}\,f$).
  - Function equality $\equiv$ equality of the domains and the mappings. Formally:
    $f = g \;\equiv\; \mathcal{D}\,f = \mathcal{D}\,g \wedge \forall\, x : \mathcal{D}\,f \cap \mathcal{D}\,g\,.\,f\,x = g\,x$
    Remark: in point-free style, $f = g \equiv \mathcal{D}\,f = \mathcal{D}\,g \wedge \forall\,(f \,\widehat{=}\, g)$

- **Making functionals generic**
  - Motivation: in Funmath, sharing by many more objects than usual.
  - Shortcomings of traditional operators: restrictions on the arguments, e.g.,
    - the usual $f \circ g$ requires $\mathcal{R}\,g \subseteq \mathcal{D}\,f$, in which case $\mathcal{D}\,(f \circ g) = \mathcal{D}\,g$
    - the usual f^- requires f injective, in which case $\mathcal{D}\,f^- = \mathcal{R}\,f$
  - Approach here: no restrictions on the arguments, but refine domain of the result function such that its mapping definition does not contain out-of-domain applications for values in this domain (guarded)

2.1 Functionals designed generically

For transforming and combining functions: for any func. f, pred. P, set S,

$\downarrow$	**Filtering**	$f \downarrow P = x : \mathcal{D}\, f \cap \mathcal{D}\, P \mathrel{\dot\wedge} P\, x \,.\, f\, x$
$\rceil$	**Restriction**	$f \rceil S = f \downarrow (S^\bullet 1)$
$\circ$	**Composition**	$f \circ g = x : \mathcal{D}\, g \mathrel{\dot\wedge} g\, x \in \mathcal{D}\, f \,.\, f\,(g\, x)$
$\&$	**Dispatching**	$f \mathbin{\&} g = x : \mathcal{D}\, f \cap \mathcal{D}\, g \,.\, f\, x, g\, x$
$\parallel$	**Parallel**	$f \parallel g = (x, y) : \mathcal{D}\, f \times \mathcal{D}\, g \,.\, f\, x, g\, y$
$\widehat{\ }$	**Extension**	$f \mathbin{\widehat{\star}} g = x : \mathcal{D}\, f \cap \mathcal{D}\, g \mathrel{\dot\wedge} (f\, x, g\, x) \in \mathcal{D}\,(\star) \,.\, f\, x \star g\, x$
$\ogreaterthan$	**Override**	$f \ogreaterthan g = x : \mathcal{D}\, f \cup \mathcal{D}\, g \,.\, (x \in \mathcal{D}\, f) \mathbin{?} f\, x \dagger g\, x$
$\dot\cup$	**Merge**	$f \mathbin{\dot\cup} g = x : \mathcal{D}\, f \cup \mathcal{D}\, g \mathrel{\dot\wedge} (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x) \,.\, (f \ogreaterthan g)\, x$

Relational functionals: for any func. f, pred. P, set S,

$\copyright$	**Compatibility**	$f \copyright g \equiv f \rceil \mathcal{D}\, g = g \rceil \mathcal{D}\, f$
$\subseteq$	**Subfunction**	$f \subseteq g \equiv f = g \rceil \mathcal{D}\, f$

Examples of algebraic properties:

$f \subseteq g \equiv \mathcal{D}\, f \subseteq \mathcal{D}\, g \wedge f \copyright g$ and $f \copyright g \Rightarrow f \ogreaterthan g = f \mathbin{\dot\cup} g = f \olessthan g$

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

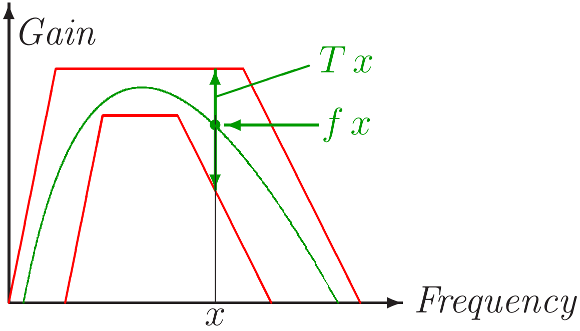
For equality: $f \copyright g \equiv \forall\,(f \mathrel{\widehat{=}} g)$ and $f = g \equiv \mathcal{D}\, f = \mathcal{D}\, g \wedge f \copyright g$.

2.2 Elastic extensions for generic operators

- **Elastic operators in general**
  - Principle: functionals replacing the common ad hoc abstractors, e.g., $\forall\, x : X$ and $\sum_{i=m}^{n}$ and $\lim_{x \to a}$
  - Simple examples: $\forall\, P \equiv P = \mathcal{D}\, P^{\bullet} 1$ and $\exists\, P \equiv P \neq \mathcal{D}\, P^{\bullet} 0$
  - Syntactic properties: (i) together with function abstraction, they yield familiar forms of expressions, e.g., $\forall\, x : X \,.\, P\, x$ and $\sum i : m \,..\, n \,.\, x_i$
    (ii) with tuples: $x \wedge y \equiv \forall\, (x, y)$, i.e., $\forall$ is *elastic extension* of $\wedge$, etc.
- **A typical elastic generic functional: transposition ($—^{\mathsf{T}}$)**
  - Intersecting variant: for any family f of functions,
    $$f^{\mathsf{T}} = y : (\bigcap\, x : \mathcal{D}\, f \,.\, \mathcal{D}\,(f\, x)) \,.\, x : \mathcal{D}\, f \,.\, f\, x\, y$$
    This is an elastic extension of & since $f \mathbin{\&} g = (f, g)^{\mathsf{T}}$
  - Uniting variant: for any family f of functions,
    $$f^{\mathsf{U}} = y : (\bigcup\, x : \mathcal{D}\, f \,.\, \mathcal{D}\,(f\, x)) \,.\, x : \mathcal{D}\, f \mathrel{\dot\wedge} y \in f\, x \,.\, f\, x\, y$$
- **Analogous elastic extensions:** $\|$ for $\|$, $\bigcup\!\!\!\cdot$ for $\cup\!\!\!\cdot$, Ⓒ for ©, etc.

- **A generic type constructor**

  - Purpose: formalizing *tolerances* for *functions*: a function f *meets* tolerance T iff $\mathcal{D}\, f = \mathcal{D}\, T \wedge \forall\, x : \mathcal{D}\, f \cap \mathcal{D}\, T \,.\, f\, x \in T\, x$. Illustration:



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

$$f \in \times T \;\equiv\; \mathcal{D}\, f = \mathcal{D}\, T \wedge \forall\, x : \mathcal{D}\, f \cap \mathcal{D}\, T \,.\, f\, x \in T\, x. \qquad (1)$$

  - Properties of (1):
    * The usual Cartesian product as a special case: $\times(A, B) = A \times B$
    * Dependent types as a special case: $\times(a : A \,.\, B_a)$

- **More about the funcart operator $\times$**
  - "Workhorse" for typing all structures unified by functional mathematics.
    - Recall $A \times B = \times(A, B)$ Also $A \to B = \times(A^{\bullet} B)$
    - Array types $A^n = \times(\square n^{\bullet} A)$, list types $A^* = \bigcup n : \mathbb{N} \,.\, A^n$,
    - Record types: instead of using projection/selection functions (Haskell, Scheme etc.), we define *records as first-class functions*.
      Domain = *enumeration type* for field labels. Example: given

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

      $person : Person$ satisfies $person\, \mathtt{name} \in \mathbb{A}^*$ and $person\, \mathtt{age} \in \mathbb{N}$.
      Sugar: defining $\mathrm{record} : \mathrm{fam}\,(\mathrm{fam}\,\mathcal{T}) \to \mathcal{P}\,\mathcal{F}$ with $\mathrm{record}\, F = \times(\cup F)$,
      we can write $Person := \mathrm{record}\,(\mathtt{name} \mapsto \mathbb{A}^*, \mathtt{age} \mapsto \mathbb{N})$.
  - Is a genuine functional, not an ad hoc abstractor! Noteworthy: inverse.
    - Choice axiom $\times T \neq \emptyset \equiv \forall x : \mathcal{D}\, T \,.\, T\, x \neq \emptyset$ characterizes Bdom $\times$
    - Implicit image definition: If $\times T \neq \emptyset$, then $\times^{-}(\times T) = T$
    - Explicit image definition: $\times^{-} S = x : Dom\, S \,.\, \{f\, x \mid f : S\}$ for $S : \mathcal{D}\, \times^{-}$

3 Applications in programming

3.0 Typical applications

(Presented at the 2002 Working Conference on Generic Programming)
Applications in:

0. **Functional programming:** composition, inversion, transposition of lists, pattern matching as function inverse
1. **Aggregate data types and structures**
2. **Overloading and polymorphism** various styles
3. **Functional predicate calculus** point-free and point-wise
4. **Formal semantics**
5. **Relational databases** functional style
6. **Relation algebra**

3.1 Two illustrations

– **Using generic functionals in the relational theory of data types**

* Example A Let relations be defined as boolean-valued functions (as in programming, logic) and write $\rho\, f$ for the relation representing function f ($y\, \rho f\, x \equiv y = f\, x$). Backhouse's definition for $\rightarrow$ is then

$$f\, R{\rightarrow}S\, g \;\equiv\; R \sqsubseteq (\rho f)^{\smile} \bullet S \bullet (\rho g)$$

Using generic functionals: $(\rho f)^{\smile} \bullet S \bullet (\rho g) = (S) \circ (f \parallel g)$.

* Example B With data types as functions, commuting datatypes can be expressed as follows. Given $f, g : F\, A \times G\, B$
  · Transformation via $F\,(A \times G\, B)$ to $F\,(G\,(A \times B))$ by broadcasting yields successively $x : \mathcal{D}\, f\, .\, (f\, x, g)$ and $x : \mathcal{D}\, f\, .\, y : \mathcal{D}\, g\, .\, (f\, x, g\, y)$
  · Transformation via $G\,(F\, A \times B)$ to $G\,(F\,(A \times B))$ by broadcasting yields successively $y : \mathcal{D}\, g\, .\, (f, g\, y)$ and $y : \mathcal{D}\, g\, .\, x : \mathcal{D}\, f\, .\, (f\, x, g\, y)$

Clearly $(x : \mathcal{D}\, f\, .\, y : \mathcal{D}\, g\, .\, (f\, x, g\, y))^{\mathsf{T}} = y : \mathcal{D}\, g\, .\, x : \mathcal{D}\, f\, .\, (f\, x, g\, y)$

– **Using generic functionals in abstract syntax description**

Reason for this example: archetype for expressing structures as functions.

* For *aggregate constructs* and *list productions*: functional record and *.
  This is $\times$ actually: record $F = \times(\uplus F)$ and $A^* = \bigcup n : \mathbb{N} \,.\, \times(\square\, n \,^\bullet A)$.
* For *choice productions* needing disjoint union: generic elastic $\big|$-operator
  For any family F of types,

$$\Big| F = \bigcup x : \mathcal{D}\,F \,.\, \{x \mapsto y \mid y : F\,x\} \tag{2}$$

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

  Remarks

  · Variadic shorthand: $A \mid B = \Big|(A, B) = \{0 \mapsto a \mid a : A\} \cup \{1 \mapsto b \mid b : B\}$

  · Using $x \mapsto y$ rather than the common x, y yields more uniformity.

  · Same 3 operators can describe directory and file structures, XML, ...

  · For program semantics, disjoint union is often "overengineering".

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

$$\mathbf{def}\ \mathit{Program} := \mathsf{record}\,(\mathit{declarations} \mapsto \mathit{Dlist}, \mathit{body} \mapsto \mathit{Instruction})$$
$$\mathbf{def}\ \mathit{Dlist} := D^*$$
$$\mathbf{def}\ D := \mathsf{record}\,(v \mapsto \mathit{Variable}, t \mapsto \mathit{Type})$$
$$\mathbf{def}\ \mathit{Instruction} := \mathit{Skip} \cup \mathit{Assignment} \cup \mathit{Compound} \cup \mathsf{etc.}$$

A few items are left undefined here (easily inferred).
If disjoint union wanted: $\mathit{Skip} \mid \mathit{Assignment} \mid \mathit{Compound} \mid \mathsf{etc.}$
Instances of programs, declarations, etc. can be defined as

$$\mathbf{def}\ p : \mathit{Program}\ \mathbf{with}\ p = \mathit{declarations} \mapsto \mathit{dl} \mathbin{\dot\cup} \mathit{body} \mapsto \mathit{instr}$$

Observation: very similar to Haskell data type definitions.

4 Issues to be further explored

- Data types directly expressed as function types (e.g., records, lists, trees): no problem, but functional data types derived from recursive definitions would be more convenient in uncurried form (domain = set of paths)
- Expressing data as functions does not affect decidability issues, but may require defining a special class of (more than first-class) functions
- Feasibility of full implementation of domain computations in current languages is unclear (small experiment with shallow embedding in Haskell recently assigned to a student)