Data processing by calculation

EWSCS'01 — 6th Estonian Winter School in Computer Science
4-9 March 2001
Palmse
Estonia

J. Nuno Oliveira
Departamento de Informática
Universidade do Minho
Braga
Portugal

email: jno@di.uminho.pt
http://www.di.uminho.pt/~jno

Motivation

Context

- State-of-the-art information systems are by-and-large based on the **relational data model**.
- Despite the Codd's calculus (c. 1970s), most relational **databases** are still designed and maintained in a fairly informal way.

This has raised a world-wide **data legacy** problem.

- Currently, this aging technology is under pressure, challenged to scale-up to **heterogeneous** data-models such as those implicit in mark-up languages spreading over the **Internet**.

There is a need for much clever techniques for **data-mining**, on-line data analysis (**OLAP**) *etc.*

- This series of lectures will address information system design from an alternative, **formal** viewpoint.

Overview

- Abstract **mathematical** data-models will be built to capture user requirements.

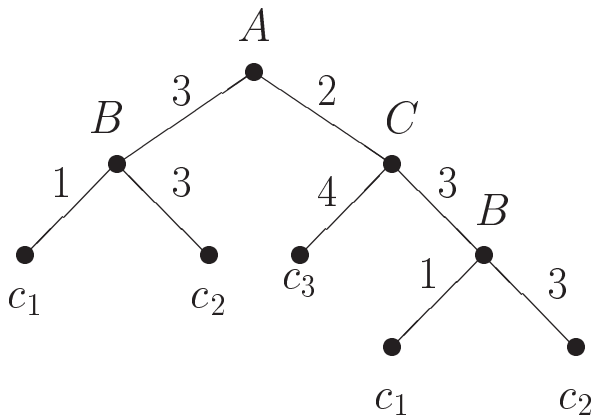
  $\Rightarrow$ not just *drawings*, such as in *e.g.* the ***UML***

- Abstract models are based on a **calculus** of data combinators which exhibit universal **properties** which help in **reasoning**, transforming a calculating about them.
- Data-model calculation encompasses not only data **refinement** but also data transformation, **data-mining** and data migration.
- Thanks to the **genericity** of the approach — which stems from its functorial basis — some standard relational results are shown to extend polytypically, thus giving birth to a new field of research in information systems — that of **"polytypic data-processing"**.
- This exhibits some potential to cope with the **heterogeneous** data sources which characterize modern INTERNET-based information system solutions.
- The data combinators presented in these lectures are naturally expressed in a functional, **pointfree** style. All examples will be expressed in the HASKELL functional programming language.

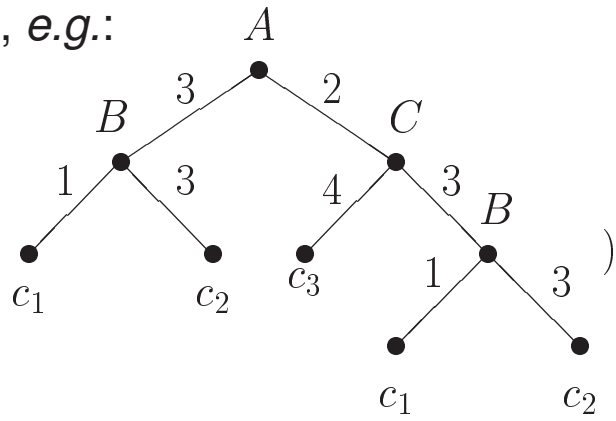
Motivation

Consider the following informal **requirements**:

1. A factory builds **equipments** involving individual **components** obtained from a second source;
2. each individual component has a *cost* and is **stocked** in some *quantity*;
3. components participate, in fixed quantities, in the production of *equipments* (*e.g.* PC board ref. X has n integrated circuits of ref. Y);
4. equipments may contain, in fixed quantities, other equipments as *sub-blocks* (*e.g.* personal computer ref. Z has m PC boards of ref. X).
5. (sub)equipments are stocked along with components in order to be readily available for production.
6. a **production tree** can be displayed for each equipment which will involve individual components and/or other production sub-trees, for instance:



7. the functionality to be provided includes that of computing the **part explosion** of one equipment according to its production tree, *e.g.*:

$explode($ 

$) =$

$Comp$	$\#$
c_1	9
c_2	27
c_3	8

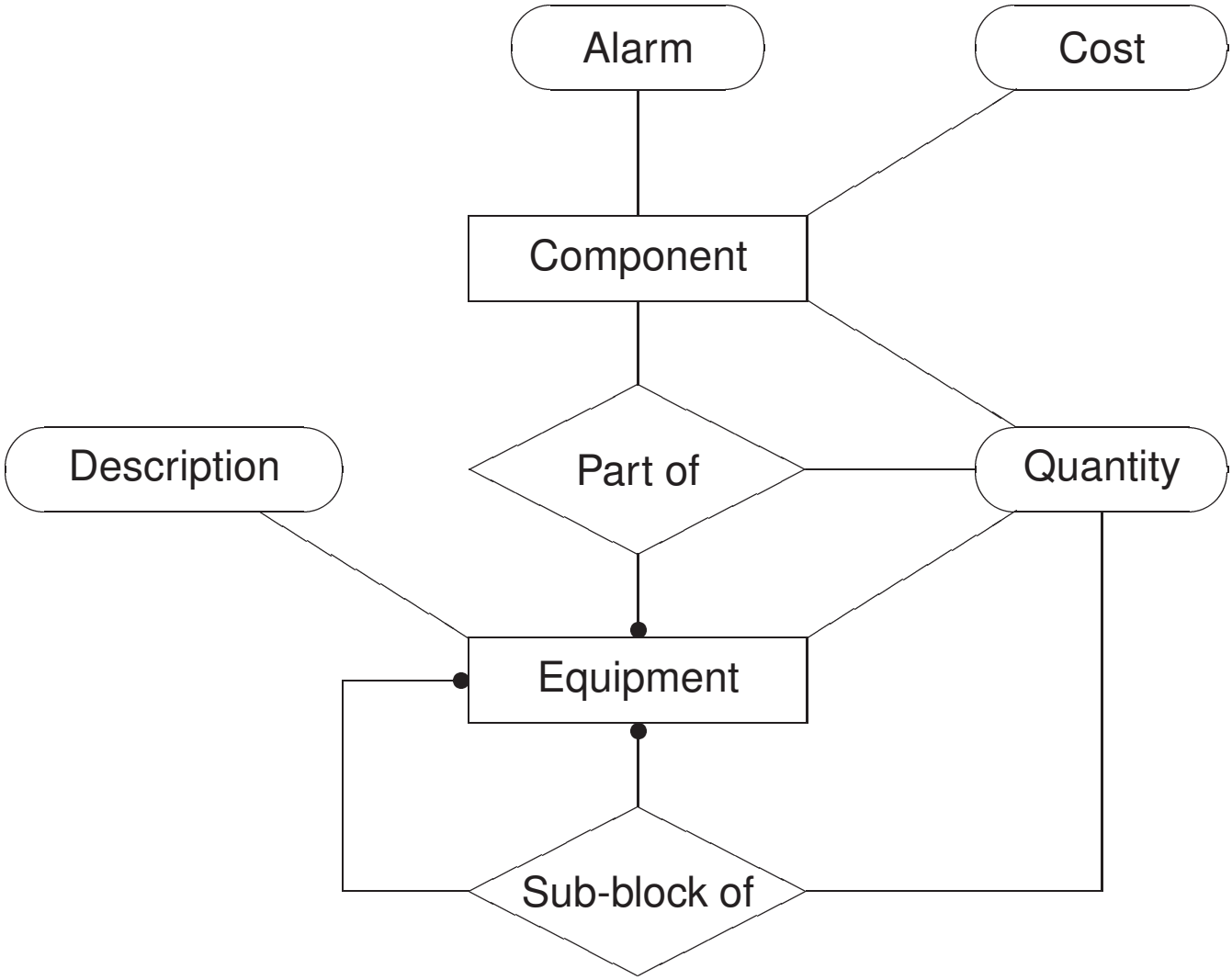
$\Rightarrow$ The **bill-of-materials** problem.

Comments:

- This is a typical **IS** (information system) problem.
- How would the **IS** community address it?

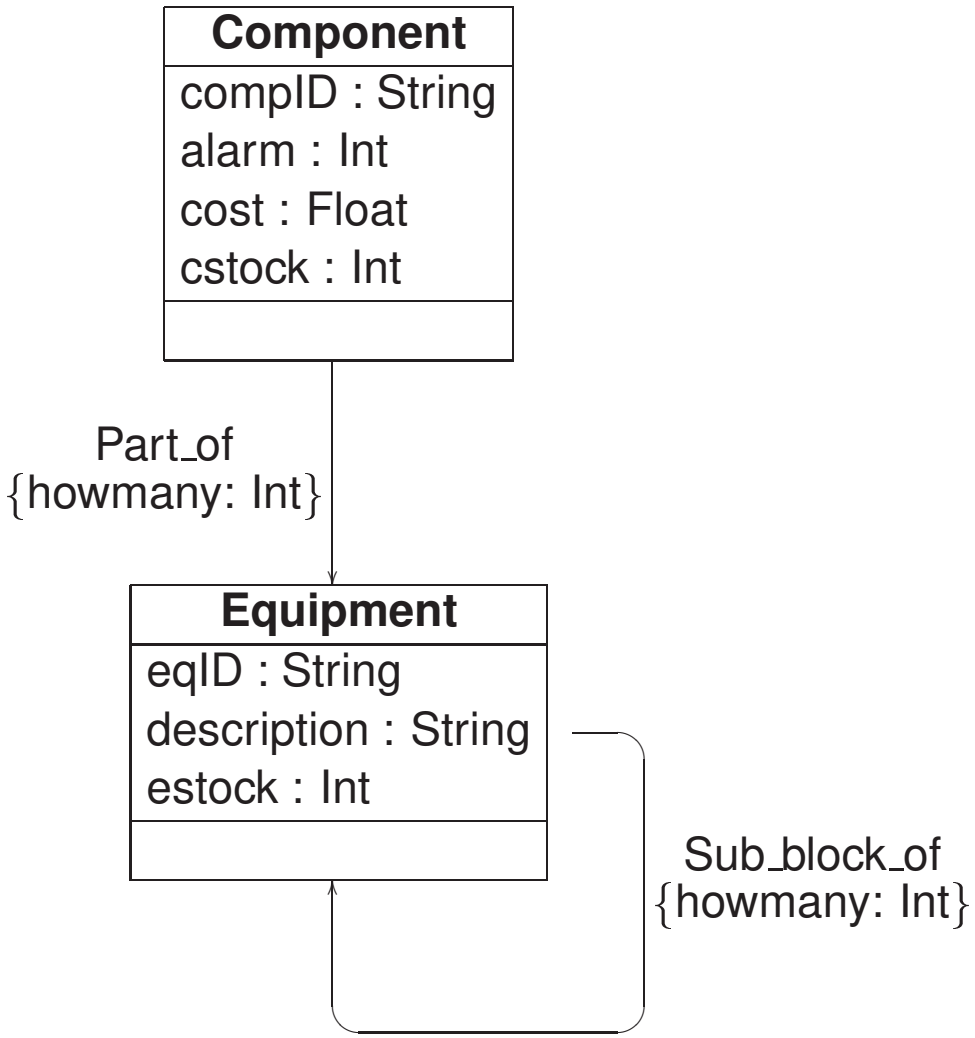
One first needs to build a **data-model**.

Why not “good old” **E-R diagrams**?



Too *old-fashioned* !

Try the ***UML***:



However,

What do these **drawings** *mean* ?

How does one record and **ensure**. . .

1. . . . desirable ***properties*** *of the data model such as,* e.g.*:*

*No equipment can (**recursively**) be a sub-block of itself.*

2. . . . correct ***behaviour*** *of functionalities such as* $explode$ *above?*

Anyway,

(informal) drawings are better than diving straight into raw SQL code . . .

```
CREATE TABLE COMPONENTS (
  CompId CHAR (8) NOT NULL,
  Alarm NUMBER (10) NOT NULL,
  Cost NUMBER (3,3) NOT NULL,
  CStock NUMBER (10) NOT NULL,
  CONSTRAINT COMPONENTS_pk PRIMARY KEY(Comp
);

CREATE TABLE EQUIPMENTS (
  EqId CHAR (8) NOT NULL,
  Description CHAR (73) NOT NULL,
  EStock NUMBER (10) NOT NULL,
```

```
  CONSTRAINT EQUIPMENTS_pk PRIMARY KEY (EqI
);

CREATE TABLE PART_OF (
  Comp CHAR (8) NOT NULL,
  Equip CHAR (8) NOT NULL,
  HowManyC NUMBER (10) NOT NULL,
  CONSTRAINT EQUIPMENTS_pk PRIMARY KEY (Com
);

CREATE TABLE SUB_BLOCK_OF (
  EquipL CHAR (8) NOT NULL,
  EquipS CHAR (8) NOT NULL,
  HowManyE NUMBER (10) NOT NULL,
  CONSTRAINT EQUIPMENTS_pk PRIMARY KEY (Equ
);

ALTER TABLE PART_OF ADD CONSTRAINT PART_OF_
FOREIGN KEY (Comp) REFERENCES COMPONENTS(Co

ALTER TABLE PART_OF ADD CONSTRAINT PART_OF_
FOREIGN KEY (Equip) REFERENCES EQUIPMENTS(E

ALTER TABLE SUB_BLOCK_OF ADD CONSTRAINT SUB
FOREIGN KEY (EquipL) REFERENCES EQUIPMENTS(

ALTER TABLE SUB_BLOCK_OF ADD CONSTRAINT SUB
FOREIGN KEY (EquipS) REFERENCES EQUIPMENTS(
```

. . . although some might be tempted to accept a *similar* model if encoded into a high-level language such as *e.g.* HASKELL:

```
-- (1) Entities --

data Component = Comp_Record {
        compID :: String,
        alarm  :: Int,
        cost   :: Float,
        cstock  :: Int
        }

data Equipment = Eq_Record {
        eqID          :: String,
        description   :: String,
        estock        :: Int
        }

-- (2) Relationships --

data Part_of    = Part_of_Record {
        comp    :: String,
        equip   :: String,
        howManyC :: Int
        }

data Sub_Block_of = Sub_Block_of_Record {
        equipL   :: String,
        equipS   :: String,
```

```
        howManyE :: Int
        }

-- (3) Relational tables --

type Components = Set Component
type Equipments = Set Equipment
type Parts_of   = Set Part_of
type Sub_Blocks_of = Set Sub_Block_of
```

Questions:

- Which of the alternatives above the **"best"**, if any?
- Is there a framework **binding** all these together?

We have to...

- **go "more abstract"**;
- specify things in a **formal notation** able to express **IS** requirements free from *ambiguities*.
- **calculate** implementations from such specifications rather than **inventing** them (*cf.* coding errors).
- In between, we may need to *rapid prototype* our specs (*cf.* design errors).

$\Rightarrow$ For spec "animation" we will resort to HASKELL.

Introducing the Notation

Records *versus* products

- Broadly speaking, a database is that part of an **IS** which collects **facts** or **records** about a particular organization, which are under permanent retrieval and processing.
- There may be *millions* of such records archived in a database and so efficiency may be an issue. But, prior to that...

$\Rightarrow$ ... what *is* a record?

Example

A collection of facts or records:

compID	**alarm**	**cost**	**cstock**
012.01.03	1200	1.5	2500
⋮	⋮	⋮	⋮
914.15.30	200	10.5	500

where, for instance, | 012.01.03 | 1200 | 1.5 | 2500 | means

...2500 units of component "012.01.03" are stored at unitary cost 1.5 EURO; an alarm message is to be raised whenever the stock level of this component is below 1200 units.

However,

- a matematician would have written $(\texttt{012.01.03}, 1200, 1.5, 2500)$ instead of *drawing* the tabular stuff;

- clearly, he would have inferred

$$\frac{\begin{array}{c}\texttt{012.01.03} \in String,\\ 1200 \in Int,\\ 1.5 \in Float,\\ 2500 \in Int\end{array}}{(\texttt{012.01.03}, 1200, 1.5, 2500) \in String \times Int \times Float \times Int}$$

Summary: records are **tuples** which inhabit **products** of types.

First of all, we need a teory for (Cartesian) products:

$$A \times B \stackrel{\text{def}}{=} \{(a,b) \mid a \in A \wedge b \in B\}$$

Comments:

- Product is but an example of a *data constructor* which arises very naturally if expressed in terms of its build/inspect operators — *i.e.*, its **algebra**.
- A "category-theory flavoured" notation is very effective in describing not only **structured data** and associated **operators** but also their inherent **properties**.
- For this—and more to come—we will resort to categorial **arrow** notation and diagrams.

Vital rôle of categorial foundations

- Makes it possible to **integrate** seemingly disparate schools of thought.

- Paves the way to truly **generic** results which go beyond polymorphism.
- Dualities: categorial notions "come in pairs", *e.g. product*, *coproduct*.
- Categorial notions more and more familiar to (functional?) programmers.

However,

- No deep background on category theory required!
- Many technical details omitted for the sake of brevity.
- Categorial concepts interpreted **functionally**:
  - Object are **sets**
  - Morphisms are set-theoretic **functions**.
- HASKELL helpful in "animating" this notation.

Notation preliminaries

- **Functions**: given two sets A and B, the notion of a function f from A to B, denoted $A \xrightarrow{f} B$ or $B \xleftarrow{f} A$ or $f :: A \to B$ will be regarded as primitive.
- **Functional application** is denoted by juxtaposition, *e.g.* $f\ a$, or $f(a)$ in case a is a structured expression.
- **Function composition**:

$$\frac{B \xleftarrow{f} C \quad , \quad C \xleftarrow{g} A}{B \xleftarrow{f \cdot g} A}$$

where

$$(f \cdot g)\, a \;=\; f(g\, a) \tag{1}$$

- **Identity functions**: every set A is equipped with identity function $A \xrightarrow{id_A} A$ such that $id_A\, a = a$, for every $a \in A$ (id will be used wherever A is context-implicit).
- **Invertibility**: we draw

$$A \underset{f}{\overset{r}{\leq}} B \quad \text{iff} \quad f \cdot r = id_A \tag{2}$$

  and say that:
  - r is a *right inverse* of f
  - f is a *left inverse* of r
- **Isomorphism** (or abstract equality): we draw

$$A \underset{f}{\overset{f^{-1}}{\cong}} B \quad \text{iff} \quad \begin{cases} f \cdot f^{-1} = id_A \\ f^{-1} \cdot f = id_B \end{cases} \tag{3}$$

  that is, f^{-1} is both a *right-inverse* and a *left-inverse* of f.

A and B are said to be **isomorphic** (or "abstractly equal") and f, f^{-1} are said to be isomorphisms.

- **Constant functions**: The "everywhere c" function, for $c \in C$, is denoted by $\underline{c}$ (`const c` in HASKELL):

$$\begin{array}{rcl} \underline{c} & : & A \longrightarrow C \\ \underline{c}\, a & \overset{\text{def}}{=} & c \end{array} \tag{4}$$

The following property defines constant functions at point-free level:

$$\underline{c} \cdot f \;=\; \underline{c} \quad \text{as in diagram} \quad \begin{array}{ccc} C & \xleftarrow{\underline{c}} & A \\ {\scriptstyle id}\downarrow & & \downarrow{\scriptstyle f} \\ C & \xleftarrow[\underline{c}]{} & B \end{array} \tag{5}$$

Pointfree 'versus' pointwise notation

$$\underbrace{f \cdot g}_{\text{pointfree}} \quad = \quad \underbrace{\lambda a.f\,(g\,a)}_{\text{pointwise}}$$

- **Pointfree** notation is more compact, and leads to shorter descriptions in which patterns are easier to identify and to reason about.
- **Properties** can be stated in clear-cut, one-line long equations which are easy to memorize, *e.g.*:

$$(f \cdot g) \cdot h = f \cdot (g \cdot h) \tag{6}$$

$$f \cdot id = f = id \cdot f \tag{7}$$

- **Diagrams** can be drawn which enable us to visualize maths in a graphical format, *e.g.*:

$$\begin{array}{ccc} A & \xleftarrow{id} & A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\ B & \xleftarrow[id]{} & B \end{array} \tag{8}$$

Categorial *"toolboxes"*

Toolbox = an universal construction
+ associated **functor**
+ a kernel theory:
  - an **universal** property
  - a **reflection** law
  - a **fusion** law
  - a **natural** law
  - a **cancellation** law
  - an **absorption** law

The Product ($A \times B$) toolbox

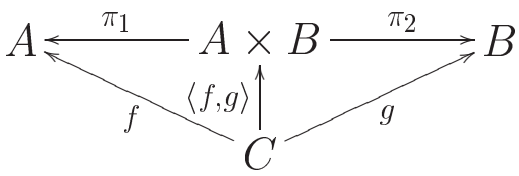
The **data** construct first:

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

where *projections* π_1 and π_2 are as follows:

$$\pi_1(a,b) \stackrel{\text{def}}{=} a \quad , \quad \pi_2(a,b) \stackrel{\text{def}}{=} b \tag{9}$$

Next, its standard "constructor" (the universal **morphism**):



(10)

where pointwise definition

$$\begin{array}{rcl} \langle f,g\rangle & : & C \longrightarrow A \times B \\ \langle f,g\rangle\, c & \stackrel{\text{def}}{=} & (f\, c, g\, c) \end{array} \tag{11}$$

is captured by a pointfree *universal property*

$$k = \langle f, g \rangle \Leftrightarrow \begin{cases} \pi_1 \cdot k = f \\ \pi_2 \cdot k = g \end{cases} \tag{12}$$

which establishes the *uniqueness* of the $\langle \_, \_ \rangle$ "record making" operator.

In HASKELL,

- the $A \times B$ construct is supported directly as `(a,b)`
- π_1 is written `fst`
- π_2 is written `snd`
- $\langle f, g \rangle$ is

```
split :: (a -> b) -> (a -> c) -> a -> (b,c)
split f g x = (f x, g x)
```

Associated (bi)functor

$$f \times g \stackrel{\text{def}}{=} \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \tag{13}$$

is pronounced "product of f and g" and has typing-rule

$$\frac{\begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} D \end{array}}{A \times B \xleftarrow{f \times g} C \times D} \tag{14}$$

cf.

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\ {\scriptstyle f}\uparrow & & {\scriptstyle f \times g}\uparrow & & \uparrow{\scriptstyle g} \\ C & \xleftarrow{\pi_1} & C \times D & \xrightarrow{\pi_2} & D \end{array} \tag{15}$$

In HASKELL:

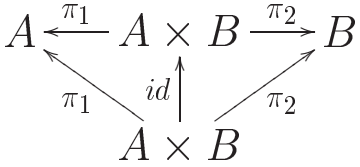
```
(><) :: (a -> b) -> (c -> d) -> (a,c) -> (b,d)
f >< g = split (f . fst) (g . snd)
```

Kernel properties

which can be derived from the universal property (12):

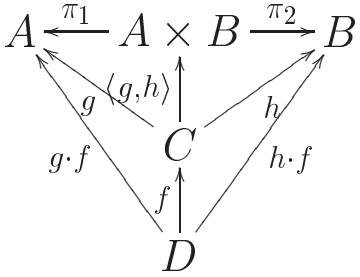
×-reflexion :

$$\langle \pi_1, \pi_2 \rangle \;=\; id_{A \times B} \tag{16}$$



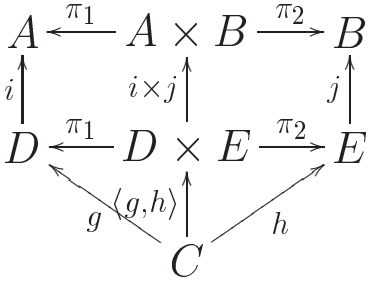
×-fusion :

$$\langle g, h \rangle \cdot f = \langle g \cdot f, h \cdot f \rangle \tag{17}$$



×-absorption :

$$(i \times j) \cdot \langle g, h \rangle = \langle i \cdot g, j \cdot h \rangle \tag{18}$$

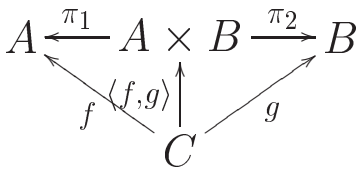


×-natural :

$$\begin{array}{rcl} i \cdot \pi_1 & = & \pi_1 \cdot (i \times j) \\ j \cdot \pi_2 & = & \pi_2 \cdot (i \times j) \end{array} \tag{19}$$

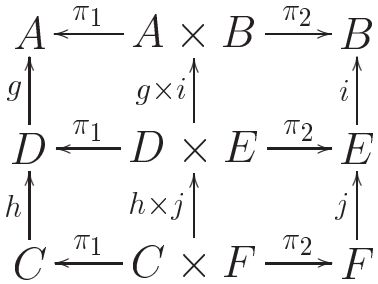
×-cancellation :

$$\pi_1 \cdot \langle f, g \rangle = f \quad , \quad \pi_2 \cdot \langle f, g \rangle = g \tag{20}$$



×-functor :

$$(g \cdot h) \times (i \cdot j) \;=\; (g \times i) \cdot (h \times j) \tag{21}$$



×-functor-id :

$$id_A \times id_B \;=\; id_{A \times B} \tag{22}$$

Terminology (*cf. natural*, *functor*) to be justified later

Isomorphism properties

Set theoretic **equality** (as adopted in standard database theory) is too poor and restrictive.

Equality up to **isomorphism** is much more useful, for instance:

$$A \times B \underset{swap}{\overset{swap}{\cong}} B \times A \tag{23}$$

where

$$swap \stackrel{\text{def}}{=} \langle \pi_2, \pi_1 \rangle \tag{24}$$

Intuitively:

- Swapping the entries in a database record does not change its meaning — if performed consistently.
- $A \times B$ and $B \times A$ are "different" but they "contain the *same amount* of information".

Another fact:

$$A \times (B \times C) \underset{assocr}{\overset{assocl}{\cong}} (A \times B) \times C \tag{25}$$

where

$$assocl \stackrel{\text{def}}{=} \langle id \times \pi_1, \pi_2 \cdot \pi_2 \rangle \tag{26}$$

From binary to n-ary records ($n \geq 0$)

- (23) and (25) mean that product is **commutative** and **associative**.

- So we can drop parentheses and write $A \times (B \times C)$ simply as $A \times B \times C$, leading to **finite products**

$$A_1 \times \ldots \times A_n$$

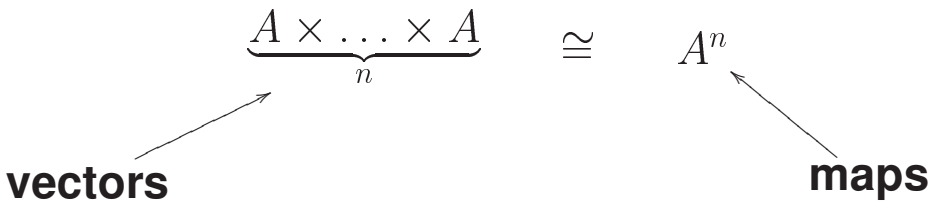
bearing π_i projections ($i = 1, \ldots, n$).

Question:

Does

$$\underbrace{A \times \ldots \times A}_{n} \;\cong\; A^n \tag{27}$$

hold? It does but A^n is not just an abbreviation — it deserves a proper explanation:



So,

- $r \in \underbrace{A \times \ldots \times A}_{n}$ is an n-ary data record or **vector**

$$r = (a_1, \ldots, a_n)$$

where $a_i \in A, i \in n$.

- better than "…"-notation: $r \in A^n$ is a **map** from n indexes to A-values

$$r :: \mathbf{n} \longrightarrow A$$

where $\mathbf{n}$ denotes the *initial segment* of $\mathbb{N}$ determined by $n \geq 0$:

$$\begin{aligned} \mathbf{0} &= \{\} \\ \mathbf{n+1} &= \{n+1\} \cup \mathbf{n} \end{aligned}$$

Product unit

For $n = 0$ we have a "degenerated" product containing a single datum, which is denoted by 1. The corresponding universal morphism is usually denoted by $!$:

$$\begin{array}{c} 1 \\ \uparrow ! \\ A \end{array}$$

Clearly, for all a:

$$1 \cong \{a\}$$

In HASKELL, 1 is datatype `()` whose single inhabitant is also denoted by `()`. Therefore:

```
(!) :: a -> ()
(!) = const ()
```

Datatype 1 is useful for modelling not only NULL values in database records, but also NIL *pointers* in C/C++.

Property: 1 is the *unit* of product

$$A \times 1 \underset{\langle id,! \rangle}{\overset{\pi_1}{\underset{\cong}{\rightleftarrows}}} A \tag{28}$$

Exponentials

Data object A^n above is an instance of the *exponential* data type

$$B^A \;=\; \{f \;::\; A \longrightarrow B\}$$

inhabited by total maps from A to B — *i.e.* A-indexed collections of B-values. Therefore, projection is generalized to *application* in B^A:

$$B^A \times A \xrightarrow{ap} B \tag{29}$$

where $ap(f, a) \stackrel{\text{def}}{=} f\ a$.

The exponentials "toolbox"

In the laws which follow,

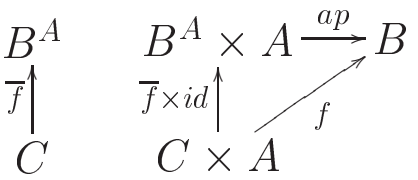
- $curry\ f$ is abbreviated to $\overline{f}$ and
- for $B \xrightarrow{f} C$ one defines $B^A \xrightarrow{f^A} C^A$ as follows:

$$f^A = \overline{f \cdot ap} \text{ cf. diagram: } \begin{array}{ccc} C & & C^A \times A \xrightarrow{ap} C \\ \uparrow f & & \uparrow f^A \times id \quad \uparrow f \\ B & & B^A \times A \xrightarrow{ap} B \end{array} \tag{30}$$

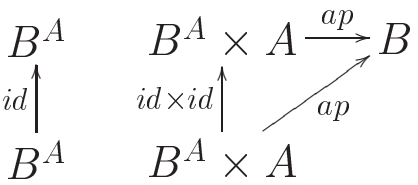
i.e. $f^A\ g = f \cdot g$.

Exp.-universal : $$k = \overline{f} \;\Leftrightarrow\; f = ap \cdot (k \times id) \tag{31}$$

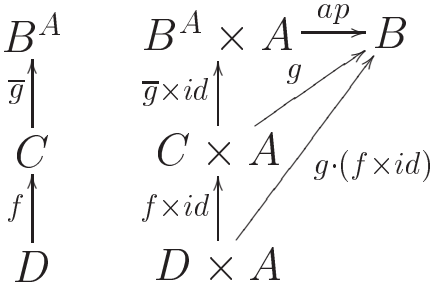
Exp.-cancellation : $$f = ap \cdot (\overline{f} \times id) \tag{32}$$



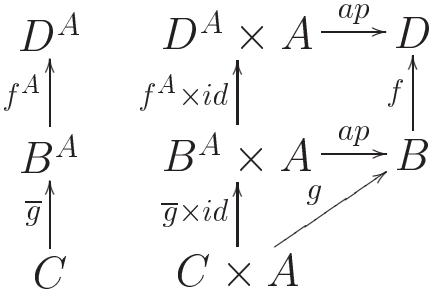
Exp.-reflexion : $$\overline{ap} = id \tag{33}$$



Exp.-fusion : $$\overline{g \cdot (f \times id)} = \overline{g} \cdot f \tag{34}$$



Exp.-absorption : $$\overline{f \cdot g} = f^A \cdot \overline{g} \tag{35}$$



Exp.-natural : $$f \cdot ap = ap \cdot (f^A \times id) \tag{36}$$

Exp.-functor : $$(g \cdot h)^A = g^A \cdot h^A \tag{37}$$

Exp.-functor-id : $$id^A = id \tag{38}$$

Notation:

Very often we shall abbreviate $\overline{f}\, a$ — *i.e.* $curry\, f\, a$ — to f_a. For instance, $=_a$ denotes the unary "equal to a" predicate — `(==a)` in HASKELL.

Some useful isomorphisms

$$1^A \underset{!}{\overset{\underline{()}}{\cong}} 1 \tag{39}$$

$$A^1 \underset{const}{\overset{\lambda f.f()}{\cong}} A \tag{40}$$

$$(B \times C)^A \underset{\langle -,- \rangle}{\overset{\langle \pi_1^A, \pi_2^A \rangle}{\cong}} B^A \times C^A \tag{41}$$

$$B^{C \times A} \underset{uncurry}{\overset{curry}{\cong}} (B^A)^C \tag{42}$$

where

$$const\, a = \underline{a} \tag{43}$$

$$curry\, f\, c\, a = f(c, a) \tag{44}$$

$$uncurry\, f\, p = f(\pi_1 p)(\pi_2 p) \tag{45}$$

Variant records *versus* coproducts

Although absent from classical database theory, *variant records*

are supported by ordinary programming languages such as PASCAL or C, *e.g.*

```
record                          struct {
   case                            int tag; /* 1,2 */
    tag:  integer                  union {
        of x =        or              A ifA;
     1:  (P:A);                       B ifB;
     2:  (S:B)                     } data;
end;                            };
```

Set-theoretically:

$$A + B = (\{1\} \times A) \cup (\{2\} \times B)$$

i.e.

$$A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$$

where *injections* i_1 and i_2 are as follows:

$$i_1\, a \stackrel{\text{def}}{=} (1, a) \quad , \quad i_2\, b \stackrel{\text{def}}{=} (2, b) \tag{46}$$

$A+B$ — disjoint-union or **coproduct** — *dualizes* $A \times B$ and is among the most useful devices in the mathematics of program construction.

The coproduct ($A + B$) "toolbox"

First, its standard "parser" (the universal **morphism**):

$$\begin{array}{ccccc} A & \xrightarrow{i_1} & A+B & \xleftarrow{i_2} & B \\ & \searrow^{f} & \downarrow{\scriptstyle [\,f,g\,]} & \swarrow_{g} & \\ & & C & & \end{array} \tag{47}$$

where

$$\begin{array}{rcl} [\,f,g\,] & : & A+B \longrightarrow C \\ [\,f,g\,]\,x & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lcl} x = i_1\,a & \Rightarrow & f\,a \\ x = i_2\,b & \Rightarrow & g\,b \end{array} \right. \end{array} \tag{48}$$

is again captured by a **universal** property:

$$k = [\,f,g\,] \;\Leftrightarrow\; \left\{ \begin{array}{l} k \cdot i_1 = f \\ k \cdot i_2 = g \end{array} \right. \tag{49}$$

The $f + g$ bifunctor:

$$\frac{\begin{array}{c} C \xleftarrow{f} A \\ D \xleftarrow{g} B \end{array}}{C + D \xleftarrow{f+g} A + B} \tag{50}$$

where

$$f + g \stackrel{\text{def}}{=} [\,i_1 \cdot f, i_2 \cdot g\,] \tag{51}$$

cf. diagram:

$$\begin{array}{ccccc} A & \xrightarrow{i_1} & A+B & \xleftarrow{i_2} & B \\ {\scriptstyle f}\downarrow & & {\scriptstyle f+g}\downarrow & & \downarrow{\scriptstyle g} \\ C & \xrightarrow[i_1]{} & C+D & \xleftarrow[i_2]{} & D \end{array}$$

Note that $f + g$ dualizes $f \times g$ (13).

In HASKELL, the $A + B$ datatype is supported directly via

```
data Either a b = Left a | Right b
```

Then $[\ f, g\ ]$ is

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  = f x
either f g (Right y) = g y
```

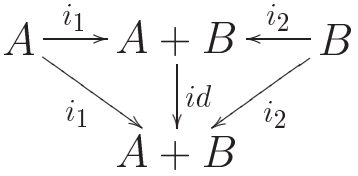
and $f + g$ can be defined by:

```
(-|-) :: (a -> b) -> (c -> d) -> Either a c -> Either b d
f -|- g = either (Left . f) (Right . g)
```

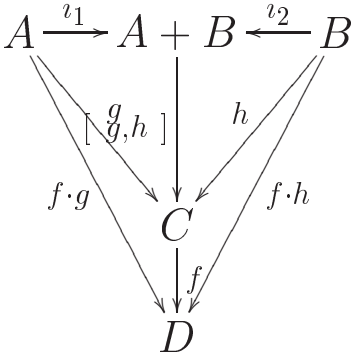
Kernel properties

$+$-cancellation :
$$[\ g, h\ ] \cdot i_1 = g\ ,\ [\ g, h\ ] \cdot i_2 = h \qquad (52)$$

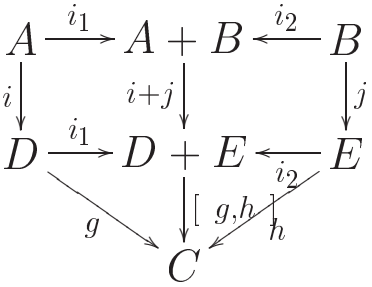
$+$-reflection :
$$[\ i_1, i_2\ ] = id \qquad (53)$$



+-fusion :
$$f \cdot [\ g, h\ ] = [\ f \cdot g, f \cdot h\ ] \tag{54}$$



+-absorption :
$$[\ g, h\ ] \cdot (i + j) = [\ g \cdot i, h \cdot j\ ] \tag{55}$$



. . . plus the "functorial goodies":

+-natural :
$$\begin{aligned}(i + j) \cdot i_1 &= i_1 \cdot i \\ (i + j) \cdot i_2 &= i_2 \cdot i\end{aligned} \tag{56}$$

+-functor :
$$(g \cdot h) + (i \cdot j) = (g + i) \cdot (h + j) \tag{57}$$

+-functor-id :
$$id_A + id_B = id_{A+B} \tag{58}$$

Coproduct unit

The empty object 0 is a "degenerated" coproduct containing no information. The corresponding universal morphism is an empty map usually denoted by $?$:

$$\begin{array}{c} 0 \\ {\scriptstyle ?}\downarrow \\ A \end{array}$$

Isomorphism properties

$$A + B \cong B + A \tag{59}$$
$$A + (B + C) \cong (A + B) + C \tag{60}$$
$$A + 0 \cong A \tag{61}$$
$$A \times 0 \cong 0 \tag{62}$$
$$A^0 \cong 1 \tag{63}$$

(Exercise: guess the arrows)

$$A \times (B + C) \underset{undistr}{\overset{distr}{\cong}} (A \times B) + (A \times C) \tag{64}$$

$$A^{B+C} \underset{[\ -,-\ ]}{\overset{[\ -,-\ ]^{-1}}{\cong}} A^B \times A^C \tag{65}$$

(66)

where

$$undistr \stackrel{\text{def}}{=} [\ id \times i_1, id \times i_2\ ] \tag{67}$$

(Exercise: guess $distr$ and the inverse of $[\ \_, \_\ ]$).

Therefore, it will make sense to write finitary sums:

$$A_1 + \cdots + A_n \ \text{ that is } \ \sum_{i=1}^{n} A_i$$

and we have

$$\underbrace{A + \ldots + A}_{n} \ \cong \ n \times A \tag{68}$$

Mixing products and coproducts: the *exchange law*

Every arrow of type $B \times D \longleftarrow A + C$ can be expressed in two alternative ways:

$$[\ \langle f, g\rangle, \langle h, k\rangle\ ] \ = \ \langle [\ f, h\ ], [\ g, k\ ]\rangle \tag{69}$$

for $B \xleftarrow{f} A$, $D \xleftarrow{g} A$, $B \xleftarrow{h} C$ and $D \xleftarrow{k} C$.

Iso $undistr$ is an example of a function which is in the exchange-law format.

Conditionals

Given a predicate $\mathsf{Bool} \xleftarrow{p} A$, we may define its *guard*

$$A + A \xleftarrow{p?} A$$

as follows:

$$(p?)a \ = \ \begin{cases} p\,a & \Rightarrow \ i_1\,a \\ \neg(p\,a) & \Rightarrow \ i_2\,a \end{cases} \tag{70}$$

Then we introduce "McCarthy's conditional" combinator defined by

$$p \to g, h \ \stackrel{\text{def}}{=} \ [\ g, h\ ] \cdot p? \tag{71}$$

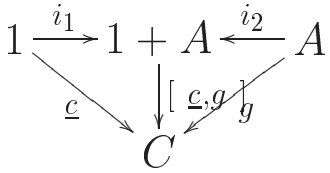
Several laws about conditionals can be derived from this definition, *e.g. McCarthy's conditional fusion law*:

$$f \cdot (p \to g, h) \;=\; p \to f \cdot g, f \cdot h \tag{72}$$

— a consequence of $+$-fusion (54).

Pointers

The $1 + A$ "pointer" datatype is "parsed" as follows:



In HASKELL, $1 + A$ can also be modelled by `Maybe A`, where

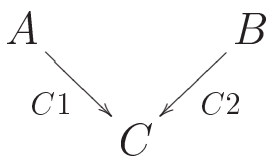
```
data Maybe a = Nothing | Just a
```

(Co)Product "costumization"

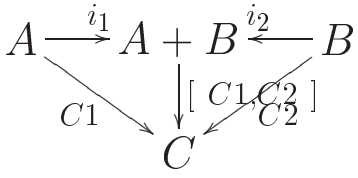
Abstract data models such as $A + B$ and $A \times B$ can be customized to particular situations by introducing "domain-specific" syntax. If one writes, in HASKELL

```
data C = C1 A | C2 B
```
(73)

one obtains a new datatype C equipped with constructors



— in fact the only ones available for constructing values of C:



Coproduct algebras/coalgebras

- $[\, C1, C2 \,]$ is called the **algebra** of datatype C, which contains the constructors of $C1$ and $C2$ of type C, that is, it is used to "build" C-values.
- In the opposite direction, inverse **co-algebra** inv enables us to "destroy" or observe values of C:

$$\begin{aligned} inv &: C \longrightarrow A + B \\ inv(C1\, a) &= i_1\, a \\ inv(C2\, b) &= i_2\, b \end{aligned}$$

- So $C1$ is a "renaming" of i_1, $C2$ is a "renaming" of i_2 and C is a "replica" of $A + B$:

$$C \underset{[\, C1,C2 \,]}{\overset{inv}{\cong}} A + B$$

Product algebras/coalgebras

Algebra/coalgebras can also be built around product datatypes. For instance,

```
data Point a = Point { x :: a, y :: a }
```

leads to the explicit introduction of selectors `x` and `y` renaming π_1 and π_2:

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times A & \xrightarrow{\pi_2} & A \\ & {\scriptstyle x}\nwarrow & \downarrow {\scriptstyle Point} & \nearrow{\scriptstyle y} & \\ & & Point & & \end{array} \qquad (74)$$

for some geometric type A. This rises algebra $Point$ and coalgebra $\langle x, y \rangle$ for datatype $Point$:

$$Point \underset{Point}{\overset{\langle x,y \rangle}{\cong}} A \times A$$

Be warned that HASKELL delivers `Point` in curried form: `Point :: a -> a -> Point a`

Examples

of costumized finite products:

```
data Component = Comp_Record {
        compID :: String,
        alarm  :: Int,
        cost   :: Float,
        cstock  :: Int
        }

data Equipment = Eq_Record {
        eqID          :: String,
        description   :: String,
        estock        :: Int
        }

data Part_of    = Part_of_Record {
        comp    :: String,
        equip   :: String,
        howManyC :: Int
```

```
        }

data Sub_Block_of = Sub_Block_of_Record {
        equipL   :: String,
        equipS   :: String,
        howManyE :: Int
        }
```

etc.

Functorial and natural properties in the toolboxes

"Functorial", *cf.* **functor** — what is that?

A brief introduction to functorhood

Where it all started: the "Manes & Arbib" approach [MA86]:

> E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.

A very well-known functor

$$\begin{array}{c} A \\ \downarrow f \\ B \end{array} \qquad \begin{array}{c} A^\star \\ \downarrow f^\star \\ B^\star \end{array}$$

where

$$f^\star\, l \stackrel{\text{def}}{=} [\, f\ a \mid a \leftarrow l\,]$$

is nothing but LISP "good-old" `mapcar` — *i.e.* `map f` in HASKELL:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Another functor: the powerset functor

$$\begin{array}{c} A \\ \downarrow f \\ B \end{array} \qquad \begin{array}{c} \mathcal{P}A \\ \downarrow \mathcal{P}f \\ \mathcal{P}B \end{array} \quad \text{where} \quad (\mathcal{P}f\,)s \stackrel{\text{def}}{=} \{f\ a \mid\ a \in s\}$$

Question:

"Why" are these things called "functors"? What do we gain from them?

Functors in general

$$\begin{array}{ccc} A & \cdots\cdots & \mathsf{F}\,A \\ {\scriptstyle f}\big\downarrow & & \big\downarrow{\scriptstyle \mathsf{F}\,f} \\ B & \cdots\cdots & \mathsf{F}\,B \end{array}$$

where

- F A: the A instantiation of a parametrical F-structure.
- F f: extends f to F-structures in a way such that two basic properties are met: it commutes with identity

$$\mathsf{F}\,id_A \;=\; id_{(\mathsf{F}\,A)} \tag{75}$$

and with composition

$$\mathsf{F}(g \cdot h) \;=\; (\mathsf{F}\,g) \cdot (\mathsf{F}\,h) \tag{76}$$

cf. e.g.

$$\begin{aligned} id^\star &= id \\ (f \cdot g)^\star &= f^\star \cdot g^\star \end{aligned}$$

(How special this was to write about LISP `mapcar` — more than 20 years ago...)

Two very basic functors:

- the **identity** functor (F $A = A$, F $f = f$)
- the **constant** functor (F $A = K, \mathsf{F}\,f = id_K$).

Bifunctors

These are just binary functors $\mathsf{B}\,(A, B)$ which preserve identities and composition in both arguments:

$$\mathsf{B}\,(id_A, id_B) \;=\; id_{\mathsf{B}\,(A,B)} \tag{77}$$

and

$$\mathsf{B}\,(g \cdot h, i \cdot j) \;=\; \mathsf{B}\,(g, i) \cdot \mathsf{B}\,(h, j) \tag{78}$$

In general: n-ary functors $\mathsf{N}(A_1, \ldots, A_n)$.

We have met two bifunctors earlier on:

- $\mathsf{B}\,(A, B) = A \times B$ and
- $\mathsf{B}\,(A, B) = A + B$.

Functor classes in HASKELL

Classes:

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)

class BiFunctor f where
      bmap :: (a -> b) -> (c -> d) -> (f a c -> f b d)
```

Instances:

```
instance BiFunctor (,) where
    bmap f g = f >< g

instance BiFunctor Either where
    bmap f g = f -|- g
```

Functor bridging: natural transformations

Consider the $elems$ (often called $setify$) function declaration:

$$\mathcal{P}\,A \xleftarrow{elems} A^\star$$

Because it bridges two functors — $\_\_^\star$ and $\mathcal{P}\_\_$ — a property pops up even before we provide its definition:

$$(\mathcal{P}\,f)\cdot elems \;=\; elems\cdot f^\star \tag{79}$$

cf. diagram

$$\begin{array}{ccc} A & \mathcal{P}\,A & \xleftarrow{elems} & A^\star \\ {\scriptstyle f}\downarrow & {\scriptstyle \mathcal{P}\,f}\downarrow & & \downarrow{\scriptstyle f^\star} \\ B & \mathcal{P}\,B & \xleftarrow[elems]{} & B^\star \end{array}$$

This is known as the *theorem for free* of the $elems$ function.

- **Quotation** (from *Theorems for free!*, by Philip Wadler [Wad89]):

  From the type of a polymorphic function we can derive a theorem that is satisfies. (...) *How useful are the theorems so generated? Only time and experience will tell* (...)

- **Comment**:

  Wadler was careful in disguising in traditional functional programming discourse a categorial result — **polymorphic** functions can be regarded as *natural transformations* between appropriate functors in an appropriate category-theoretical setting:

$$\begin{array}{ccc} A & \mathsf{G}\,A & \xleftarrow{\eta} & \mathsf{F}\,A \\ {\scriptstyle f}\downarrow & {\scriptstyle \mathsf{G}\,f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,f} \\ B & \mathsf{G}\,B & \xleftarrow[\eta]{} & \mathsf{F}\,B \end{array} \quad \textit{so} \quad (\mathsf{G}\,f)\cdot\eta = \eta\cdot(\mathsf{F}\,f)$$

Examples

The identity morphism $A \xleftarrow{id} A$ is a natural transformation bridging the identity functor and itself:

$$\begin{array}{ccc} A & A & \xleftarrow{id} & A \\ {\scriptstyle f}\downarrow & {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\ B & B & \xleftarrow[id]{} & B \end{array} \qquad \text{— recall (7)}$$

The constant morphism $K \xleftarrow{\underline{k}} A$ is natural too:

$$\begin{array}{ccc} A & K & \xleftarrow{\underline{k}} & A \\ {\scriptstyle f}\downarrow & {\scriptstyle id}\downarrow & & \downarrow{\scriptstyle f} \\ B & K & \xleftarrow[\underline{k}]{} & B \end{array} \qquad \text{— recall (5)}$$

"Free theorems" are *extremely* important due to their high degree of parametricity — recall the "natural" laws in the toolboxes earlier on.

Polynomial functors

We may put constant, product, coproduct and identity functors together to obtain so-called **polynomial functors**, which are described by polynomial expressions, for instance

$$\mathsf{F}\, X = 1 + A \times X \tag{80}$$

Definition

A polynomial functor is either

- a constant functor or the identity functor, or
- the (finitary) product or coproduct (sum) of other polynomial functors, or
- the composition of other polynomial functors.

□

So the effect on arrows of a polynomial functor is computed in an easy and structured way, for instance:

$$
\begin{array}{rcl}
\mathsf{F}\, f & = & (1 + A \times X) f \\
& = & \qquad \{ \text{ sum of two functors } \} \\
& & (1) f + (A \times X) f \\
& = & \qquad \{ \text{ constant functor and product of two functors } \} \\
& & id_1 + (A) f \times (X) f \\
& = & \qquad \{ \text{ constant functor and identity functor } \} \\
& & id_1 + id_A \times f \\
& = & \qquad \{ \text{ subscripts dropped for simplicity } \} \\
& & id + id \times f
\end{array}
$$

So, $1 + A \times f$ denotes the same as $id_1 + id_A \times f$, which abbreviates to $id + id \times f$.

Polynomial canonical form

Thanks to the isomorphism laws studied earlier on,

- every polynomial functor F can be put into the canonical

form,

$$\begin{array}{rcl} \mathsf{F}\,X & \cong & C_0 + (C_1 \times X) + (C_2 \times X^2) + \cdots + (C_n \times X^n) \\ & = & \Sigma_{i=0}^{n}\, C_i \times X^i \end{array} \tag{81}$$

- *Newton's binomial formula*

$$(A+B)^n \;\cong\; \sum_{p=0}^{n} {}^nC_p \times A^{n-p} \times B^p \tag{82}$$

can be used in such conversions (up to isomorphism.)

Example

Consider, for instance, functor

$$\mathsf{F}\,X \stackrel{\text{def}}{=} A \times (1+X)^2$$

(where A is a constant datatype) and check the following reasoning:

$$\begin{array}{rcl} \mathsf{F}\,X & = & A \times (1+X)^2 \\ & \cong & \quad \{\ \text{law (27)}\ \} \\ & & A \times ((1+X) \times (1+X)) \\ & \cong & \quad \{\ \text{law (64)}\ \} \\ & & A \times ((1+X) \times 1 + (1+X) \times X)) \\ & \cong & \quad \{\ \text{laws (28), (23) and (64)}\ \} \\ & & A \times ((1+X) + (1 \times X + X \times X)) \\ & \cong & \quad \{\ \text{laws (28) and (27)}\ \} \\ & & A \times ((1+X) + (X + X^2)) \\ & \cong & \quad \{\ \text{law (60)}\ \} \end{array}$$

$$
\begin{array}{ll}
 & A \times (1 + (X + X) + X^2) \\
\cong & \quad \{ \text{ canonical form obtained via laws (64) and (68) } \} \\
 & \underbrace{A}_{C_0} + \underbrace{A \times 2}_{C_1} \times X + \underbrace{A}_{C_2} \times X^2
\end{array}
$$

Epilogue

- Main reference: [BdM97].
- A “gentle introduction” is attempted in [Oli99a, Oli99b].

Collective Datatypes

Motivation

- Databases are large *collections* of records.
- Formally, a record is a member of some finite product, *e.g.* | 012.01.03 | 1200 | 1.5 | 2500 | in

```
data Component = Comp_Record {
        compID :: String,
        alarm  :: Int,
        cost   :: Float,
        cstock  :: Int
        }
```

- How do we model a **collective** datatype, *e.g.* the one which

compID	**alarm**	**cost**	**cstock**
012.01.03	1200	1.5	2500
⋮	⋮	⋮	⋮
914.15.30	200	10.5	500

  is a member of?

Programming approach

Polymorphic **finite lists** ($\forall A.A^\star$) are among the most popular programming devices for collecting data, as built-in in HASKELL:

```
-- data [a] = [] | a : [a]
```

So we might define

```
type Collection = [ Component ]
```

or define, on purpose

```
data Collection = Empty |
                  Insert (Component , Collection)
```

It is easy to abstract from the algebras implicit in both approaches — resp. $[\,\underline{[\,]}, uncurry(:)\,]$ and $[\,\underline{Empty}, Insert\,]$ — the following linear domain **equation**:

$$x \;=\; 1 + Component \times x \tag{83}$$

Linguistic approach

Start from the following GF-grammar fragment:

$$\begin{aligned}
G &= (NT, T, \langle \textit{Collection} \rangle, P) \\
NT &= \{\langle \textit{Collection} \rangle, \langle \textit{Component} \rangle\} \\
T &= \{\, \texttt{Empty} \,,\, \texttt{Insert} \,,\, \texttt{into} \,\} \\
P &= \left\{ \begin{array}{lll} \langle \textit{Collection} \rangle & ::= & \texttt{Empty} \mid \\ & & \texttt{Insert}\ \langle \textit{Component} \rangle\ \texttt{into}\ \langle \textit{Collection} \rangle \end{array} \right.
\end{aligned}$$

and submit it a kind of "transformation":

Given some CFG $G = (N, T, S, P)$*,*

- *eliminate all terminal symbols everywhere in* P*;*
- *keep every non-terminal symbol* $\langle$A$\rangle$ *as variable* A *and infer an equation from each production in* P*, ac-*

cording to

BNF		ABSTRACT NOTATION
$\alpha \mid \beta$	$\mapsto$	$\alpha + \beta$
$\alpha\ \beta$	$\mapsto$	$\alpha \times \beta$
ϵ	$\mapsto$	1
α^*	$\mapsto$	α^*
α^+	$\mapsto$	$\alpha \times \alpha^*$

One gets

$$Collection \ = \ 1 + Component \times Collection$$

easily recognizable as a “renaming” of equation (83).

Comments:

Of course! The *abstract-syntax* tree of a CFG valid sentence is a (recursive) data-structure.

So we have reached the same linear, **inductive** structure from different programming paradigms.

Polynomial collective types

- The “=” symbol in equation (83) should be understood as “$\cong$”
- So any solution X to the equation should carry along an algebra in and its inverse out thus providing evidence of the required *isomorphism*:

$$X \underset{in}{\overset{out}{\cong}} 1 + Component \times X$$

For instance,

$$Collection \underset{in=[\ \underline{Empty},Insert\ ]}{\overset{out}{\underset{\leftarrow}{\overset{\rightarrow}{\cong}}}} 1 + Component \times Collection$$

where out is the obvious inverse of in.

Equation (83) is an instance of

$$X \ \cong \ \mathsf{F}\, X$$

for $\mathsf{F}\, X = 1 + A \times X$, and $A = Component$.

Fact:

For F a polynomial functor, equation $X \cong \mathsf{F}\, X$

- Admits a standard solution — its **least fixpoint** solution

$$\mu\mathsf{F} \underset{in}{\overset{out}{\underset{\leftarrow}{\overset{\rightarrow}{\cong}}}} \mathsf{F}\, \mu\mathsf{F}$$

  which is *syntactic in nature*.

- $\mu\mathsf{F}$ is *initial* among all other F-structures — that is to say, for a given $(A,\ A \xleftarrow{\alpha} \mathsf{F}\, A)$, arrow k in

$$\begin{array}{ccc} \mu\mathsf{F} & \xleftarrow{in} & \mathsf{F}\, \mu\mathsf{F} \\ {\scriptstyle k}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\, k} \\ A & \xleftarrow[\alpha]{} & \mathsf{F}\, A \end{array}$$

  is unique.

Notation:

To express this uniqueness of k, dependent on α, we write $(\!|\alpha|\!)$ — read "α-catamorphism" (from $\kappa\alpha\tau\alpha$ = "downwards") — and

state the following **universal property**:

$$k = (\!|\alpha|\!) \;\Leftrightarrow\; k \cdot in = \alpha \cdot \mathsf{F}\, k$$

Any structure $(A,\ A \xleftarrow{\alpha} \mathsf{F}\, A)$ is said to be an F-algebra.

Important:

So we get another "toolbox", **the initial F-algebra toolbox**, this time involving F-algebras.
The arrows which are relevant in this context are the F-algebra *homomorphisms*:

$$\begin{array}{ccc} A & \xleftarrow{\alpha} & \mathsf{F}\, A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\, f} \\ B & \xleftarrow[\beta]{} & \mathsf{F}\, B \end{array}$$

Therefore:

Cata-cancellation : $$(\!|\alpha|\!) \cdot in = \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \tag{84}$$

Cata-reflection : $$(\!|in|\!) = id \tag{85}$$

$$\begin{array}{ccc} \mu\mathsf{F} & \xleftarrow{in} & \mathsf{F}\,\mu\mathsf{F} \\ {\scriptstyle (\!|in|\!)}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|in|\!)} \\ \mu\mathsf{F} & \xleftarrow[in]{} & \mathsf{F}\,\mu\mathsf{F} \end{array}$$

Cata-fusion : $$f \cdot (\!|\alpha|\!) = (\!|\beta|\!) \quad \text{if} \quad f \cdot \alpha = \beta \cdot \mathsf{F}\, f \tag{86}$$

$$\begin{array}{ccc} \mu\mathsf{F} & \xleftarrow{in} & \mathsf{F}\,\mu\mathsf{F} \\ {\scriptstyle (\!|\alpha|\!)}\downarrow & & \downarrow{\scriptstyle \mathsf{F}(\!|\alpha|\!)} \\ A & \xleftarrow[\alpha]{} & \mathsf{F}\, A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\, f} \\ B & \xleftarrow[\beta]{} & \mathsf{F}\, B \end{array}$$

What about **Cata-absorption**? Where is the associated functor?

Type functors

In writing

$$\mu \mathsf{F} \;\cong\; \mathsf{F}\,\mu \mathsf{F}$$

for $\mathsf{F}\,X = 1 + A \times X$ we don't want A to be regarded as a **constant** datatype. So we should give room for A as a parameter and write

$$\mu \mathsf{F}\,A \;\cong\; \mathsf{B}(A, \mu \mathsf{F}\,A)$$

where $\mathsf{B}(X, Y) = 1 + X \times Y$.

In general, datatypes "are" functors built on top of other (**base**) functors:

$$\underbrace{\mathsf{T}A}_{\text{type functor}} \quad \cong \quad \underbrace{\mathsf{B}(A, \mathsf{T}A)}_{\text{base functor}}$$

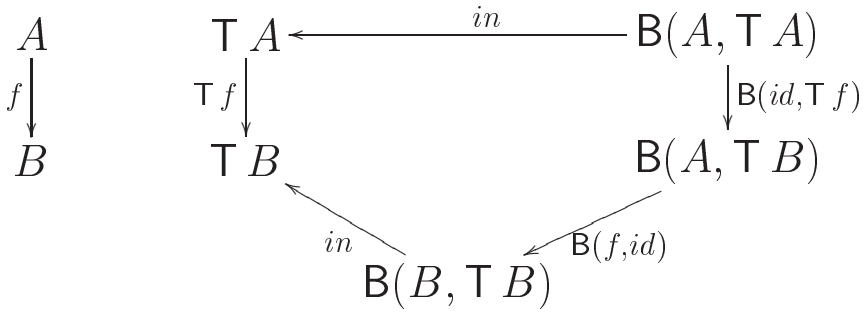
for

$$\mathsf{T}\,A \cong \mu X.\mathsf{B}(A, X)$$

Example

$$\mathsf{BTree}\,A \;\cong\; \underbrace{1 + A \times (\mathsf{BTree}\,A)^2}_{\substack{\text{polynomial} \\ \text{base functor} \\ \mathsf{B}(X,Y) = 1 + X \times Y^2}}$$

Type functor definition:



So:

$$\mathsf{T}\, f \;=\; (\!|\, in \cdot \mathsf{B}(f, id) \,|\!)$$

Then we have:

Cata-absorption :	$(\!|\alpha|\!) \cdot \mathsf{T}\, f \;=\; (\!|\alpha \cdot \mathsf{B}(f, id)|\!)$	(87)
Cata-functor :	$\mathsf{T}\,(f \cdot g) \;=\; \mathsf{T}\, f \cdot \mathsf{T}\, g$	(88)
Cata-functor-id :	$\mathsf{T}\, id_A \;=\; id_{\mathsf{T}\, A}$	(89)

Further properties

The *mutual-recursion law*, also called “Fokkinga law”:

$$\begin{array}{l} f \cdot in = h \cdot \mathsf{F}\,\langle f, g\rangle \\ \qquad\qquad \wedge \qquad\qquad \Rightarrow\; \langle f, g\rangle = (\!|\langle h, k\rangle|\!) \\ g \cdot in = k \cdot \mathsf{F}\,\langle f, g\rangle \end{array} \tag{90}$$

From (90), we get the so-called “banana-split” law:

$$\langle (\!|i|\!), (\!|j|\!)\rangle = (\!|\langle i \cdot \mathsf{F}\,\pi_1, j \cdot \mathsf{F}\,\pi_2\rangle|\!)$$

that is

$$\langle (\!|i|\!), (\!|j|\!)\rangle = (\!|(i \times j) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2\rangle|\!) \tag{91}$$

by (reverse) $\times$-absorption (18). This law provides us with a very useful tool for “parallel loop” inter-combination.

Regular functors and types

Type functors can participate in the construction of other types, leading to **regular types** as, for instance, a datatype of formal expressions

$$Exp \cong Var + Op \times Exp^{\star}$$

whose base-functor

$$\mathsf{B}(A, B, X) = A + B \times X^{\star}$$

is regular.

Fact:

The theory above extends to **regular types**.

An example of pointfree datatype definition in HASKELL

Binary trees:

```
-- (1) Datatype definition --------------------


data BTree a = Empty | Node(a, (BTree a, BTree a))

-- (2) Initial algebra / coalgebra ------------
```

```
inBTree ::
     Either () (b,(BTree b,BTree b)) -> BTree b
inBTree = either (const Empty) Node

outBTree ::
     BTree a -> Either () (a,(BTree a,BTree a))
outBTree Empty                = Left ()
outBTree (Node (a,(t1,t2))) = Right(a,(t1,t2))

-- (3) Base functor ---------------------------


baseBTree f g = id -|- (f >< (g >< g))

-- (4) Catamorphism ---------------------------


cataBTree g =
       g . (recBTree (cataBTree g)) . outBTree

recBTree f = baseBTree id f

-- (5) Type functor ---------------------------


instance Functor BTree
    where fmap f = cataBTree
                     ( inBTree . baseBTree f id )

-----------------------------------------------
```

Unorderd collective types

Both polynomial or regular inductive types suffer from a drawback in modelling data collections: they retain "two much information" — *e.g.* record **order** and **duplication** — which very often is meaningless in data modelling.

Recall from above that an inductive datatype is defined up to isomomphism

$$\mathsf{T} \underset{in}{\overset{out}{\cong}} \mathsf{F}\,\mathsf{T}$$

and that from its definition two isomorphisms emerge — algebra in and coalgebra out — which are each other's inverses.

The powerset collective type

Not every datatype is inductive in this way. Think for instance of the *powerset* datatype $\mathcal{P}A$ containing all subsets of a finite set A

$$\mathcal{P}A \;=\; \{s \mid\; s \subseteq A\}$$

and define algebra $\mathcal{P}A \xleftarrow{ins} 1 + A \times \mathcal{P}A$, where

$$ins \;\stackrel{\text{def}}{=}\; [\,\underline{\emptyset}, puts\,] \tag{92}$$

$$puts \;\stackrel{\text{def}}{=}\; \cup \cdot (sings \times id) \tag{93}$$

$$sings\, x \;\stackrel{\text{def}}{=}\; \{x\} \tag{94}$$

Conversely, any $\emptyset \subset s \subseteq A$ can always be decomposed into pair $(a, s - \{a\})$ for some $a \in A$. So we can think of

$$\mathcal{P}A \xrightarrow{outs} 1 + A \times \mathcal{P}A$$

where

$$outs = (! + gets) \cdot (=_{\emptyset})? \tag{95}$$

and where, for nonempty s, we define:

$$\begin{array}{rcll} gets\, s & \stackrel{\text{def}}{=} & let & a \in s \\ & & in & (a, s - \{a\}) \end{array} \tag{96}$$

Comments:

- $\mathcal{P}A$ is similar to $A^\star$: both share the same inductive "shape", defined by functor $\mathsf{F}\, X = 1 + A \times X$.
- Major distinction: $outs$ is not a left inverse of ins on $\mathcal{P}A$, that is,

  $$outs \cdot ins \;\neq\; id$$

  in general.

  Because of the **unordered** structure of a set s, one cannot reconstruct the steps along which it was built.

  So a powerset is not inductively generated, *i.e.* it is *non initial*.

- Nevertheless,

  $$ins \cdot outs \;=\; id$$

  holds, so

  $$\mathcal{P}A \underset{ins}{\overset{outs}{\leftrightarrows}} 1 + A \times \mathcal{P}A \tag{97}$$

We have glossed over the nondeterminacy of the choice of a in $gets$. Striclty speaking, we should move to a **relational** approach to inductive datatypes, as in *e.g.* [BH93].

Comment:

The lack of in/out invertibility prevents us from defining catamorphisms over $\mathcal{P}A$: only $outs$-based **hylomorphisms**

$$\begin{array}{ccc} \mathcal{P}A & \xrightarrow{outs} & 1 + A \times \mathcal{P}A \\ {\scriptstyle [\![outs,g]\!]}\downarrow & & \downarrow{\scriptstyle id+id\times[\![outs,g]\!]} \\ B & \xleftarrow[g]{} & 1 + A \times B \end{array}$$

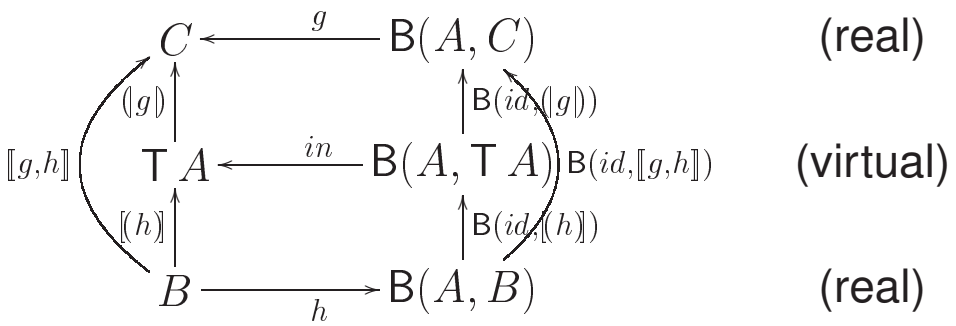
can be defined over some suitable target $(1 + A \times \_)$-algebra g.

About hylomorphisms

A hylomorphism $[\![g, h]\!]$ is the combination of two morphisms, a catamorphism and a so-called *anamorphism* (the dual of a cata):

$$[\![g, h]\!] = (\!|g|\!) \cdot [\!(h)\!]$$

The hylomorphism is a very general programming concept:



in which an intermediate "virtual" inductive type $\mathsf{T}\,A$ controls the algorithmic process.

Anamorphisms bring in the problem of nontermination, which is solved in our case by restricting to **well-founded** coalgebras h — see details in [Oli00].

Notation:

We will adopt notation $\{\!|g|\!\}$ as an abbreviation of $(\!|g|\!) \cdot [\!(outs)\!]$ over "virtual type" $A^\star$:

$$\{\!|g|\!\} \stackrel{\text{def}}{=} (\!|g|\!) \cdot [\!(outs)\!] \tag{98}$$

Due to the nondeterminism of $outs$ (*cf.* $gets$), g must be insensitive to the order of A values picked up by $gets$. So, for $g = [\ g_1, g_2\ ]$:

$$g_2(a, g_2(a', b)) \;=\; g_2(a', g_2(a, b)) \tag{99}$$

must hold.

Properties of $\{\!|g|\!\}$

Which properties of catamorphisms extend to $\{\!|g|\!\}$?

No universal property, but...

$\mathcal{P}$-fusion:

$$f \cdot \{\!|g|\!\} = \{\!|h|\!\} \;\Leftarrow\; f \cdot g = h \cdot (id + id \times f) \tag{100}$$

$\mathcal{P}$-reflection:

$$\{\!|ins|\!\} \;=\; id_{(\mathcal{P}\,A)} \tag{101}$$

Powerset **cancellation**, however, will no longer remain valid:

$$\{\!|g|\!\} \cdot ins \;=\; g \cdot (id + id \times \{\!|g|\!\}) \tag{102}$$

does **not** hold in general.

Fact:

$$(102) \;\Rightarrow\; \{\!|g|\!\} \cdot elems = (\!|g|\!) \tag{103}$$

where

$$elems \stackrel{\text{def}}{=} (\!|ins|\!) \tag{104}$$

is the function which "setifies" a finite list.

Example

The powerset (type) functor

$$\mathcal{P}f \;=\; \{\!|ins \cdot (id + f \times id)|\!\} \tag{105}$$

— *i.e.* pointwise set-theoretical comprehension

$$(\mathcal{P}f)s \;=\; \{f\, a \mid\; a \in s\}$$

— satisfies (102):

$$\begin{array}{ll} & \mathcal{P}f \cdot ins = (ins \cdot (id + f \times id)) \cdot (id + id \times \mathcal{P}f) \\ \Leftrightarrow & \quad \{ + \text{ and } \times \text{ functors and identity}\} \\ & \mathcal{P}f \cdot ins = ins \cdot (id + f \times \mathcal{P}f) \\ \Leftrightarrow & \quad \{ \text{ free theorem of polymorphic } ins \} \\ & TRUE \end{array}$$

Powerset-absorption

$$\{\!|g|\!\} \cdot (\mathcal{P}f) \;=\; \{\!|g \cdot (id + f \times id)|\!\} \tag{106}$$

is valid only to algebras g satisfying (102) (details in [Oli00]). Then:

$$\begin{aligned}(\mathcal{P}g)\cdot(\mathcal{P}f) &= \mathcal{P}(g\cdot f)\\ \mathcal{P}id &= id\end{aligned}$$

A well-known isomorphism

$$\mathcal{P}A \underset{zf}{\overset{cf}{\cong}} 2^A \tag{107}$$

where $2 \cong \mathsf{Bool}$ and

$$\begin{aligned}zf\ \phi &\stackrel{\text{def}}{=} \{a \in A \mid \phi\, a\}\\ cf\ s &\stackrel{\text{def}}{=} \lambda a.a \in s\end{aligned}$$

Some operators

$$\#\ \stackrel{\text{def}}{=}\ \{\!|\,[\ \underline{0}, succ\cdot\pi_2\ ]\,|\!\}\ \ \text{/*cardinal*/} \tag{108}$$

$$s \cup r\ \stackrel{\text{def}}{=}\ \{\!|\,[\ \underline{s}, puts\ ]\,|\!\}\ \ \text{/*union*/} \tag{109}$$

$$\bigcup\ =\ \{\!|\,[\ \underline{\emptyset}, \cup\ ]\,|\!\}\ \ \text{/*distributed union*/} \tag{110}$$

$$filter(f,p)\ \stackrel{\text{def}}{=}\ \bigcup\cdot\mathcal{P}(p \to sings\cdot f, \underline{\emptyset}) \tag{111}$$

$$the\{a\}\ \stackrel{\text{def}}{=}\ a\ \ \text{/*otherwise undefined*/} \tag{112}$$

ZF-set-abstraction (=set-comprehension)

$$\{f\ a \mid\ a \in s \wedge (p\, a)\}\ =\ filter(f,p)$$

Binary relations and finite mappings

These are two particularly useful specializations of the power-set datatype:

- **Finite binary relations** are inhabitants of datatype $\mathcal{P}(A \times B)$, for finite A and B.
- **A finite mapping** is a finite binary relation subject to

$$fdp\, r \stackrel{\text{def}}{=} \forall a \in A : \#\left((collect\, r)\, a\right) \leq 1$$

  where $collect$ is isomorphism

$$\mathcal{P}(A \times B) \underset{discollect}{\overset{collect}{\cong}} (\mathcal{P}B)^A \tag{113}$$

  which stems from (107).

fdp means "functional dependence", a very important concept in information modelling

Notation:

- **Restricted datatypes**: datatypes involving restrictions (such as fdp) will be denoted by A_ϕ, for restriction $Bool \xleftarrow{\phi} A$, meaning:

$$A_\phi \stackrel{\text{def}}{=} \{a \in A \mid \phi\, a\} \tag{114}$$

- **Partial mappings**: The finite mapping data model (restricting binary relations) is so common that deserves special notation:

$$A \rightharpoonup B \stackrel{\text{def}}{=} \mathcal{P}(A \times B)_{fdp} \tag{115}$$

  To emphasize its functional semantics, every entry (pair) of a finite mapping (a, b) will be written $a \mapsto b$.

Datatype invariants and partial algebras

Restrictions such as fdp are called *datatype invariants* because they are properties which "remain invariant" along the $outs$ coalgebra:

$$A \rightharpoonup B \underset{ins}{\overset{outs}{\leq}} 1 + (A \times B) \times A \rightharpoonup B \tag{116}$$

(*i.e.* any subset of a finite mapping will satisfy fdp).

In the context of $A \rightharpoonup B$, $outs$ is a partial algebra! For instance ($A = B = \mathbb{N}$), $puts(1 \mapsto 2, \{1 \mapsto 1\})$ will fall outside $A \rightharpoonup B$.

Operators over finite mappings

$$\begin{array}{l} dom \;::\; A \rightharpoonup B \longrightarrow \mathcal{P}A \;\; \textit{/* finite mapping domain */} \\ dom \stackrel{\text{def}}{=} \{\!|ins \cdot (id + \pi_1 \times id)|\!\} \end{array} \tag{117}$$

$$\begin{array}{l} rng \;::\; A \rightharpoonup B \longrightarrow \mathcal{P}B \;\; \textit{/* finite mapping range */} \\ rng \stackrel{\text{def}}{=} \{\!|ins \cdot (id + \pi_2 \times id)|\!\} \end{array} \tag{118}$$

$$\begin{array}{l} A \rightharpoonup \_ \;::\; C^B \longrightarrow A \rightharpoonup B \longrightarrow A \rightharpoonup C \;\; \textit{/* type functor */} \\ A \rightharpoonup f \;=\; \{\!|ins \cdot (id + (id \times f) \times id)|\!\} \end{array} \tag{119}$$

$$\begin{array}{l} | \;::\; \mathcal{P}A \times (A \rightharpoonup B) \longrightarrow A \rightharpoonup B \\ |s \stackrel{\text{def}}{=} filter(id, (\in s) \cdot \pi_1) \;\; \textit{/* domain restriction */} \end{array} \tag{120}$$

$$\begin{array}{lcl} \setminus & :: & \mathcal{P}A \times (A \rightharpoonup B) \longrightarrow A \rightharpoonup B \\ \setminus s & \stackrel{\text{def}}{=} & filter(id, (\notin s) \cdot \pi_1) \quad \textit{/* domain subtraction */} \end{array} \tag{121}$$

$$\begin{array}{lcl} \dagger & :: & (A \rightharpoonup B) \times (A \rightharpoonup B) \longrightarrow A \rightharpoonup B \\ \dagger & \stackrel{\text{def}}{=} & \cup \cdot \langle (\setminus) \cdot (dom \times id), \pi_2 \rangle \quad \textit{/* mapping overwrite */} \end{array} \tag{122}$$

$$\begin{array}{lcl} pap & :: & (A \rightharpoonup B) \times A \longrightarrow B \\ pap & \stackrel{\text{def}}{=} & the \cdot rng \cdot | \cdot (sings \times id) \cdot swap \quad \textit{/* application */} \end{array} \tag{123}$$

For monic (injective) f one also has

$$f \rightharpoonup B \;=\; \{|ins \cdot (id + (f \times id) \times id)|\} \tag{124}$$

and, of course, the binary map operator:

$$f \rightharpoonup g \;=\; \{|ins \cdot (id + (f \times g) \times id)|\} \tag{125}$$

Notation:

Wherever defined, $pap(\sigma, a)$ abbreviates to $\sigma\, a$.

Properties of finite mappings

Finite mappings are finite "approximations" of exponentials — they behave like functions which have been partly "tabulated", *e.g.*

$$\{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6\}$$

is a finite approximation of $n!$.

Fact:

A finite (partial) mapping converts to a total function by introduction of an explicit "error" value. In fact:

$$A \rightharpoonup B \underset{untot}{\overset{tot}{\cong}} (B+1)^A \tag{126}$$

where

$$\begin{aligned} tot\,\sigma &\stackrel{\text{def}}{=} \lambda a.\ if\ \ a \in dom\,\sigma \\ &\qquad\quad then\ \ i_1(\sigma\,a) \\ &\qquad\quad else\ \ i_2\,() \\ untot\,f &\stackrel{\text{def}}{=} \{a \mapsto b |\ a \in A \wedge f\,a = (i_1 b)\} \end{aligned}$$

Question:

Which properties of exponentials are "preserved" by finite partial mappings?

- (63) holds:

$$\begin{aligned} 0 \rightharpoonup A &\cong (A+1)^0 \\ &\cong 1 \end{aligned} \tag{127}$$

- (40) becomes:

$$\begin{aligned} 1 \rightharpoonup A &\cong (A+1)^1 \\ &\cong A+1 \end{aligned} \tag{128}$$

- (65) holds:

$$\begin{aligned} (B+C) \rightharpoonup A &\cong (A+1)^{B+C} \\ &\cong (A+1)^B \times (A+1)^C \\ &\cong (B \rightharpoonup A) \times (C \rightharpoonup A) \end{aligned} \tag{129}$$

So

$$(B+C) \rightharpoonup A \underset{peither}{\overset{unpeither}{\cong}} (B \rightharpoonup A) \times (C \rightharpoonup A) \tag{130}$$

where

$$peither \stackrel{\text{def}}{=} \cup \cdot ((i_1 \rightharpoonup A) \times (i_2 \rightharpoonup A))$$

(Exercise: guess $unpeither$).

- (39) becomes:

$$\begin{aligned} A \rightharpoonup 1 &\cong (1+1)^A \\ &\cong \mathcal{P}A \end{aligned} \tag{131}$$

The remaining laws (41) and (42) do not generalize to isomorphisms (left-invertibility (2) is lost):

- Concerning (41):

$$A \rightharpoonup (B \times C) \underset{\bowtie}{\overset{unjoin}{\underset{\leq}{\rightleftarrows}}} (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{132}$$

where

$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \{a \mapsto (\sigma\, a, \tau\, a) \mid a \in dom\, \sigma \cap dom\, \tau\} \tag{133}$$

(read "join") is the finite mapping counterpoint of the $unzip$ operator [Jan00] on finite lists, and

$$unjoin \stackrel{\text{def}}{=} \langle A \rightharpoonup \pi_1, A \rightharpoonup \pi_2 \rangle \tag{134}$$

- Concerning (42):

$$(C \times A) \rightharpoonup B \underset{unpcurry}{\overset{pcurry}{\underset{\leq}{\rightleftarrows}}} C \rightharpoonup (A \rightharpoonup B)$$

where

$$\begin{aligned} &pcurry\, f \stackrel{\text{def}}{=} \\ &\quad \{a \mapsto sel(f, a) \mid a \in \mathcal{P}(\pi_1)(dom\, f)\} \end{aligned} \tag{135}$$

(for $sel(f,a) \stackrel{\text{def}}{=} \{\pi_2\, p \mapsto f\ p \mid p \in dom\ f \wedge (\pi_1\, p) = a\}$) and

$$unpcurry\, \sigma \stackrel{\text{def}}{=} \{(a,b) \mapsto (\sigma\, a)b \mid a \in dom\, \sigma \wedge b \in dom\, (\sigma\, a)\}$$

Clearly,

$$pcurry \cdot unpcurry \ \neq \ id$$

This "$\leq$-subcalculus" encompasses other $\rightharpoonup$-laws which cannot be adapted from similar results concerning exponentials. For instance,

$$(B + C)^A \ncong B^A \times B^C$$

and yet we have the following map-decomposition law:

$$A \rightharpoonup (B + C) \underset{\overset{+}{\bowtie}}{\overset{uncojoin}{\underset{\leq}{\rightleftarrows}}} (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{136}$$

where

$$\overset{+}{\bowtie} \stackrel{\text{def}}{=} \dagger \cdot ((A \rightharpoonup i_1) \times (A \rightharpoonup i_2))$$

Other standard representations

For finite A:

$$\mathcal{P}A \ \leq \ A \rightharpoonup B \tag{137}$$

(representation consists of classifying data, $b_0 \in B$).

$$\mathcal{P}A \ \leq \ \mathbb{N} \rightharpoonup A \tag{138}$$

(representation consists of indexing data).

$$A^\star \leq \mathbb{N} \rightharpoonup A \tag{139}$$

(refinement consists of rendering sequence order explicit).

$$A \rightharpoonup B \leq B \rightharpoonup \mathcal{P} A \tag{140}$$

(refinement consists of representing a function by its "kernel").

Powersets and finite mappings in HASKELL

One may define

```
data Set a = Empty | Put (a , Set a)
```

— which is (syntactically) isomorphic to `[a]` — and handle this non-initially by

```
inSet :: Eq a => Either b (a,Set a) -> Set a
inSet x = either (const Empty) (uncurry puts) x
```

(note the need for `a` an equality type) and its inverse

```
outSet :: Set a -> Either () (a,Set a)
outSet Empty       = Left ()
outSet (Put(a,x)) = Right(a,x)
```

where

```
puts :: Eq a => a -> Set a -> Set a
puts a s = if a |- s then s else Put(a,s)
```

for membership test

```
a |- Empty = False
a |- (Put (b,s)) =
           if a==b then True else a |- s
```

Alternatively, HASKELL lists can be used directly,

```
data Set a = Set [a]
```

taking advantage of HASKELL's standard libraries, as *e.g.* in

```
puts :: Eq a => a -> Set a -> Set a
puts a (Set s) = Set (union [a] s)
```

Equality can no longer be *derived* because it no longer is syntactic. We have to define

```
instance Eq a => Eq (Set a)
        where s == r = s <= r && r <= s

instance Eq a => Ord (Set a)
        where (Set s) <= r = all (-| r) s
```

Similarly for finite mappings:

```
data Fmap a b = Fmap [(a,b)]

instance (Eq a, Eq b) => Ord (Fmap a b) where
    f <= g =
          (dom f) <= (dom g) && compatible f g

instance (Eq a, Eq b) => Eq (Fmap a b) where
    f == g = f <= g && g <= f
```

Modelling with finite mappings

Multisets

The notion of an unordered data collection in which multiplicity is relevant is captured by a *multiset*:

$$\mathsf{Mset}\, A \stackrel{\text{def}}{=} A \rightharpoonup \mathbb{N}$$

Comments:

- Multisets are “half way” between finite list and finite sets:

  $$2^A \leq \mathsf{Mset}\, A \leq A^\star$$

  (Multiplicity is relevant but order is not.)
- Finite set algebra can be extended to multisets, for instance

by defining **multiset union**:

$$v \oplus r \stackrel{\text{def}}{=} v \dagger r \dagger \{a \mapsto v\, a + r\, a \mid a \in (dom\, v) \cap (dom\, r)\}$$

Structure $(\mathsf{Mset}\, A; \oplus, \emptyset)$ is a monoid and so finite multiset union $\oplus_{x \in \ldots}$ makes sense.

- Further operators can be defined, *e.g.*

$$n \otimes v \;=\; \underbrace{v \oplus \cdots \ldots \oplus v}_{n \text{ times}}$$

NB:

- If we identify A with the dimensions of a **vector space**, then each $v \in \mathsf{Mset}\, A$ may be understood as a vector.
- For instance, 3-dimensional vector $v = 2x + 3y + z$ described by multiset $m = \begin{pmatrix} x & y & z \\ 2 & 3 & 1 \end{pmatrix}$.
- Under the analogy, $\oplus$ becomes **vectorial sum** and $\otimes$ becomes **scalar** (external) multiplication.
- In later calculations will require $\otimes$ to distribute over $\oplus$, *etc.*

□

A multiset model of "bill-of-materials"

The data model underlying the "bill-of-materials" problem stated earlier on (p.4) can now bw identified as a recursive ("hierarchical") multiset

$$\mathsf{Hmset}\, A \;\cong\; \mathsf{Mset}\, (A + \mathsf{Hmset}\, A)$$

cf. equation

$$X \;\cong\; (A + X) \multimap \mathbb{N} \tag{141}$$

Specifying part explosion

The following specification of $explode$ exhibits a "flattening" effect (hierarchical multisets converted to flat multisets):

$$\begin{array}{rcl} explode & :: & \mathsf{Hmset}\, A \longrightarrow \mathsf{Mset}\, A \\ explode\, \sigma & \stackrel{\text{def}}{=} & \oplus_{x \in dom\, \sigma} \left\{ \begin{array}{lcl} (x = i_1\, a) & \Rightarrow & \{a \mapsto \sigma\, x\} \\ (x = i_2\, \sigma') & \Rightarrow & (\sigma\, x) \otimes explode\, \sigma' \end{array} \right. \end{array} \quad (142)$$

$explode$ can be also viewed as a **data-mining** operation where *data consolidation* is performed over the $(\mathsf{Mset}\, A; \oplus, \otimes)$ "vector space".

In HASKELL

```
type Mset a  = Fmap a Int

data Hmset a  = In (Mset (Either a (Hmset a)))
                deriving Show

instance (Eq a) => Eq (Hmset a)
          where (In f) == (In g)  = f == g

explode :: (Eq a)  => Hmset a -> Mset a
explode (In sigma) =
oPlus([ case x of
     Left  a -> wrap a (pap sigma x)
     Right b -> otimes (pap sigma x) (explode b)
      | x <- get (dom sigma) ])
```

assuming some predefined notation):

```
wrap a b = Fmap [ a |-> b ]

(<+>) :: Eq a => Mset a -> Mset a -> Mset a
(<+>) = monfm (+)

otimes :: Int -> Mset b -> Mset b
otimes b (Fmap l) = Fmap [ (a,b*n) | (a,n) <- l ]

oPlus :: Eq a => [Mset a] -> Mset a
oPlus x = foldl (<+>) bottom x
```

etc.

A semantics for a tiny subset of UML class diagrams

- One of the problems of data modelling by diagrams — such as with the ***UML*** [BRJ99] — is lack of precision: what do the drawings mean?
- There are attempts to identify subsets of the ***UML*** — *e.g.* the **PUML** (=*Precise UML*) [BF98] — bearing formal semantics.
- Below we sketch an exercise in this direction.

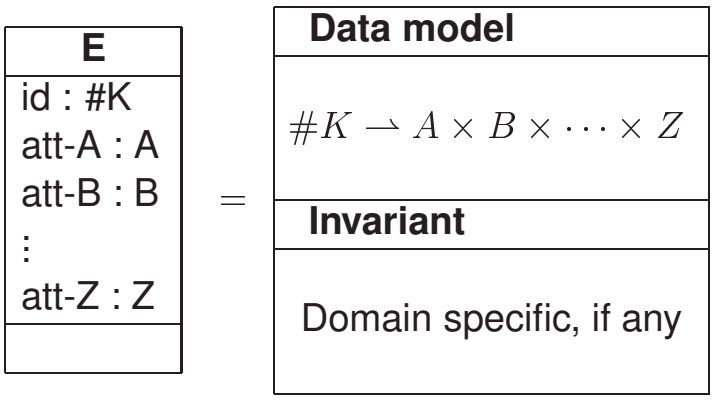
Examples of translation rules

In the diagrams which follow, symbols **E**, **F** denote UML classes; symbol $\#K$ denotes a denumerable domain of keys (typically, $\#K \cong \mathbb{N}$); and A, ..., Z denote arbitrary attribute domains. Each rule is of the form

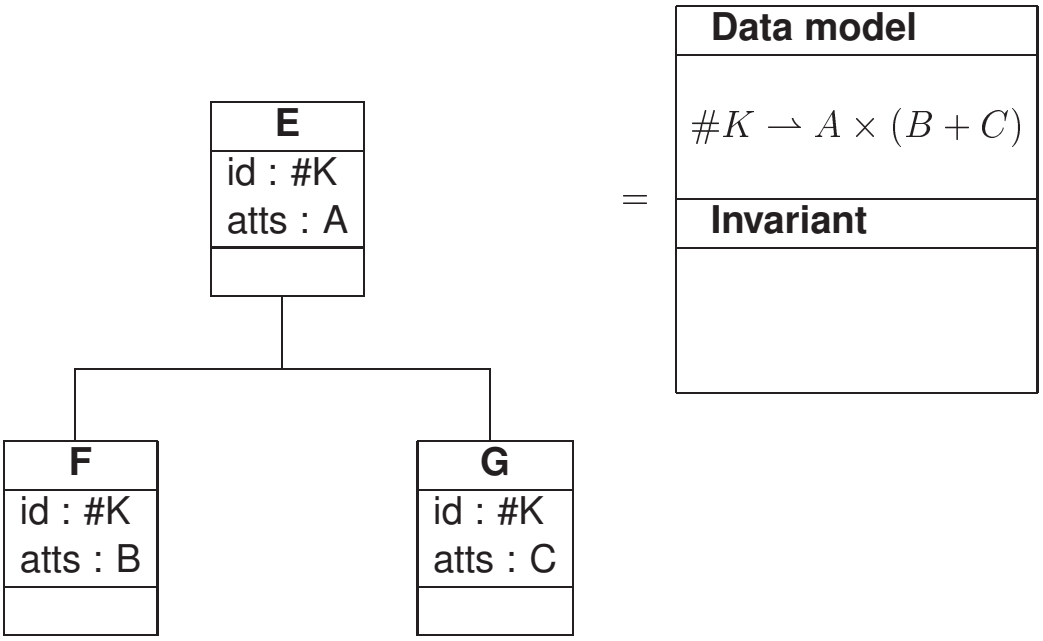
$$\delta = \sigma$$

where δ is a diagram and σ is the corresponding formal semantics — a collection of finite mappings + invariants ensuring *referential integrity*.

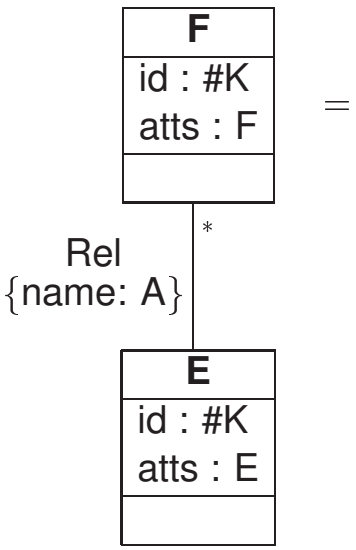
Class declaration



Sub-classing



N:1 Relationships



=

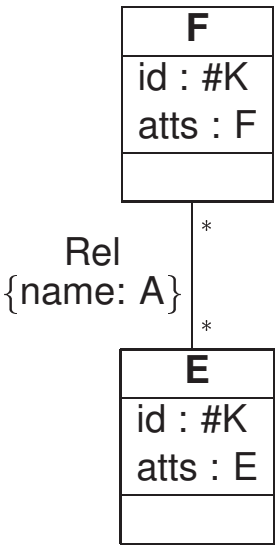
Data model

$$((\#K \rightharpoonup F) \times (\#K \rightharpoonup E) \times (\#K \rightharpoonup \#K \times A))_{\phi}$$

Invariant

$$\phi(f, e, r) \stackrel{\text{def}}{=} subd(f, r) \wedge rng\,((\#K \rightharpoonup \pi_1)r) \subseteq dom\, e$$

N:M Relationships



=

Data model

$$((\#K \rightharpoonup F) \times (\#K \rightharpoonup E) \times ((\#K \times \#K) \rightharpoonup A))_{\phi}$$

Invariant

$$\begin{array}{l} \phi(f, e, r) \stackrel{\text{def}}{=} \\ \quad let \quad d = dom\, r \\ \qquad\quad df = (\mathcal{P}\pi_1)d \\ \qquad\quad de = (\mathcal{P}\pi_2)d \\ \quad in \quad df \subseteq dom\, f \wedge de \subseteq dom\, e \end{array}$$

Data Refinement

Data representation and abstraction

- **Isomorphism** provides us with a notion of data-equivalence.
- How do we *order* data-structures according to their "size" or "expressive power"?
- Are there "better" or "worse" data-structures for a given purpose?

Below we address the field of *formal* data processing and **refinement** by transformation.

Right-invertibility revisited

Right-invertibility conveys the notions of data **representation** and **abstraction**: wherever

$$A \underset{f}{\overset{r}{\leq}} B \quad \text{that is} \quad \left\{ f \cdot r = id_A \right. \tag{143}$$

holds, this means that B contains "more information" than A. Inequation (143) has several informal interpretations:

- A is "smaller" than B
- B is able to "represent" A
- B is "abstracted" by A
- A is "implemented" by B
- B is a refinement ("refines") A

In set theory one says that f is surjective.

Terminology:

- f = the **abstraction** function
- r = the **representation** function

Comment:

In general, the same f will admit many r, *e.g.*

$$A \underset{f}{\overset{r}{\underset{\leq}{\rightleftarrows}}} A \times B$$

where

$$f \stackrel{\text{def}}{=} \pi_1$$

and, for every "default" $b_0 \in B$,

$$r \stackrel{\text{def}}{=} \langle id, b \rangle$$

is a right-inverse of f:

$$\pi_1 \cdot \langle id, b \rangle = id$$

by $\times$-cancellation (20).

Basic properties of $\leq$:

Reflexivity

$$A \underset{id}{\overset{id}{\underset{\leq}{\rightleftarrows}}} A \quad \text{cf} \quad id \cdot id = id$$

Transitivity

$$A \underset{f}{\overset{r}{\leq}} B \;\wedge\; B \underset{g}{\overset{s}{\leq}} C \;\Rightarrow\; A \underset{f \cdot g}{\overset{s \cdot r}{\leq}} C$$

cf.

$$\begin{array}{ll} & (f \cdot g) \cdot (s \cdot r) \\ = & \quad \{ \text{ associativity} \} \\ & f \cdot (g \cdot s) \cdot r \\ = & \quad \{ \ s \text{ is right-inverse of } g \} \\ & f \cdot id \cdot r \\ = & \quad \{ \text{ identity} \} \\ & f \cdot r \\ = & \quad \{ \ r \text{ is right-inverse of } f \} \\ & id \end{array}$$

Monotonicity

$$A \underset{f}{\overset{r}{\leq}} B \;\Rightarrow\; \mathsf{F}\,A \underset{\mathsf{F}\,f}{\overset{\mathsf{F}\,r}{\leq}} \mathsf{F}\,B$$

where F is an arbitrary functor:

$$\begin{array}{ll} & id \\ = & \quad \{ \text{ functors commute with } id \} \\ & \mathsf{F}\,id \end{array}$$

$$
\begin{array}{ll}
= & \{\ r \text{ is right-inverse of } f\} \\
& \mathsf{F}\,(f \cdot r) \\
= & \{\ \text{functors commute with composition}\} \\
& (\mathsf{F}\,f) \cdot (\mathsf{F}\,r)
\end{array}
$$

so $\mathsf{F}\,r$ is a right-inverse of $\mathsf{F}\,f$. Of course, this result extends to bifunctors, or n-ary functors in general.

Structural representation/abstraction

Given two inductive datatypes $\mu\mathsf{F}$ and $\mu\mathsf{G}$

$$
\begin{array}{ccc}
\mu\mathsf{F} & \cong & \mathsf{F}\,\mu\mathsf{F} \\
& \underset{in_{\mu\mathsf{F}}}{\longleftarrow} &
\end{array}
\qquad
\begin{array}{ccc}
\mu\mathsf{G} & \cong & \mathsf{G}\,\mu\mathsf{G} \\
& \underset{in_{\mu\mathsf{G}}}{\longleftarrow} &
\end{array}
$$

such that

$$
\mathsf{F}\,X \underset{g}{\overset{s}{\leq}} \mathsf{G}\,X \tag{144}
$$

Then

$$
\mu\mathsf{F} \underset{f=(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}}}{\overset{r=(\!|in_{\mu\mathsf{G}} \cdot s|\!)_{\mathsf{F}}}{\leq}} \mu\mathsf{G} \tag{145}
$$

holds, *cf.* diagram

$$
\begin{array}{ccccc}
\mu\mathsf{G} & \overset{in_{\mu\mathsf{G}}}{\longleftarrow} & \mathsf{F}\,\mu\mathsf{G} & \overset{g}{\longleftarrow} & \mathsf{G}\,\mu\mathsf{G} \\
{\scriptstyle (\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}}}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}}} & & \downarrow{\scriptstyle \mathsf{G}\,(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}}} \\
\mu\mathsf{F} & \underset{in_{\mu\mathsf{F}}}{\longleftarrow} & \mathsf{F}\,\mu\mathsf{F} & \underset{g}{\longleftarrow} & \mathsf{G}\,\mu\mathsf{F}
\end{array}
$$

This result is very important in (constructive) data-refinement calculations.

Example

Finite sequences represented by non-empty ones — $\mu \mathsf{F} = A^\star$ and $\mu \mathsf{G} = A^+$, for $1 \leq A$:

$$A^\star \xleftarrow[[\ \underline{[\ ]}, cons\ ]]{} 1 + A \times A^\star$$

$$A^+ \xleftarrow[[\ sing, cons\ ]]{} A + A \times A^+$$

where $sing\, a = [a]$.

Of course

$$1 + A \times X \underset{!+id}{\overset{\underline{a_0}+id}{\underset{\leq}{\rightleftarrows}}} A + A \times X$$

holds for some $a_0 \in A$ since $1 \leq A$ (via $!$) by hypothesis.

Via (145) we infer not only that A^+ implements A^*,

$$A^\star \underset{blast}{\overset{embed}{\underset{\leq}{\rightleftarrows}}} A^+$$

(obvious!) but how it does it:

$$blast = (\!|\ [\ \underline{[\ ]}, cons\ ] \cdot (! + id)\ |\!)$$
$$embed = (\!|\ [\ sing, cons\ ] \cdot (! + id)\ |\!)$$

(less obvious).

Let us calculate further:

$$blast \cdot [\ sing, cons\ ]$$

$$
\begin{array}{ll}
= & \{\text{ cata-cancellation}\} \\
& [\,\underline{[\,]}, cons\,] \cdot (! + id) \cdot (id + id \times blast) \\
= & \{\ +\text{-bifunctor and trivia}\} \\
& [\,\underline{[\,]}, cons\,] \cdot (! + id \times blast) \\
= & \{\ +\text{-fusion and trivia}\} \\
& [\,\underline{[\,]}, cons \cdot (id \times blast)\,]
\end{array}
$$

that is

$$
\begin{array}{rcl}
blast\,[a] & = & [\,] \\
blast\,(cons(a,l)) & = & cons(a, blast\,l)
\end{array}
$$

($blast$="all <u>b</u>ut <u>last</u> ")

Similarly:

$$
\begin{array}{rcl}
embed\,[\,] & = & [a_0] \\
embed\,(cons(a,l)) & = & cons(a, embed\,l)
\end{array}
$$

Constructive proof of (145)

We want to find a right-inverse for $(\!|in_{\mu\mathsf{G}} \cdot g|\!)$ in (145) which, of course, will be expressible as a catamorphism $(\!|\alpha|\!)$, *cf.* diagram

$$
\begin{array}{ccc}
\mu\mathsf{F} & \xleftarrow{in_{\mu\mathsf{G}}} & \mathsf{F}\,\mu\mathsf{F} \\
{\scriptstyle (\!|\alpha|\!)_{\mathsf{F}}}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|\alpha|\!)_{\mathsf{F}}} \\
\mu\mathsf{G} & \xleftarrow[\alpha]{} & \mathsf{F}\,\mu\mathsf{G} \\
{\scriptstyle (\!|in_{\mu\mathsf{G}}\cdot g|\!)_{\mathsf{G}}}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!|in_{\mu\mathsf{G}}\cdot g|\!)_{\mathsf{G}}} \\
\mu\mathsf{F} & \xleftarrow[in_{\mu\mathsf{G}}\cdot g]{} & \mathsf{F}\,\mu\mathsf{F}
\end{array}
$$

So α is the unknown. Calculation of α:

$$(\!|in_{\mu\mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot (\!|\alpha|\!) = id$$

$$
\begin{array}{ll}
\Leftrightarrow & \{ \text{ by F-cata-reflexion} \} \\
& (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot (\!|\alpha|\!)_{\mathsf{F}} = (\!|in_{\mu \mathsf{F}}|\!)_{\mathsf{F}} \\
\Leftarrow & \{ \text{ by F-cata-fusion} \} \\
& (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot \alpha = in_{\mu \mathsf{F}} \cdot \mathsf{F}\, (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \\
\Leftrightarrow & \{ \text{ decompose } \alpha = in_{\mu \mathsf{G}} \cdot \alpha' \} \\
& (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot in_{\mu \mathsf{G}} \cdot \alpha' = in_{\mu \mathsf{F}} \cdot \mathsf{F}\, (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \\
\Leftrightarrow & \{ \text{ by G-cata-cancellation} \} \\
& in_{\mu \mathsf{F}} \cdot g \cdot \mathsf{G}\, (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot \alpha' = in_{\mu \mathsf{F}} \cdot \mathsf{F}\, (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \\
\Leftrightarrow & \{ \ g \text{ is natural (146)} \} \\
& in_{\mu \mathsf{F}} \cdot \mathsf{F}\, (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \cdot g \cdot \alpha' = in_{\mu \mathsf{F}} \cdot \mathsf{F}\, (\!|in_{\mu \mathsf{F}} \cdot g|\!)_{\mathsf{G}} \\
\Leftarrow & \{ \ g \cdot s = id \} \\
& \alpha' = s
\end{array}
$$

So

$$\alpha \ = \ in_{\mathsf{G}} \cdot s$$

NB: both g and s are natural, *e.g.*

$$(\mathsf{F}\, f) \cdot g = g \cdot (\mathsf{G}\, f) \qquad \begin{array}{ccc} A & & \mathsf{F}\, A \xleftarrow{\ g\ } \mathsf{G}\, A \\ {\scriptstyle f}\downarrow & & {\scriptstyle \mathsf{F}\, f}\downarrow \qquad\qquad \downarrow{\scriptstyle \mathsf{G}\, f} \\ B & & \mathsf{F}\, B \xleftarrow[\ g\ ]{} \mathsf{G}\, B \end{array} \qquad (146)$$

holds.

Representation invariants

Given some abstraction map f, the choice of a particular rep-

resentation r induces a low-level invariant ϕ

$$A \underset{f}{\overset{r}{\underset{\leq}{\rightleftarrows}}} B \xrightarrow{\phi} 2$$

called the *representation invariant* induced by r. This emerges as characteristic predicate of the range of r,

$$\phi\, b \;\Leftrightarrow\; \exists a \in A.b = r\; a$$

Because r is always injective, one has

$$A \underset{f}{\overset{r}{\underset{\cong}{\rightleftarrows}}} B_\phi$$

So the replacement of low-level structure B by abstract structure A is safe provided ϕ is known to hold in the particular situation in hand:

$$f \cdot r = id$$
$$\phi\, b \stackrel{\text{def}}{=} (r \cdot f)b = b$$

Examples

$$A \rightharpoonup (B \times C) \underset{\bowtie}{\overset{unjoin}{\underset{\cong}{\rightleftarrows}}} ((A \rightharpoonup B) \times (A \rightharpoonup C))_{eqd}$$

where

$$eqd \stackrel{\text{def}}{=} (=) \cdot (dom \times dom)$$

while $\overset{+}{\bowtie}$ $/uncojoin$ (136) induce

$$djd \stackrel{\text{def}}{=} =_\emptyset \cdot \cap \cdot (dom \times dom)$$

Representation invariant handling rule

$$A \underset{f}{\overset{r}{\cong}} B_{\phi} \quad \wedge \quad B \underset{g}{\overset{s}{\cong}} C_{\psi}$$

implies

$$A \underset{f \cdot g}{\overset{s \cdot r}{\cong}} C_{\wedge \cdot \langle \psi, \phi \cdot g \rangle}$$

Example

From (115) and

$$\mathcal{P}A \underset{elems = (\!|ins|\!)}{\overset{\{\!|in|\!\}}{\leq}} A^{\star} \tag{147}$$

we infer:

$$A \rightharpoonup B \;\cong\; (A \times B)^{\star}_{fdp \cdot elems}$$

(finite mappings implemented by finite lists).

Representing inductive datatypes in ordinary programming languages

The relational database model

A data model is said to be compliant with the *relational database model* if it is a finitary product of expressions of the following:

- **relations** — $\mathcal{P}(A_1 \times \times \ldots A_n)$ where every A_i is regarded as an "atomic type" in the target relational environment;

- **mappings** — $(A_1 \times \ldots \times A_n) \rightharpoonup (B_1 \times \ldots \times B_m)$ where every A_i and B_j is regarded as an "atomic type" in the target relational environment.

$A_1 \times \ldots \times A_n$ are said to be **key**-attributes and $B_1 \times \ldots \times B_m$ are said to be **dependent**-attributes.

Then:

- Laws such as (132,130, 136) and (139) play an important rôle in derived construct reification. They can be used to "decompose" complex/nested mappings or sequences into tuples of simpler maps.
- Since finite maps are easily refined into binary relations — recall (115) — our calculus fragment above is useful in refining elaborate, finite-mapping/sequence based data models into relational database schemata.

Normalization

It has been proved elsewhere [RO97] that **3NF** is guaranteed by the laws mentioned above.

However, one often ignores normalization wherever data storage is not a problem, *i.e.*, one is dealing with small data sets. The following law describes one such refinement step:

$$(A \rightharpoonup B) \times (A \rightharpoonup C) \;\leq\; A \rightharpoonup ((B + 1) \times (C + 1)) \quad (148)$$

It merges two tables into a single one by, for instance, representing

NUMBER	STUDENT
1010	John
11230	Mary

and

NUMBER	COURSE
11230	C.Science
15234	Biology

by

NUMBER	STUDENT	COURSE
1010	John	-
11230	Mary	C.Science
15234	NIL	BIOLOGY

Question:

What if the data model to be refined is recursive?

Pointer introduction

For target languages like C/C++ recursive data models are coped with by resorting to pointers.

Fact:

Every non-empty datatype A can be represented by a "pointer":

$$A \underset{f=[\ \underline{a_0}, id\ ]}{\overset{r=i_2}{\leq}} 1 + A \qquad (a_0 \in A)$$

i.e.

$$A \ \cong \ (1 + A)_{\phi\, x \stackrel{\text{def}}{=} \exists a \in A : x = (i_2\ a)}$$

Then we have the following implementation rules:

- Inductive datatypes of form

$$\mathsf{T} \underset{in}{\overset{out}{\cong}} 1 + \mathsf{G}\,\mathsf{T}$$

can be directly implemented in such languages, *e.g.*

$$A^\star \cong 1 + \underbrace{A \times A^\star}_{\mathsf{G}\,A^\star}$$

implemented by

```
typedef struct G {
        A first;
        struct G *next;
        } *T;
```

- Inductive datatypes of form

$$\mathsf{T} \underset{in}{\overset{out}{\cong}} \mathsf{G}\,(1 + \mathsf{T})$$

are also directly implementable, *e.g.*

$$GenDia \cong Ind \times (1 + GenDia) \times (1 + GenDia) \qquad (149)$$

implemented by

```
typedef struct GenDia {
        Ind individual;          /* data about an i
dividual */
        struct GenDia *father; /* genealogy of hi
                                  (if known) */
        struct GenDia *mother; /* genealogy of hi
                                  (if known) */
        };
```

- All other cases:

$$\mathsf{T} \underset{in}{\overset{out}{\cong}} \mathsf{F}\,\mathsf{T}$$

implemented by forcing a pointer structure:

$$\mathsf{F}\,X \underset{\underline{k}}{\overset{i_2}{\leq}} 1 + \mathsf{F}\,X$$

and therefore (145)

$$\mathsf{T} \underset{(\!|in \cdot \underline{k}|\!)_{\mathsf{F}}}{\overset{(\!|in \cdot i_2|\!)_{\mathsf{G}}}{\leq}} \mu\mathsf{G} \tag{150}$$

where

$$\mu\mathsf{G} \;\cong\; 1 + \mathsf{F}\,\mu\mathsf{G}$$

Example:

$$DecTree \;\cong\; What \times (Answer \rightharpoonup DecTree)$$

Via (115) this is implemented by

$$DecTree_1 \;\cong\; What \times (Answer \times DecTree_1)^{\star}$$

On its turn, via (150), $DecTree_1$ is implemented by

$$DecTree_2 \;\cong\; 1 + What \times (Answer \times DecTree_2)^{\star}$$

that is

$$\begin{cases} DecTree_2 \;\cong\; 1 + What \times subTrees \\ subTrees \;\cong\; 1 + (Answer \times DecTree_2) \times subTrees \end{cases}$$

This has a direct encoding in C as follows (for $Answer$ and $What$ = strings):

```
typedef struct DecTree2{
        char *What; /* Question or Decision */
        struct subTrees *R;
        };
```

```
typedef struct subTrees {
        struct node *first;
        struct subTrees *next;
        };

typedef struct node {
        char *Answer;
        struct DecTree2 *SubTree;
        };
```

Recursion “removal”

What about SQL-like languages?
How do we convert recursive structures in non-recursive tabular ones?
Let

$$X_0 \underset{in}{\overset{\cong}{\rightleftarrows}} \mathsf{F}\, X_0$$

be a solution to

$$X \cong \mathsf{F}\, X \tag{151}$$

Fact:

$$X_0 \underset{f}{\overset{\cong}{\rightleftarrows}} (K \times (K \rightharpoonup \mathsf{F}\, K))_\phi \tag{152}$$

for K a domain of “pointers” such that $K \cong \mathbb{N}$.

Abstraction Function f

We want to define a surjection

$$X_0 \xleftarrow{f} (K \times (K \rightharpoonup \mathsf{F}\, K))_\phi$$

that is, to find appropriate f and ϕ.
Temporarily assume that $\sigma \in K \rightharpoonup \mathsf{F}\, K$ is a total function, and draw the diagram of coalgebra (K, σ):

$$\begin{array}{c} K \\ \downarrow \sigma \\ \mathsf{F}\, K \end{array}$$

Assuming a given piece of "linear storage" σ ("database"), let f_σ denote the function which, for each "pointer" $k \in K$, retrieves the value of X_0 corresponding to a σ "scan" starting from k:

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ & & \downarrow \sigma \\ & & \mathsf{F}\, K \end{array}$$

The diagram can immediately completed,

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ in \uparrow & & \downarrow \sigma \\ \mathsf{F}\, X_0 & \xleftarrow[\mathsf{F}\, f_\sigma]{} & \mathsf{F}\, K \end{array}$$

expressing f_σ as F-hylomorphism $[\![\sigma, in]\!]_{\mathsf{F}}$, that is,

$$f_\sigma = in \cdot (\mathsf{F}\, f_\sigma) \cdot \sigma$$

or

$$\begin{array}{rcl} f & :: & K \times (K \rightharpoonup \mathsf{F}\, K) \longrightarrow X_0 \\ f(k, \sigma) & \stackrel{\text{def}}{=} & in((\mathsf{F}\, f_\sigma)(\sigma\, k)) \end{array} \tag{153}$$

writing $f(k, \sigma)$ instead of "curried" $f_\sigma\, k$.

Synthesis of the representation invariant ϕ

Face pointer undefinedness in case σ is partial — $\sigma\, k$ in (153)

is undefined wherever $k \notin dom\,\sigma$ and other pointers k' in the range of σ, reachable from k, may be in the same situation.

- Pointer reachability — transitive closure of

$$k_1 <_{\mathsf{F}} k \stackrel{\text{def}}{=} k \in dom\,\sigma \wedge k_1 \in_{\mathsf{F}} \sigma\, k \tag{154}$$

(for structural membership $\in_{\mathsf{F}}$ as defined in [Oli00]), that is

$$\begin{array}{rcl} \phi(k,\sigma) & \stackrel{\text{def}}{=} & let \;\; P = \{k\} \cup \{k' \in K \mid \; k' <^{+}_{\mathsf{F}} k\} \\ & & in \;\; P \subseteq dom\,\sigma \end{array}$$

- Well-foundedness — Restrict to least fixpoints, *i.e.* guarantee that $f(k,\sigma)$ does not yield infinite results, forcing P to be well-founded *wrt.* $<_{\mathsf{F}}$:

$$\begin{array}{rcl} \phi(k,\sigma) & \stackrel{\text{def}}{=} & let \;\; P = \{k\} \cup \{k' \in K \mid \; k' <^{+}_{\mathsf{F}} k\} \\ & & in \;\; P \subseteq dom\,\sigma \wedge \\ & & \quad \forall \emptyset \subset C \subseteq P : \exists m \in C : \forall k' <_{\mathsf{F}} m : k' \notin C \end{array} \tag{155}$$

Bill of materials: refinement to the relational model

A relational implementation of this problem has to face equation (141):

$$X \;\cong\; (A + X) \rightharpoonup \mathbb{N}$$

By (152) we obtain

$$\mathsf{Hmset}\, A \;\leq\; K \times \underbrace{(K \rightharpoonup ((A + K) \rightharpoonup \mathbb{N}))}_{(*)} \tag{156}$$

Calculating a normalized relational version of $(*)$:

$$K \rightharpoonup ((A + K) \rightharpoonup \mathbb{N})$$

$$\cong \quad \{ \text{ law (135), assuming nonempty mappings} \}$$
$$(K \times (A + K)) \rightharpoonup \mathbb{N}$$
$$\cong \quad \{ \text{ representation is } distr \rightharpoonup \mathbb{N}, \textit{ cf.} \text{ law (64)} \}$$
$$((K \times A) + (K \times K)) \rightharpoonup \mathbb{N}$$
$$\cong \quad \{ \text{ representation is } unpeither, \textit{ cf.} \text{ law (129)} \}$$
$$((K \times A) \rightharpoonup \mathbb{N}) \times ((K \times K) \rightharpoonup \mathbb{N})$$

Comments:

- From the recursive multiset model we have calculated a "flat" model which expresses the semantics of the two **N:M** relationships — **Part_of** and **Sub_block_of** — present in the ***UML*** diagram given in the motivation, earlier on.
- The full semantics is easily obtained by completing the starting model with *stock* and *price* information — details in [ABNO97].

Bill of materials: refinement of the $explode$ operation

We now address the calculation of a non-recursive implementation of the $explode$ function.

Following [Oli98], we shall do it in a **generic way**, by inspection of the abstract structure of the Hmset model:

1. $A \rightharpoonup B$ is regarded as an instance of a "flat" two-parameter datatype $\mathsf{F}(A, B)$ defined by linear domain equation $X \cong 1 + (A \times B) \times X$.

2. Its hierarchical extension, $\mathsf{H}(A,B)$, defined by domain equation

$$X \cong \mathsf{F}((A+X),B)$$

(for $\mathsf{F}(A \times B) = A \rightharpoonup B$, this corresponds to Hmset A).

3. A "wrap up" arrow $\mathsf{F}(A,B) \xleftarrow{wrap} A \times B$, *cf.* $wrap(a,b) = \{a \mapsto b\}$.

4. A target monoid algebra whose carrier is the flat type,

$$\mathsf{F}(A,B) \xleftarrow{[\ \underline{\epsilon},\theta\ ]} 1 + \mathsf{F}(A,B) \times \mathsf{F}(A,B)$$

enriched with a "scalar multiplication" operator,

$$\mathsf{F}(A,B) \xleftarrow{\otimes} B \times \mathsf{F}(A,B)$$

which extends naturally to $\mathsf{H}(A,B)$. (Cf. the multiset algebra operators $\oplus$ and $\otimes$.)

The whole picture is captured by an **hylomorphism** $[\![[\ \underline{\epsilon},\theta\ ] \cdot \nu, \beta]\!]$ depicted by

$$\begin{array}{ccccc} \mathsf{H}(A,B) & & \xrightarrow{\quad\beta\quad} & & \mathsf{J}\,\mathsf{H}(A,B) \\ \downarrow{\scriptstyle f} & & & & \downarrow{\scriptstyle \mathsf{J}\,f} \\ \mathsf{F}(A,B) & \xleftarrow[[\ \underline{\epsilon},\theta\ ]]{} & 1 + \mathsf{F}(A,B) \times \mathsf{F}(A,B) & \xleftarrow[\nu]{} & \mathsf{J}\,\mathsf{F}(A,B) \end{array} \qquad (157)$$

where

$$\mathsf{J}\,X = 1 + ((A+X) \times B) \times X$$

β is the appropriate input "parser" algebra — *cf.* $outs$ in the case of $A \rightharpoonup B$ — and

$$\nu = 1 + [\ wrap, \otimes \cdot swap\ ] \cdot distl \times id$$

Deriving a linear implementation

Our purpose is to try and find an arrow ν' able to provide a linear recursive path alternative to $\nu \cdot \mathsf{J}\, f$ in diagram (157):

$$\begin{array}{ccccc}
\mathsf{H}(A,B) & & \xrightarrow{\quad\beta\quad} & & \mathsf{J}\,\mathsf{H}(A,B) \\
\Big\downarrow{\scriptstyle f} & & 1+\mathsf{F}(A,B)\times\mathsf{H}(A,B) & \xleftarrow[\nu']{} & \Big\downarrow{\scriptstyle \mathsf{J}\,f} \\
\mathsf{F}(A,B) & \xleftarrow[[\ \underline{\epsilon},\theta\ ]]{} & 1+\mathsf{F}(A,B)\times\mathsf{F}(A,B) & \xleftarrow[\nu]{} & \mathsf{J}\,\mathsf{F}(A,B)
\end{array}$$

The following is a state-of-the-art calculation, assuming natural isomorphisms

$$A \times B + C \times B \xleftarrow{distl} (A + C) \times B$$

and

$$B \times A \xleftarrow{swap} A \times B$$

and fact

$$[\ g, h\ ] \times f \ = \ [\ g \times f, h \times f\ ] \cdot distl \qquad (158)$$

Calculation steps

$$\begin{array}{lll}
f & = & [\ \underline{\epsilon}, \theta\ ] \cdot (1 + [\ wrap, \otimes \cdot swap\ ] \cdot distl \times id) \cdot (1 + ((id + f) \times id) \times f \\
& = & \quad \{\ \text{bifunctors} + \text{and} \times\ \} \\
& & [\ \underline{\epsilon}, \theta\ ] \cdot (1 + [\ wrap, \otimes \cdot swap\ ] \cdot distl \cdot ((id + f) \times id) \times f) \cdot \beta \\
& = & \quad \{\ distl \text{ is natural}\ \}
\end{array}$$

$$
\begin{array}{ll}
 & [\,\underline{\epsilon}, \theta\,] \cdot (1 + [\,wrap, \otimes \cdot swap\,] \cdot (id + f \times id) \cdot distl \times f) \cdot \beta \\
= & \quad \{\ +\text{-absorption and } swap \text{ is natural}\ \} \\
 & [\,\underline{\epsilon}, \theta\,] \cdot (1 + [\,wrap, \otimes \cdot (id \times f) \cdot swap\,] \cdot distl \times f) \cdot \beta \\
= & \quad \{\ \text{fact (159) below and identity of composition}\ \} \\
 & [\,\underline{\epsilon}, \theta\,] \cdot (1 + [\,wrap, f \cdot \otimes \cdot swap\,] \cdot distl \times f \cdot id) \cdot \beta \\
= & \quad \{\ \text{bifunctor} \times\ \} \\
 & [\,\underline{\epsilon}, \theta\,] \cdot (1 + ([\,wrap, f \cdot \otimes \cdot swap\,] \times f) \cdot (distl \times id)) \cdot \beta \\
= & \quad \{\ \text{fact (158)}\ \} \\
 & [\,\underline{\epsilon}, \theta\,] \cdot (1 + [\,wrap \times f, f \cdot \otimes \cdot swap \times f\,] \cdot distl \cdot (distl \times id)) \cdot \beta \\
= & \quad \{\ \text{identity of composition and bifunctor} \times\ \} \\
 & [\,\underline{\epsilon}, \theta\,] \cdot (1 + [\,wrap \times f, (f \times f) \cdot (\otimes \cdot swap \times id)\,] \cdot distl \cdot (distl \times i \\
= & \quad \{\ \text{bifunctor} +\ \} \\
 & [\,\underline{\epsilon}, \theta\,] \cdot (1 + [\,wrap \times f, (f \times f) \cdot (\otimes \cdot swap \times id)\,]) \cdot (1 + distl \cdot (dis \\
= & \quad \{\ +\text{-absorption, } +\text{-fusion and fact (160) below}\ \} \\
 & [\,\underline{\epsilon}, [\,\theta \cdot (wrap \times f), \underbrace{f \cdot \theta \cdot (\otimes\ \cdot swap \times id)}_{\rho}\,]\,] \cdot \underbrace{(1 + distl \cdot (distl \times id}_{\beta'} \\
= & \quad \{\ \text{introducing abbreviations } \rho \text{ and } \beta'\ \} \\
 & [\,\underline{\epsilon}, [\,\theta \cdot (wrap \times f), \rho\,]\,] \cdot \beta' \\
= & \quad \{\ \text{forcing } \epsilon \text{ in, which is the unit of } \theta\ \} \\
 & [\,\underline{\epsilon}, [\,\theta \cdot (wrap \times f), \theta \cdot \langle \underline{\epsilon}, \rho \rangle\,]\,] \cdot \beta' \\
= & \quad \{\ \text{reverse } +\text{-fusion followed by exchange law}\ \} \\
 & [\,\underline{\epsilon}, \theta \cdot \langle [\,wrap \cdot \pi_1, \underline{\epsilon}\,], [\,f \cdot \pi_2, \rho\,] \rangle\,] \cdot \beta'
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ expansion of } \rho \text{ followed by reverse } + \text{-fusion } \} \\
& [\, \underline{\epsilon}, \theta \,] \cdot (1 + \langle [\, wrap \cdot \pi_1, \underline{\epsilon} \,], f \cdot [\, \pi_2, \theta \cdot (\otimes \cdot swap \times id) \,] \rangle) \cdot \beta' \\
= & \{ \text{ reverse } \times \text{-absorption and bifunctor } + \} \\
& [\, \underline{\epsilon}, \theta \,] \cdot (1 + id \times f) \cdot (1 + \langle [\, wrap \cdot \pi_1, \underline{\epsilon} \,], [\, \pi_2, \theta \cdot (\otimes \cdot swap \times id) \,] \rangle) \cdot \\
= & \{ \text{ expanding } \beta' \text{ and introducing } \nu' \} \\
& [\, \underline{\epsilon}, \theta \,] \cdot (1 + id \times f) \cdot \nu' \cdot \beta
\end{array}
$$

From this reasoning we extract

$$
\begin{array}{l}
\nu' = (1 + \\
\quad \langle [\, wrap \cdot \pi_1, \underline{\epsilon} \,], [\, \pi_2, \theta \cdot (\otimes \cdot swap \times id) \,] \rangle) \cdot (1 + distl \cdot (distl \times id))
\end{array}
$$

which is equivalent to

$$
\begin{array}{l}
\nu' = 1 + \\
\quad \langle [\, wrap \cdot \pi_1 \cdot distl, \underline{\epsilon} \,], \theta \cdot [\, \underline{\epsilon}, \otimes \cdot swap \,] \times id \rangle \cdot (distl \times id)
\end{array}
$$

Important

Two facts required above,

$$f \cdot \otimes \;=\; \otimes \cdot (id \times f) \tag{159}$$

$$f \cdot \theta \;=\; \theta \cdot (f \times f) \tag{160}$$

can be derived provided properties

$$b \otimes (x \;\theta\; y) \;=\; (b \otimes x) \;\theta\; (b \otimes y) \tag{161}$$

and

$$(n \star b) \otimes x \;=\; n \otimes (b \otimes x) \tag{162}$$

hold, where

$$B \xleftarrow{\star} B \times B$$

is the *internal multiplication* operator which underlies the generic definition of scalar multiplication:

$$\begin{array}{ccc} B \times \mathsf{F}(A,B) & \xrightarrow{swap} & \mathsf{F}(A,B) \times B \\ & & \downarrow \varphi \\ & & \mathsf{F}(A, B \times B) \xrightarrow{\mathsf{F}(id,\star)} \mathsf{F}(A,B) \end{array} \tag{163}$$

"Vector spaces" are required by $flatten$

- Facts (161) and (162) can be found in the axiomatization of *vector spaces* in Universal Algebra compendia.
- Adopting the full axiomatization would turn B into a **field** (of scalars) and $\mathsf{F}(A,B)$ into a commutative **group** (of vectors).
- Another operation would become available, that of **scalar addition** (say $+$), which, despite being absent from our reasoning, would make sense in our context, including its axiom

$$(n+m) \otimes x \;=\; (n \otimes x)\;\theta\;(m \otimes x) \tag{164}$$

- In the *bill of materials* context, $+$ would be natural number addition.

Going imperative

Knowing that ϵ, θ form a monoid makes it possible to convert the **linear** version of f we have derived into a `while`-loop, via the technique of ***accumulation* parameter** introduction:

We omit the details and present only the outcome of the reasoning:

```
explode' :: Eq a => Hmset a -> Mset a -> Mset a
explode' (In sigma) r =
    if sigma == bottom
    then r
    else let ((x,n),ff) = getf sigma
         in explode'
              (In (ff <+> (case x of
                             Left  a -> bottom
                             Right b -> otimes n
                                          (out b))))
              (r <+> (case x of
                        Left  a -> wrap a n
                        Right b -> bottom))
```

Writing this in "pseudo-C"-like notation:

$$
\begin{array}{l}
\{\ \texttt{Parts}\ \ \texttt{r} = \bot\texttt{;} \\
\ \ \texttt{Structure}\ \ \texttt{ff} = \texttt{y;} \\
\ \ \texttt{while}\ \ (\texttt{ff} \mathrel{!}= \bot) \\
\qquad \{\ \texttt{Unit x = get(dom ff);} \\
\qquad\ \ \texttt{Quantity n = ff x;} \\
\qquad\ \ \texttt{r} = \texttt{r} \oplus \begin{cases} (\texttt{x}{==}i_1\,\texttt{k}) & \Rightarrow \begin{pmatrix} \texttt{k} \\ n \end{pmatrix} \\ (\texttt{x}{==}i_2\,\texttt{y'}) & \Rightarrow \bot \end{cases} \texttt{;} \\
\qquad\ \ \texttt{ff} = \texttt{ff} \setminus\{x\} \oplus \begin{cases} (x == i_1\,\texttt{k}) & \Rightarrow \bot \\ (x == i_2\,\texttt{y'}) & \Rightarrow n \otimes \texttt{y'} \end{cases} \texttt{;} \\
\qquad \};
\end{array}
$$

$$\}$$

where

- `Parts`: Mset A
- `Unit` : $A + $ Hmset A
- `Structure` : Hmset A

A bit of `if-then-else` logic finally turns this into something even simpler:

```
{ Parts r = ⊥;
  Structure ff = y ;
  Quantity n ;
  Unit x;
  while (ff != ⊥)
     { x = get(dom ff) ;
       n = ff x;
       ff = ff \{x};
       if (x == i1 k) r = r ⊕ (k n) ;
       if (x == i1 y') ff = ff ⊕ (n ⊗ y');

     }
}
```

At RDB-level — this can be straightforwardly encoded on top of the tabular version of Hmset A calculated earlier on.

Final Remarks

Going generic

The calculation of $explode'$ shows the prospect of thinking *generic* or *polytypically* — see [Oli98] for other instantiations of the same result.

Question:

Is there room for a single, generic theory of data calculation able to cope with *heterogeneous* data sources — relational, textual, structured, semi-structured *etc.*?

Comment:

This calls for the **unification** of (from the relational side) normalization, browsing, analytical processing with the theory of generic programming and calculation:

- However heterogeneous a data source may happen to be, if it is "structured" this means that it has the shape of an inductively finite data structure.
- So the main task appears to be that of generalizing the functorial constructs which describe relational data to arbitrary regular functors and *see what happens*.
- Of course we have to broaden our view of functional programming to that of generic (polytypic) programming [Bo98].

Sketch of an Example

Consider the following (rather simple) relational database schema

$$S = (\{A, B, C\}, A \longrightarrow B)$$

consisting of one table, three attributes and a functional dependence between attributes A and B such as in, *e.g.*, the following tabular illustration:

A	B	C
a1	b1	c1
a2	b2	c2
a1	b1	c2
a2	b2	c3

(165)

The set-theoretical model of S is

$$\mathcal{P}(A \times B \times C)_{\phi}$$

where invariant ϕ is defined by

$$\phi \stackrel{\text{def}}{=} f\,dp \cdot \mathcal{P}\langle \pi_1, \pi_2 \rangle \tag{166}$$

Let us sketch a pointfree version of the conventional relational technique which extracts functional dependences by schema decomposition [RO97].

We want to show that the $A \longrightarrow B$ functional dependence can be factored out, *i.e.* that

$$\mathcal{P}(A \times B \times C)_{\phi} \leq (A \rightharpoonup B) \times \mathcal{P}(A \times C) \tag{167}$$

holds.

This means to decompose table (165) into

A	B
a1	b1
a2	b2

$+$

A	C
a1	c1
a2	c2
a1	c2
a2	c3

Then:

$$
\begin{array}{ll}
 & \mathcal{P}(A \times B \times C) \\
\cong & \quad \{ \text{ isomorphism is } f_1 = \mathcal{P} flatr \} \\
 & \mathcal{P}(A \times (B \times C)) \\
\cong & \quad \{ \text{ isomorphism is } f_2 = discollect \} \\
 & A \rightharpoonup \mathcal{P}(B \times C)_+ \\
\cong & \quad \{ \ f_3 = A \rightharpoonup strgh \text{ is an isomorphism in the context of } \phi \text{ (166} \\
 & A \rightharpoonup B \times (\mathcal{P}C)_+ \\
\leq & \quad \{ \ f_4 = \bowtie \} \\
 & (A \rightharpoonup B) \times (A \rightharpoonup \mathcal{P}C_+) \\
\cong & \quad \{ \text{ isomorphism is } f_5 = id \times collect \} \\
 & (A \rightharpoonup B) \times \mathcal{P}(A \times C)
\end{array}
$$

(where function $strgh$ ("strength") is defined by $strgh \stackrel{\text{def}}{=} \otimes \cdot (sings \times id)$ where $s \otimes r \stackrel{\text{def}}{=} \{(a,b) \mid \ a \in s \wedge b \in r\}$.)

Does the technique recorded by inequation (167) extend to arbitrary T, *i.e.* does

$$\mathsf{T}(A \times B \times C)_\phi \ \leq \ (A \rightharpoonup B) \times \mathsf{T}(A \times C) \tag{168}$$

hold for

$$\phi \ \stackrel{\text{def}}{=} \ fdp \cdot setify \cdot \mathsf{T}\langle \pi_1, \pi_2 \rangle \tag{169}$$

where $\mathcal{P}X \stackrel{setify}{\longleftarrow} \mathsf{T}X$ is the polytypic operation which collects all data from the nodes of a T-structure? (Of course, $setify = id$ for $\mathsf{T} = \mathcal{P}$.)

Themes to develop

The answer to the question in the last paragraph is “yes”, but there is some work to be done on supporting this and all other results likely to evolve from standard *normalization theory* towards genericity.

This is a subject of current research. Some experiments are under way [Nec01] on top of the POLYP library [Jan00] for polytypic programming.

References

[ABNO97] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. Bringing CAMILA and SETS together — the `bams.cam` and `ppd.cam` CAMILA Toolset demos. Technical report, DI/UM, Braga, December 1997. *[45 p. doc.]*.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.

[BF98] J.M. Bruel and R.B. France. Transforming UML models to specifications. In *OOPSLA’98 Workshop on Formalizing UML. Why? How?*, 1998.

[BH93] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In S.A. Schuman, B. Möller, and H.A. Partsch, editors, *Formal Program Development*, number 755 in Lecture Notes in Computer Science, pages 7–42. Springer, 1993.

[Bo98] R. C. Backhouse and T. Sheeard (org.).

WGP'98 — Workshop on Generic Programming, 1998. Marstrand, Sweden, 18th June, 1998 (`http://www.cse.ogi.edu/PacSoft/conf/wgp/`).

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, Inc., 1999. ISBN 0-201-57168-4.

[Jan00] P. Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University og Technology and Götebord University, 2000.

[MA86] E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.

[Nec01] C. Necco. Generic data processing in POLYP, 2001. Master's thesis in preparation (University of San Luis, Argentina).

[Oli98] J. N. Oliveira. *'Explosive' Programming Controlled by Calculation* . Technical Report UMDITR02/98, DI, University of Minho, September 1998. Presented at *AFP'98* (3rd Intern. Summer School on Advanced Functional Programming), Braga, Portugal.

[Oli99a] J.N. Oliveira. An introduction to pointfree programming, 1999. Departamento de Informática, Universidade do Minho. 37p., chapter of book in preparation.

[Oli99b] J.N. Oliveira. Recursion in the pointfree style, 1999. 33p.

[Oli00] J.N. Oliveira. Quasi-inductive datatypes, June 2000. Departamento de Informática, Universidade do Minho. Chapter of book in preparation.

[RO97] C. J. Rodrigues and J. N. Oliveira. *Normalization is Data Reification*. Technical Report UMDITR9702, University of Minho, Dec. 1997.

[Wad89] P. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, Sep. 1989. ACM.