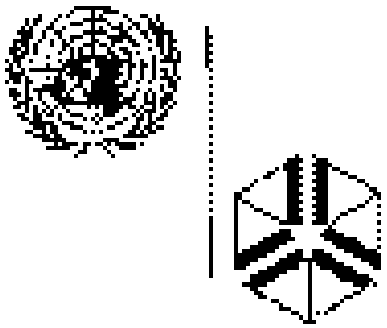
SETS —
A Data Structuring Calculus
and
Its Application to Program Development



UNU/IIST — SETS Calculus Course
5–16 May 1997
Macau

J. Nuno Oliveira
University of Minho / INESC (Group 2361)
Braga

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

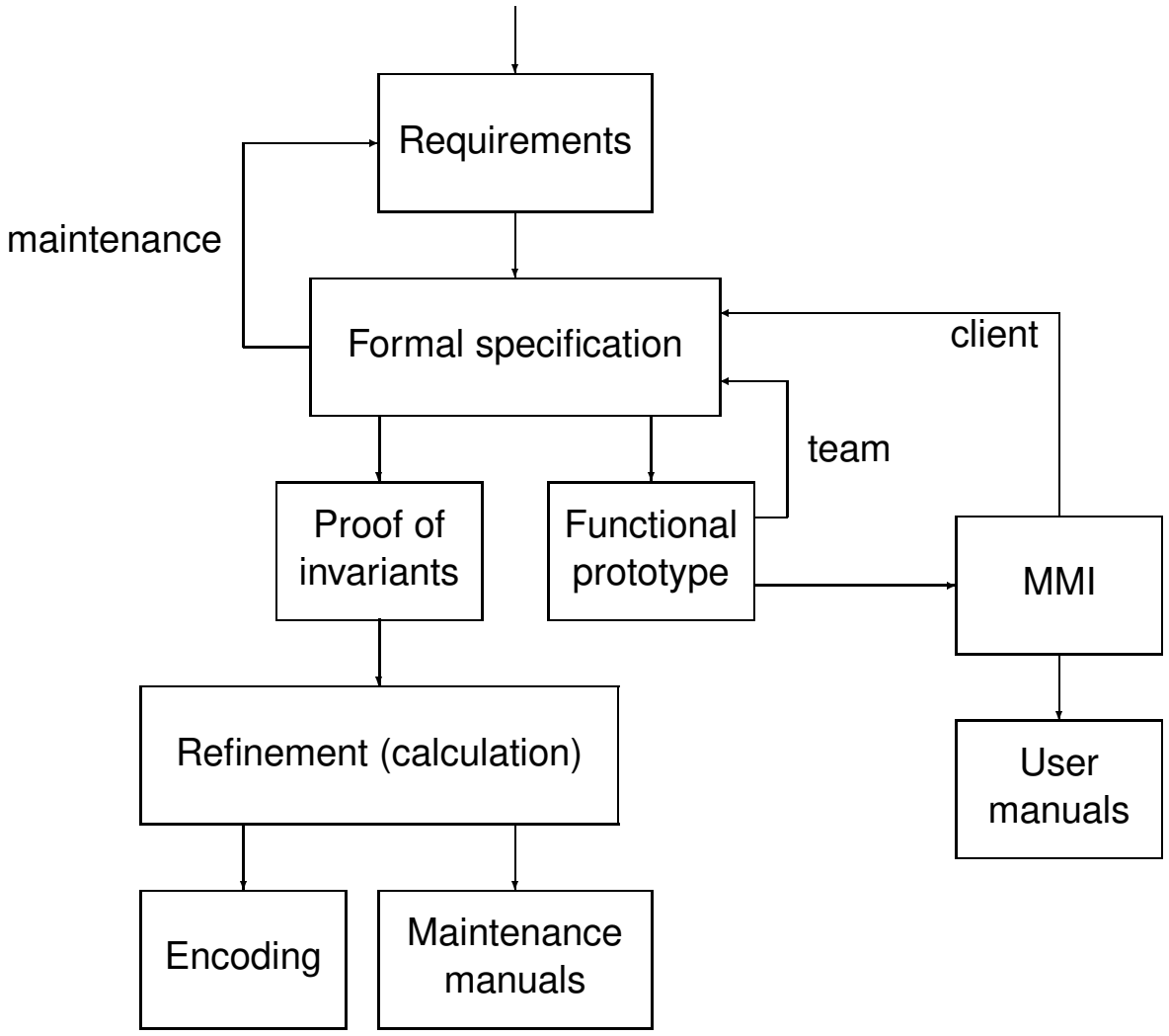
Abstract

- This course is concerned with SETS, a program design calculus for the derivation of implementations of *model-oriented* formal specifications.
- SETS is based on set-theory and (a modest use of) category theory. It can be regarded as an attempt to introduce explicit transformational design rules in program data structuring.
- SETS blends nicely with other program calculi, namely the **Fold/Unfold** or the **Oxford Calculus**.

  A main feature of SETS is that *abstraction invariants* can be synthesized by simple calculations.
- SETS is also useful in assessing, comparing, re-using or classifying specifications.
- The main results of the course are concerned with a functorial approach to specification and reification, particularly *w.r.t.* the systematic implementation of recursive data domains on non-recursive run-time environments.
- Examples of SETS's potential to replace "old" technologies such as *e.g.* E-R diagram-based database design will be provided.

The CAMILA/SETS Formal Software Development Life-cycle

- CAMILA/SETS is an emerging formal software development methodology.
- It is supported by the CAMILA rapid prototyping environment [ABNO97] and the SETS reification calculus, which is the main subject of this course.
- The methodology aims at promoting the use of *Formal Methods* at industrial level.



REMARK 1: **Why Formal Methods and Program Calculation?**

Software packages are traditionally classified as *numerical* or *non-numerical* and, in the latter case, as either *symbolic* or *data processing*. There is, however, a substantial difference in nature between the numerical and non-numerical kinds of software. While the former has a strong support by centenary mathematical disciplines (such as the differential/integral calculi, numerical analysis, analytical geometry, statistics *etc.*), the application of which is only limited by the need to finitely approximate real numbers modelling continuous entities such as space and time, the latter is usually designed by mere intuition in an *ad hoc*, trial and error style [a].

Why is this happening? Is it because there is no such mathematical support for non-numerical software design? The answer is negative. In fact, non-numerical software can be formalized by disciplines such as *abstract algebra*, *discrete mathematics* and *formal logic*. However, these are rarely found in traditional engineering curricula, which are much too concerned with the mathematics of continuity to leave room for the mathematics of discreteness. Conventional engineers are thus likely to become bad software designers. However, many curricula on software engineering have been updated with the missing background. And so-called *Formal Methods* for software production are emerging everywhere among much other literature available) albeit slow to be adopted by industry.

The main target of the formal methods trend is to drive software production into good engineering standards. As a rule, these methods split software production into a *specification phase*, in which a mathematical model is built of the contractor requirements, followed by an *implementation phase* in which such a model is somehow converted into a runnable software artifact. Relatively recent advances in formal method research show that implementations can be effectively *calculated* from specifications. So, in a sense, software technology is becoming a mature discipline in its (however late) adoption of the "universal problem solving" strategy which one is taught at school:

- understand your problem
- build a mathematical model of it
- reason in such a model
- upgrade your model, if necessary
- calculate a final solution and implement it,

the sophistication of which is only dependent upon the underlying mathematics — from simple arithmetics at primary school to systems of liner equations, then to differential/integral equations and now, in software calculi, to systems of (recursive) equations on domain spaces.

The introduction of mathematics and algebraic *calculi* in software engineering appears to be a natural evolution, when compared with the historical development of the scientific bases for other engineering areas (eg. civil and mechanical engineering etc.) which, some centuries ago, started omitting complicated geometrical proofs in favour of algebraic reasoning.

Roughly speaking, this evolution may be sketched as follows:

- Until the 1960s: intuition and craft
- 1970s: *ad hoc* (informal) methods
- 1980s: Formal methods
- 1990s: Formal calculi

We finish this remark by quoting a Portuguese mathematician of the 16^{th} century, when classic algebra was emerging and started being applied to practical problems:

> "Quien sabe por Algebra, sabe scientificamente." *Pedro Nunes (1502-1578) in libro de algebra, 1567, fol 270v.*

[a] The apparent unability to produce defect-free software and the permanent need to maintain ill software led to the identification of a so-called *software crisis* since the 1960's, which *structured programming* of the 1970's and *object-oriented programming* of the 1980's have failed to completely solve.

- SETS has been regularly taught at the University of Minho since 1989. It has become influential in the way database design is taught to Minho's undergraduate students, whereby normalized data models are obtained from formal specifications by calculation.
- SETS Home Page:

  `http://www.di.uminho.pt/~jno/html/setshp.html`

Program Calculation: An Overview

Context

SETS's context is that of *model*-oriented (or *constructive*) specification such as in *e.g.* VDM [Jon80, Jon86] and in Z [Spi89].

About the Past...

Validation { Debug; Verification { Static ('a posteriori'); Constructive (*invent & verify*) }

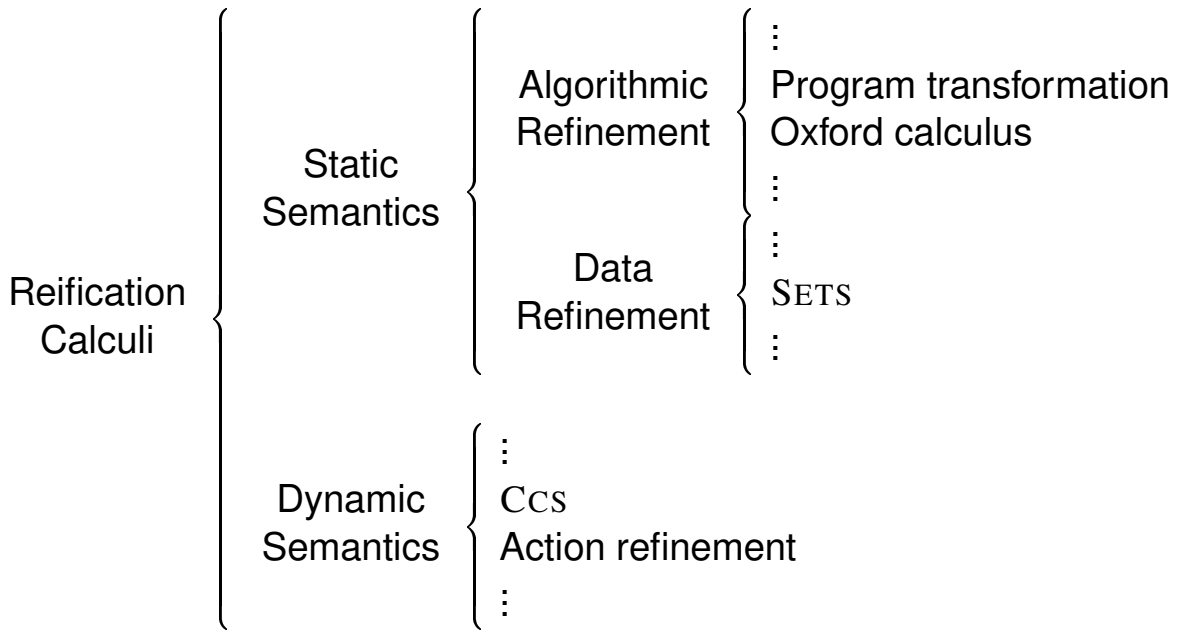
The "Invent & verify" Reification Style

- In this style for software development design is factored into as many "mind-sized" design steps as required. Every intermediate design is first proposed and then proved to follow from its antecedent.
- Such an "invent-and-verify" style is often unpractical due to the complexity of the mathematical reasoning involved in real-life software problems.

Alternative reification style

- **Idea**: develop a *calculus* allowing *programs* to be actually calculated from their specifications;
- **Style**: an intermediate design is drawn from its antecedent according to some *law* available in the calculus;
- **Proof discharge**: structural calculation instead of proofs from first principles.

A Reification Calculi Taxonomy Sketch



Starting Point

Data refinement assumes (*cf.* [MG90]):

- a domain A of "abstract" values;
- a domain B of "concrete" values;
- an *abstraction invariant* $AI \subseteq A \times B$

Functional Data Refinement

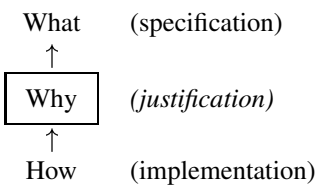
> *For any concrete value* $b \in B$ *there is* at most *one corresponding abstract value* $a \in A$

REMARK 2: **The specification/implementation/justification triad**

Research in software technology has always shown the need for splitting software design in two complementary steps: *formal specification* first (in which a mathematical text is written prescribing "what" the intended software system should do) and then *implementation* (in which machine code is produced instructing the hardware about "how" to do it).

In general, there is more than one way in which a particular machine can accomplish "what" the specifier bore in mind. Thus, the relationship between specifications and implementations is one-to-many, that is, specifications are more *abstract* than implementations. In fact, specifications are intended to be read, understood and reasoned about by humans. They prescribe behaviour rather than describe it. Implementations are intended to be executed by machines, as efficiently as possible. The programming "tricks" introduced for the sake of efficiency (in implementations) are thus irrelevant details at specification level. In summary, the "epistemological gap" between specifications and implementations is far from being a "smooth" one and is the major concern of the so-called *reification* (or *refinement*) technology, a recent branch of software engineering using formal methods.

Of course, one wants an implementation to behave exactly in the way prescribed by its specification. Thus the notion of *formal correctness* is central to any reliable reification discipline:



In the well-known *constructive style* for software development [Jon80, Jon86] design is factored into as many "mind-sized" design steps as required. Every intermediate design is first proposed and then proved to follow from its antecedent. Despite improving the primitive approach to correctness (full implementation prior to the overall correctness argument), such an "invent-and-verify" style is often impractical due to the complexity of the mathematical reasoning involved in real-life software problems.

Recent research seems to point at alternative reification styles. The idea is to develop a *calculus* allowing *programs* to be actually calculated from their specifications. In this approach, an intermediate design is drawn from a previous design according to some *law* available in the calculus, which must be *structural* in order for the components of an expression to be refined in isolation (*i.e.* pre-existing refinement results can be "re-used"). Proof discharge is achieved by performing structural calculation instead of proofs from first principles. This is the point of a calculus, as witnessed elsewhere in the past (*cf.* the differential and integral calculi, linear algebra, *etc.*). After a decade of intensive research on the foundations of *formal methods* for software design, one is tempted to forecast that year 2000 will witness the maturation phase of software technology, that of widespread software development based on formal calculi.

REMARK 3: **Why "invent and verify" is not enough**

The standard techniques for data refinement assume that the software engineer has sufficient intuition to "guess" efficient low-level model-implementations. This is unlikely, in general. Moreover, two kinds of phenomenon occur wherever model-refinement is in progress: either one is led to more *redundant* data representations, or one has to filter *invalid* data-representations (or both).

In real-life software design, it is sometimes cumbersome to formally record the relationship between data-models, and prove facts (adequacy, invariant-preservation etc. [Jon80]) about them. Unfortunately, it may take a considerable effort to prove facts which are intuitively obvious.

For example, consider the following toy-example, a META-IV syntax for a very simple bank accounting system:

$$\begin{array}{rcl} BAMS & = & AccNr \xrightarrow{m} Status \\ Status & :: & H : AccHolder\text{-set} \\ & & A : Amount \\ Amount & = & Nat0 \end{array} \tag{1}$$

where the following data-type invariant should hold,

$$\text{inv-}BAMS(\sigma) \stackrel{\text{def}}{=} \forall n \in dom(\sigma) : H(\sigma(n)) \neq \emptyset$$

enforcing that every account has, at least, one account-holder.

A VDM novice may take a while to formally discuss the correctness of the following (obvious!) *relational-model* implementation, where BAMS is modelled in terms of two binary relations (vulg. "tables"):

$$\begin{array}{rcl} BAMS1 & :: & HT : \ Row1\text{-set} \quad \textit{/*table of account-holders */} \\ & & AT : \ Row2\text{-set} \quad \textit{/*table of amounts */} \\ Row1 & :: & K : AccNr \ \ H : AccHolder \\ Row2 & :: & K : AccNr \ \ A : Amount \end{array}$$

subject to the following data-type invariant,

$$\text{inv-}BAMS1(\text{mk-}BAMS1(ht, at)) \stackrel{\text{def}}{=} dom(at) = dom(ht) \wedge depKA(at)$$

where

$$\begin{array}{rcl} dom & : & (A \ \ B)\text{-set} \longrightarrow A\text{-set} \\ dom(\rho) & \stackrel{\text{def}}{=} & \{a \in A \mid \exists b \in B : \langle a, b \rangle \in \rho\} \end{array}$$

is a generic (*domain*) relational-operator, and predicate $depKA : (Row2\text{-set}) \longrightarrow Bool$:

$$depKA(\rho) \stackrel{\text{def}}{=} \forall r, s \in \rho : (K(r) = K(s) \Rightarrow A(r) = A(s))$$

expresses a $K \rightarrow A$ *functional dependence*.

When toy-examples such as this are scaled-up to real examples, formal proofs are either discarded (and the method no longer acceptable as formal), or they become a serious bottleneck in development. Moreover, no definite answers have been given to questions such as:

- how can we define an invariant as being "correct", "too strong", or "sufficient"?
- what is the "least" abstraction *(retrieve)* function [Jon80, Jon86] between two models?
- how can we keep data *redundancy* and *validity* easily under control?
- can we equationally *derive* low-level data-models from high-level data-models?

Thus we have

$$AI(a, b) \Leftrightarrow (a = AF(b) \wedge CI(b))$$

where

- AF — *abstraction function*;
- CI — *concrete invariant*

Functional Data Refinement in SETS

Instead of being simply chosen, we want to synthesize AI by calculation.

What is a Model?

- Every piece of software is made of *data sorts* and *operations* upon them, everything forming a *syntactical category* [Wan79]:

$$\Sigma = \left\{ \begin{array}{l} \text{data sorts} \\ \text{operations} \end{array} \right.$$

- a *model* of Σ is nothing but a *functor*

$$\mathcal{A} : \Sigma \longrightarrow Set$$

- another Σ-*model*

$$\mathcal{B} : \Sigma \longrightarrow Set$$

refines $\mathcal{A}$ iff one can find an epimorphism

$$h : \mathcal{B} \longrightarrow \mathcal{A}$$

(h is a collection of abstraction functions.)

What is Set?

Recall...

REMARK 4: **Basic algebraic terminology**

Given a set Ω of function symbols, and a set S of *sorts* ("types"), a *signature* Σ is a syntactical assignment $\Sigma : \Omega \rightarrow (S^\star \times S)$ of a functionality to each function symbol; as usual, we will write $\sigma : s_1 \ldots s_n \rightarrow s$ or $s_1 \ldots s_n \xrightarrow{\sigma} s$ as shorthands of $\Sigma(\sigma) = \langle <s_1, \ldots, s_n>, s \rangle$. Let Set denote the class of all countable (finite or denumerable [a]) sets operated by set-theoretical functions. Let these be denoted by $f : X \rightarrow Y$ or $X \xrightarrow{f} Y$, where X and Y are sets.

A Σ-*algebra* $\mathcal{A}$ is a semantic assignment described by a *functor*

$$\mathcal{A} : \Sigma \rightarrow Set$$

that is, $\mathcal{A} = \langle \mathcal{A}_\Omega, \mathcal{A}_S \rangle$ where $\mathcal{A}_S$ maps sorts to corresponding carrier-sets, $\mathcal{A}_\Omega$ maps operator-symbols to set-theoretical functions, and

$$\mathcal{A}_\Omega(\sigma) : \mathcal{A}_S(s_1) \times \ldots \times \mathcal{A}_S(s_n) \rightarrow \mathcal{A}_S(s) \tag{2}$$

holds. Subscripts Ω and S may be omitted wherever they are clear from the context, *e.g.* by writing

$$\mathcal{A}(\sigma) : \mathcal{A}(s_1) \times \ldots \times \mathcal{A}(s_n) \rightarrow \mathcal{A}(s)$$

instead of expression (2).

A particular Σ-algebra is the one whose carrier-set for each sort $s \in S$ contains all the "words" (*terms*, or *morphisms*) that describe objects of that sort:

$$W_\Sigma(s) \quad \stackrel{\text{def}}{=} \quad C(s) \cup \{\sigma(t_1, \ldots, t_n) \mid \sigma : s_1 \ldots s_n \rightarrow s \wedge \forall 1 \leq i \leq n : t_i \in W_\Sigma(s_i)\}$$

where

$$C(s) \stackrel{\text{def}}{=} \{\sigma \in \Omega \mid \Sigma(\sigma) = \langle <>, s \rangle\}$$

is the set of all "constants" of type s.

In model-oriented specification, a software module is specified by producing a *model*, which is just a Σ-algebra $\mathcal{A}$, for a particular signature Σ describing its syntactical structure.

Given a specification model $\mathcal{A} : \Sigma \rightarrow Set$, a *reified model* (or refinement) of $\mathcal{A}$ is any other model $\mathcal{B} : \Sigma \rightarrow Set$ such that there is an epimorphism (*abstraction map*) from $\mathcal{B}$ to $\mathcal{A}$. In general, $\mathcal{B}$ is more *redundant* a model than $\mathcal{A}$, in the sense that a given abstract value $x \in \mathcal{A}(s)$, for some sort $s \in S$, may be represented (reified, refined) in $\mathcal{B}$ by more than one value in $\mathcal{B}(s)$. Take, for instance, the usual reification of finite sets in terms of arbitrary finite lists. The empty set $\emptyset$ is implemented by a single list value, the empty list $<>$. But any other set, *e.g.* $\{a, b\}$, has many list representatives, *e.g.* $<a, b>$, $<b, a>$, $<b, a, b>$, the abstraction map being the usual $elems$ function [Jon80].

The collection of all such (surjective) abstraction maps, one for each sort $s \in S$, makes up an epimorphism $h : \mathcal{B} \longrightarrow \mathcal{A}$ [b]. The algorithmic structure of $\mathcal{A}$ is preserved by $\mathcal{B}$ in the sense that every epimorphism h is a homomorphism, that is,

$$\mathcal{A}(\sigma)(h(b_1), \ldots, h(b_n)) \quad = \quad h(\mathcal{B}(\sigma)(b_1, \ldots, b_n)) \tag{3}$$

for every n-ary Σ-operator σ applied to (correctly type-checked) n-arguments in $\mathcal{B}$ [c].

---

[a] Following the terminology of [AKM81], by a *denumerable* set A we mean a set whose cardinal number $card(A)$ equals $\aleph_0 = card(\mathbb{N})$.

[b] The so-called *final* approach to reification [Wan79] imposes h to be unique. See [Oli92] for some discussion about this point.

[c] For improved readability, subscripts S, Ω, s, $s_1, \ldots, s_n$ have been omitted from (3), *cf.* [Oli90].

- G. Cantor (1845–1918)
- B. Russell (Paradox—1901, *Principia*—1925)
- Zermelo & Frænkl (1904)

⋮

- 1970s. . .

D. Scott	J. Adámek
C. Strachey	M. Arbib
D. Lehmann	E. Manes
B. Smyth	⋮
G. Plotkin	

For our purposes:

- Set denotes the category of countable sets (details to follow).
- SETS stands for "Set-based specification" and calculus.

The Set-based Notation

SETS should not be regarded as a competitor to specification languages such as META-IV (vulg. VDM) or Z, but rather as a framework to write and reason about specifications in a simple, "pure" mathematical notation, the one arising from the constructs of the Set Cartesian closed category:

Set Notation { Objects — *cf.* sets, data structures; Morphisms — *cf.* functions, transactions

The Primitive Notions

- Given two sets A and B, the notion of a function f from A to B, denoted $A \xrightarrow{f} B$ or $f : A \to B$ will be regarded as primitive.
- To every set A we associate the identity function 1_A which is such that $1_A(a) = a$ for every $a \in A$ (id will be used wherever A is context-implicit).

REMARK 5: **A few category-teory rudiments**

Most Set constructs which are familiar to the naïve set theorist are in fact instances of wider categorial notions which will be skimmed through in this remark.

Definition 1 (Category) *A category $\mathcal{C}$ is a mathematical concept given by a class of objects — $ob(\mathcal{C})$ — and a set of* morphisms *$\mathcal{C}(a,b)$ associated to each ordered pair $\langle a,b\rangle$ of $\mathcal{C}$-objects* [a] *such that:*

- *distinct ordered pairs of $\mathcal{C}$-objects have disjoint morphism-sets, ie.*

$$\mathcal{C}(a,b) \cap \mathcal{C}(c,d) \supset \emptyset \Rightarrow (a = c \wedge b = d)$$

- *for every three $\mathcal{C}$-objects a, b, c, if $a \xrightarrow{f} b$ and $b \xrightarrow{g} c$ are morphisms, then the* composition morphism *$g \circ f \in \mathcal{C}(a,c)$ exists;*
- *morphism-composition is associative, ie.*

$$h \circ (g \circ f) = (h \circ g) \circ f$$

- *for every $\mathcal{C}$-object a, there is an* identity morphism *$a \xrightarrow{1_a} a$ which, for every two $\mathcal{C}$-morphisms $a \xrightarrow{f} b$ and $c \xrightarrow{g} a$, is such that:*

$$\begin{aligned} f \circ 1_a &= f \\ 1_a \circ g &= g \end{aligned}$$

□

An example of a category is Set, the category such that $ob(Set)$ is the class of all countable sets and, for every two finite sets A and B, $Set(A,B)$ contains all set-theoretical functions $f : A \longrightarrow B$ (*i.e.* every Set-morphism is a total function). Clearly, for every Set-object A, the *identity morphism* $A \xrightarrow{1_A} A$ is the function such that $1_A(a) = a$ for every $a \in A$.

Definition 2 (Isomorphism) *If $a \xrightarrow{f} b$ is a $\mathcal{C}$-morphism such that*

$$\exists b \xrightarrow{f^{-1}} a.(f \circ f^{-1} = 1_b \wedge f^{-1} \circ f = 1_a)$$

*(*i.e. *f is* invertible*), then a and b are said to be isomorphical objects, and we write $a \cong b$. It is easy to show that $\cong$ is an equivalence-relation on $ob(\mathcal{C})$.* □

In Set, every *invertible* morphism is a bijection between two isomorphical sets.

Definition 3 (Functor) *A functor $\mathcal{F}$ from a given category $\mathcal{C}$ to another category $\mathcal{D}$ is a function which maps each $\mathcal{C}$-object a into a $\mathcal{D}$-object $\mathcal{F}(a)$ and each $\mathcal{C}$-arrow $a \xrightarrow{f} b$ to a $\mathcal{D}$-arrow $\mathcal{F}(a) \xrightarrow{\mathcal{F}(f)} \mathcal{F}(b)$, such that:*

- *$\mathcal{F}(1_a) = 1_{\mathcal{F}(a)}$, for every $\mathcal{C}$-object a*
- *$\mathcal{F}(g \circ h) = \mathcal{F}(g) \circ \mathcal{F}(h)$ everywhere $g \circ h$ is defined.*

□

Definition 4 (Endofunctor) *An endofunctor $\mathcal{F}$ is a functor from a category to itself, that is, such that $\mathcal{C}$ and $\mathcal{D}$ of the previous definition are the same category.* □

Set-endofunctors $\mathcal{F} : Set \longrightarrow Set$ play a prominent rôle in the SETS calculus.

[a] We will write $a \xrightarrow{f} b$ wherever $\mathcal{C}$ is implicit in the context and $f \in \mathcal{C}(a,b)$. a is said to de the *domain* of f and b is said to be the *codomain* of f.

- Function composition $f \circ g : C \to B$ is such that

$$(f \circ g)(c) \;=\; f(g(c))$$

  for $c \in C$, $f : A \to B$ and $g : C \to A$.

- We will write $A \cong B$ to mean that sets A and B are *isomorphic*, that is to say, that a bijective function can be defined between them.

Set Objects — The Primitive Constructs

The following table will help in introducing Set's primitive constructs to "program hackers":

SETS	PASCAL	C	**Informal Description**

$A \times B$	`record`
	`  P: A;`
	`  S: B`
	`end;`
		`struct {`
		`  A first;`
		`  B second;`
		`};`
			Records

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

B^A	`array[A] of B`	`B ...[A]`	Arrays
$A + 1$	`^A`	`A *...`	Pointers

In fact,

- Set is a Cartesian closed category, *i.e.* for every object A and B,
  - it has finite products ($A \times B$):

$$A \times B \;\stackrel{\text{def}}{=}\; \{\langle a, b\rangle \mid\; a \in A \wedge b \in B\}$$

  - it has exponentiations (A^B):

$$A^B \;\stackrel{\text{def}}{=}\; \{f \mid\; f : B \to A\}$$

- the empty set ($\emptyset$) is the category's initial object 0;

- every singleton set

$$\{0\} \cong \{1\} \cong \ldots \cong \{x\} \cong 1$$

  is "the" terminal object 1;

- Set also has finite co-products ($A + B$):

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

Set Objects — The Derived Constructs

On top of product, co-product and exponentiation, in Set one can build data models using the following *derived* constructs:

- subsets of a finite set A (*cf.* set of A in VDM):

$$2^A$$

- binary relations over finite sets A and B (*cf.* set of $(A \times B)$ in VDM):

$$2^{A \times B}$$

- partial maps from finite A to B (*cf.* map A to B in VDM):

$$A \hookrightarrow B = \bigcup_{K \subseteq A} B^K$$

- finite sequences on a set A (*cf.* seq of A in VDM):

$$A^\star = \bigcup_{n \geq 0} A^{\overline{n}}$$

  where each exponent $\overline{n}$ denotes the initial segment of **N** whose cardinality is n.

- union types (*cf.* $[A]$ in VDM and "pointers" in the analogy in the table of p.12):

$$A + 1$$

  where 1 stands for any singleton set — *cf.* the "pointers" entry in the table above.

Recursive Objects in Set

Recall good old "linked-lists" in PASCAL:

```
X    = ^Node;
Node = record
          P: A;
          S: X
       end;
```

The table above enables us to "reverse" this code into the following system of Set-equations:

$$\begin{cases} X &= 1 + Node; \\ Node &= A \times X \end{cases}$$

which, by substitution of $Node$, leads to the following "polynomial" equation of "degree 1":

$$X = 1 + A \times X \tag{4}$$

Many other well-known data structures are specified by Set-equations of this kind. For instance, raising X to X^2 above leads to

$$\begin{aligned} X &= 1 + A \times X^2 \\ &= 1 + A \times X \times X \\ &= 1 + X \times A \times X \end{aligned}$$

that is to say, we obtain *binary search trees*:

```
X    = ^Node;
Node = record
          Esq:  X;
          Item:  A;
          Dir:  X
       end;
```

— so binary search trees are bilinear linked lists, *etc.*

In general, (recursive) data structures will be specified by Set-equations of the form

$$X \cong \mathcal{F}(X)$$

where X is the name of a data sort and $\mathcal{F}$ is a Set-expression involving the above primitive or derived constructs.

Important: Although not every $\mathcal{F}$ can participate in the definition of a recursive data domain (we will be back to this topic), Set has solutions to most domain equations of the form $X \cong \mathcal{F}(X)$ for (endo)functors $\mathcal{F}$ involving the above constructs.

The following recursive data structure specifications will be running examples later on:

- *Decision trees:*

$$DecTree \cong What \times (Answer \hookrightarrow DecTree) \tag{5}$$

(*i.e.* dialogue controllers, *etc.*), *cf.* the following recursive VDM abstract syntax:

$$\begin{array}{rcll} DecTree & :: & Q: & What & \text{/*Question or Decision */} \\ & & R: & Answer \xrightarrow{m} DecTree & \text{/*Subtrees */} \\ What & = & ... & & \\ Answer & = & ... & & \end{array}$$

- *Genealogical diagrams*:

$$GenDia \cong Ind \times (GenDia + 1) \times (GenDia + 1) \tag{6}$$

REMARK 6: **More about conventional data structuring and** SETS

Recursive definitions are a powerful mechanism for specifying arbitrarily elaborate data structures. For example, the following concrete syntax in C for fairly unsophisticated *decision trees* where decisions are modelled by tree nodes with no answers,

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

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

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

could have been specified in the SETS notation simply as follows:

$$DecTree \cong What \times (Answer \hookrightarrow DecTree)$$

that is, $DecTree \cong \mathcal{F}(DecTree)$ for

$$\mathcal{F}(X) \cong What \times (Answer \hookrightarrow X)$$

The offset between the verbosity of the C-syntax fragment above and the economy of its Set-specification illustrates the "epistemological gap" between code (which is executable by a computer) and specification (which is understood by humans) [a].

Another direct illustration is provided by the following C-syntax fragment for *genealogical diagrams* (the "pedigree view" of family relationship):

```
typedef struct GenDia {
        Ind individual; /* data about an individual */
        struct GenDia *father; /* genealogy of his/her father (if known)
*/
        struct GenDia *mother; /* genealogy of his/her mother (if known)
*/
        }";
```

which — on the basis of the analogy of table of p.12 — is easily "reversed" into SETS notation as follows,

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

and even shortened to

$$GenDia \cong Ind \times (GenDia + 1)^2$$

as we shall see later on.

[a] Note that a few design decisions are hidden in the provided program fragment which add to its complexity. Explaining such decisions by calculation is perhaps the main 'raison d'être' of the SETS method. Quoting a well-known textbook on this subject [Mor90]: *One can also regard computer programming, or program development as it is known these days, in terms of [equation] 'solving' and 'solutions'.*

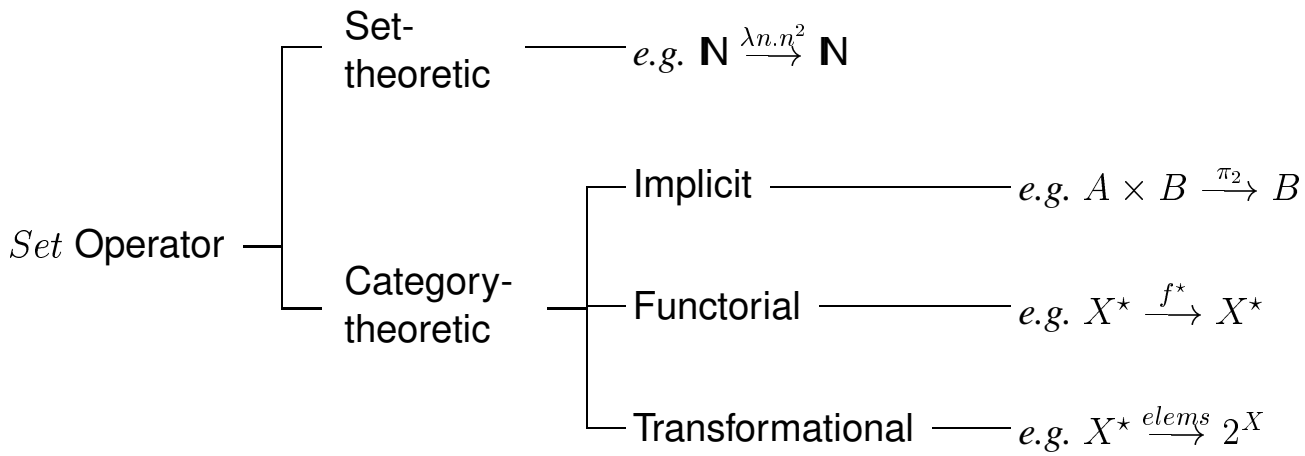
cf. the following recursive VDM abstract syntax fragment:

$$
\begin{array}{lrll}
GenDia :: & I : & Ind & \text{/*data about an individual */} \\
& F : & [GenDia] & \text{/*genealogy of his/her father (if known) */} \\
& M : & [GenDia] & \text{/*genealogy of his/her mother (if known) */}
\end{array}
$$

("pedigree view" of family relationship).

Repertoire of Set Operators (Morphisms)

The Set operator repertoire is very rich and well structured:



We will concentrate on the second branch of the tree:

CARTESIAN PRODUCT ($A \times B$)

The operators implicit in this construct are the following: projections π_1 and π_2,

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B \tag{7}$$

so that

$$
\begin{array}{rcl}
\pi_1(\langle a, b\rangle) & = & a \\
\pi_2(\langle a, b\rangle) & = & b
\end{array}
$$

and *construction*,

$$
\begin{array}{rcl}
[f, h] & : & A \longrightarrow C \times D \\
[f, h](a) & \stackrel{\text{def}}{=} & \langle f(a), h(a)\rangle
\end{array} \tag{8}
$$

for $f : A \longrightarrow C$ and $h : A \longrightarrow D$.

The functorial construct associated to products — the product $f \times g$ of

REMARK 7: **Notes about Set-based functional notation**

- SETS adopts a notation for specifying (possibly recursive) functions which is classical in mathematics, *e.g.*

$$x! = \begin{cases} x = 0 & \Rightarrow & 1 \\ \neg(x = 0) & \Rightarrow & x \times (x-1)! \end{cases}$$

instead of usual *if-then-else* notation, *e.g.*

$$\begin{array}{l} x! = \ if\ \ x = 0 \\ \qquad\ then\ \ 1 \\ \qquad\ else\ \ x \times (x-1)! \end{array}$$

In general, conditional expressions such as

$$if\ \ p(x)\ \ then\ \ g(x)\ \ else\ \ h(x)$$

are written thus:

$$\begin{cases} p & \Rightarrow & g(x) \\ \neg(p) & \Rightarrow & h(x) \end{cases}$$

Partial expressions are special cases of conditional expressions,

$$\begin{cases} p(x) & \Rightarrow & g(x) \end{cases}$$

meaning

$$\begin{cases} p & \Rightarrow & g(x) \\ \neg(p) & \Rightarrow & \bot \end{cases}$$

i.e., $p(x)$ is a pre-condition for the computation of $g(x)$.

SETS's kernel functional notation includes all set-theoretical operators typical of constructive specification languages (such as *e.g.* VDM or Z) dealing with lists (A^*), sets (2^A), mappings ($A \hookrightarrow B$) and so on. Finite mappings are enumerated by writing *e.g.*

$$\begin{pmatrix} 1 & 2 & 3 \\ a & b & c \end{pmatrix}$$

instead of (*cf.* VDM)

$$[1 \mapsto a, 2 \mapsto b, 3 \mapsto c]$$

or

$$\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}.$$

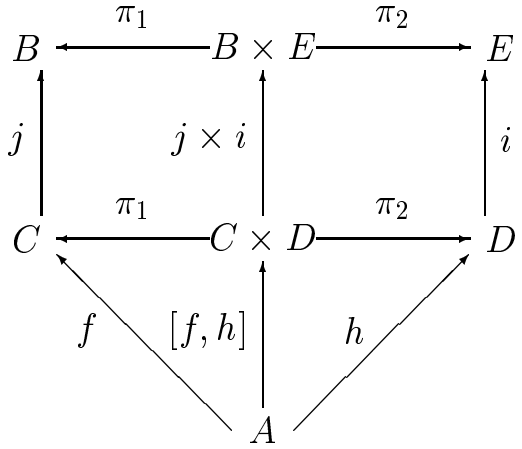
This notation extends to mapping-comprehension, *e.g.*:

$$\begin{pmatrix} i \\ i+1 \end{pmatrix}_{i \in 3}$$

two functions $f : A \longrightarrow C$ and $g : B \longrightarrow D$ — means their "parallel application":

$$\begin{array}{rcl} f \times g & : & A \times B \longrightarrow C \times D \\ f \times g(\langle a, b \rangle) & \stackrel{\text{def}}{=} & \langle f(a), g(b) \rangle \end{array} \tag{9}$$

Using a diagram:



DISJOINT UNION $(A + B)$

The operators implicit in this construct are the following injections i_1 and i_2,

$$A \stackrel{i_1}{\longrightarrow} A + B \stackrel{i_2}{\longleftarrow} B \tag{10}$$

which behave as follows,

$$i_1(a) = \langle 1, a \rangle \quad , \quad i_2(b) = \langle 2, b \rangle \tag{11}$$

and *alternation*:

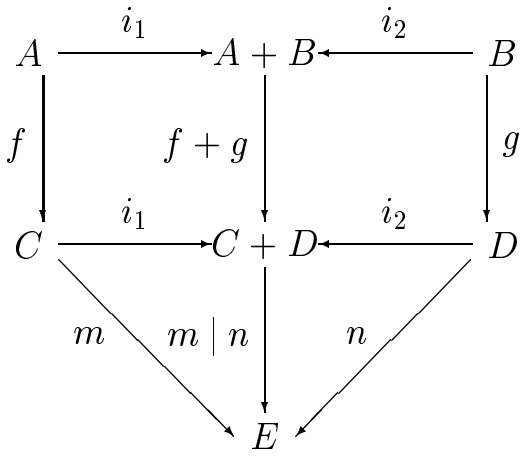
$$\begin{array}{rcl} f \mid k & : & A + B \longrightarrow C \\ f \mid k(x) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{rcl} x = i_1(a) & \Rightarrow & f(a) \\ x = i_2(b) & \Rightarrow & k(b) \end{array} \right. \end{array}$$

for $k : D \longrightarrow C$.

The functorial construct associated to co-products — the sum $f + g$ of two functions $f : A \longrightarrow C$ and $g : B \longrightarrow D$ — means their "alternative application":

$$\begin{array}{rcl} f + g & : & A + B \longrightarrow C + D \\ f + g(x) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{rcl} x = i_1(a) & \Rightarrow & i_1(f(a)) \\ x = i_2(b) & \Rightarrow & i_2(g(b)) \end{array} \right. \end{array} \tag{12}$$

Using a diagram:



EXPONENTIATION, POWERSET AND RELATIONS

The operator implicit in the exponentiation is that of *applying* a function to an argument:

$$\begin{array}{rcl} ap & : & B^A \times A \longrightarrow B \\ ap(\sigma, b) & \stackrel{\text{def}}{=} & \sigma(a) \end{array}$$

(the "array look-up" operation).

For $B = 2$, σ can be identified as the characteristic function of a subset of A. Therefore, $\mathcal{P}(A) \cong 2^A$. With some abuse of notation, we use 2^A as the *powerset of* A data domain, and so $2^{A \times B}$ as the relational space from A to B.

This leads to a very useful functorial construct (among others we will omit here): in the same way we build 2^A from A, so we derive 2^f from some $f : A \longrightarrow B$. Such f-image operator is defined as follows:

$$\begin{array}{rcl} 2^f & : & 2^A \longrightarrow 2^B \\ 2^f(x) & \stackrel{\text{def}}{=} & \{f(a) \mid a \in x\} \end{array} \tag{13}$$

In a diagram:

REMARK 8: **Some notes about Set objects, operators and diagrams**

- Binary product $A \times B$ is extensible to finitary product $A_1 \times \ldots \times A_n$, for $n \in \mathbb{N}$, also denoted by $\Pi_{i=1}^{n} A_i$, to which as many projections π_i are associated as the number n of involved factors.

- In the tradition of VDM specification, SETS tolerates user-defined *selectors* in Cartesian products, for improved readability. Their effect is as follows: we know that by writing $C = A \times B$ we enable the standard selection operators

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

  Should we alternatively write $C \cong P : A \times S : B$, selectors π_1 and π_2 will be suitably renamed:

$$A \xleftarrow{P} C \xrightarrow{S} B$$

- Disjoint union (co-product) $A + B$ is extensible to the finitary sum $A_1 + \cdots + A_n$, for $n \in \mathbb{N}$, also denoted by $\sum_{j=1}^{n} A_j$, to which as many injections i_j are assigned as the number n of involved terms.

- $\mathbb{N}$, the set of natural numbers, is a native object of Set. $\mathbb{N}$'s initial segments are defined by primitive recursion:

$$\overline{n} = \begin{cases} n = 0 & \Rightarrow \quad \emptyset \\ \neg(n = 0) & \Rightarrow \quad \{n\} \cup \overline{n-1} \end{cases}$$

  For simplicity, we will write n instead of $\overline{n}$, and therefore 0 instead of the empty set $\emptyset$.

  In Set, all sets with the same cardinality are isomorphic and therefore indistinguishable from a specification point of view. Initial segments of $\mathbb{N}$ provide an abstract mechanism for specifying enumerated types, of which only cardinality is retained. For instance, instead of enumerating

$$Weekday = \{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday\}$$

  one simply writes

$$Weekday \cong \overline{7}$$

  (note the isomorphism sign $\cong$) or even

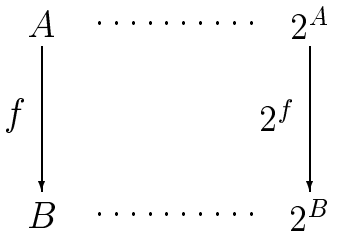
$$Weekday \cong 7$$

  Particularly common enumerated data types are 0 (the empty set), 1 (any singleton set) and 2 (the Booleans). Should the specification style become too terse, the specifier may adopt "cannonical" standards among each cardinality class, *e.g.* $2 = \{TRUE, FALSE\}$ and $1 = \{NIL\}$ (NIL is a particular good choice concerning union types and the analogy with programming language constructs).

- Diagrams provide for an elegant, graphical way of recording properties of Set-operators [a]: every time two paths in a diagram share the same source and target, equating the corresponding morphism compositions yields an axiom of the notation. For instance, from the Cartesian product diagram above (see p.19) the following axioms can be derived, concerning π_1 (*ibid* for π_2):

$$\begin{aligned} \pi_1 \circ [f, h] &= f \\ \pi_1 \circ (j \times i) &= j \circ \pi_1 \end{aligned}$$

[a] As they do in category theory in general!



Exponentiation raises a very useful notion:

POLYNOMIAL STRUCTURES

Polynomial structures are represented by Set-expressions of the form

$$\mathcal{F}(X) \;=\; \sum_{i=0}^{n} C_i \times X^i$$

cf. e.g. $\mathcal{F}(X) = 1 + C_1 \times X$ for some C_1.

Polynomial expressions are very useful in defining recursive data structures, recall (4).

FINITE MAPPINGS

Functorial construct: In the same way we derive $A \hookrightarrow C$ from two given sets A and C, from two given functions $f : A \longrightarrow B$ and $g : C \longrightarrow D$ we may define $f \hookrightarrow g$ as follows:

$$\begin{array}{rcl} f \hookrightarrow g & : & (A \hookrightarrow C) \longrightarrow (B \hookrightarrow D) \\ (f \hookrightarrow g)(\sigma) & \stackrel{\text{def}}{=} & \left(\begin{array}{c} f(a) \\ g(\sigma(a)) \end{array} \right)_{a \in dom(\sigma)} \end{array}$$

provided that f be an injective function.

$A \hookrightarrow g$ and $f \hookrightarrow C$ will be accepted as shorthands for, respectively, $1_A \hookrightarrow g$ and $f \hookrightarrow 1_C$.

In a diagram:

$$\begin{array}{ccccc} A & C & \cdots\cdots & A \hookrightarrow C \\ f \downarrow & g \downarrow & & f \hookrightarrow g \downarrow \\ B & D & \cdots\cdots & B \hookrightarrow D \end{array}$$

FINITE SEQUENCES

Functorial sequential construct: in the same way we derive $A^\star$ from a given set A, for a given function $f : A \longrightarrow B$ we may define $f^\star$ as follows:

$$\begin{array}{rcl} f^\star & : & A^\star \longrightarrow B^\star \\ f^\star(l) & \stackrel{\text{def}}{=} & <f(a) \mid a \leftarrow l> \end{array} \tag{14}$$

— recall $mapcar$ from LISP:

$$mapcar(f,l) = \left\{ \begin{array}{rcl} l = <> & \Rightarrow & <> \\ \neg(l = <>) & \Rightarrow & <f(head(l))> \frown mapcar(f, tail(l)) \end{array} \right.$$

In a diagram:

$$\begin{array}{ccc} A & \cdots\cdots\cdots & A^\star \\ f \downarrow & & f^\star \downarrow \\ B & \cdots\cdots\cdots & B^\star \end{array}$$

- How useful are these functorial constructs? — they allow for much textual and reasoning economy (see Remark 10).
- Most other constructs typical of VDM or Z fall into the next class of operators.

REMARK 9: **More about sets, sequences and relations**

- Occasionally, we may write $f[x]$ instead of $2^f(x)$. This construct (13) is often extended with a selective predicate $p : A \longrightarrow 2$ in the usual set-theoretic way:

$$\{f(a) \mid a \in x \wedge p(a)\}$$

A similar extension applies to $f^\star$ (14):

$$<f(a) \mid a \leftarrow l \wedge p(a)>$$

- In the same way binary products extend to finitary products, the relational space $2^{A \times B}$ extends to that of n-ary relations $2^{A_1 \times \ldots \times A_n}$. These are a very convenient piece of mathematics for the kind of tabular information typical of relational databases. For instance, table

A_1	A_2	A_3
a	b	c
a	d	e
f	d	g

(15)

is specified by the mathematical object

$$r \quad = \quad \{\langle a, b, c\rangle, \langle a, d, e\rangle, \langle f, d, g\rangle\}$$

Concerning n-ary relations in $2^{A_1 \times \ldots \times A_n}$, we will be using the following *projection/selection* operators:

$$\begin{array}{rcl} proj & : & \overline{n} \times 2^{A_1 \times \ldots \times A_n} \longrightarrow \bigcup_{i=1}^{n} 2^{A_i} \\ proj(i,t) & \stackrel{\text{def}}{=} & \{\pi_i(r) \mid r \in t\} \\ & = & 2^{\pi_i}(t) \end{array} \tag{16}$$

and

$$\begin{array}{rcl} sel & : & (\sum_{i=1}^{n} A_i) \times 2^{A_1 \times \ldots \times A_n} \longrightarrow 2^{A_1 \times \ldots \times A_n} \\ sel(\langle i, a\rangle, t) & \stackrel{\text{def}}{=} & \{r \in t \mid a = \pi_i(r)\} \end{array} \tag{17}$$

Exercise 1 Prove or contradict each of the following equalities, where

$$r^{-1} = \{\langle b, a\rangle \mid \langle a, b\rangle \in r\} \tag{18}$$

denotes r's inverse relation and $\circ$ denotes relational composition:

$$2^{(f \circ h)} = 2^f \circ 2^g \tag{19}$$
$$2^f(\emptyset) = \emptyset \tag{20}$$
$$2^f(x \cup y) = 2^f(x) \cup 2^f(y) \tag{21}$$
$$2^f(x \cap y) = 2^f(x) \cap 2^f(y) \tag{22}$$
$$(r \circ s)^{-1} = s^{-1} \circ r^{-1} \tag{23}$$
$$r \circ (s \cup t) = (r \circ s) \cup (r \circ t) \tag{24}$$
$$r \circ \emptyset = \emptyset \tag{25}$$

□

REMARK 10: **On the usefulness of Set-functorial approach to data structuring**

Since its early development, the SETS calculus has an inherent categorial basis, although focussed on the category Set of finite sets and set-theoretical functions. A basic ingredient of SETS is that of regarding a "data sort" as an (endo)functor in such a category [Oli92]. For instance, constant functors such as

$$\mathcal{F}(X) \quad \cong \quad \mathbb{N}_0$$

provide for primitive data sorts, while other functors such as

$$\mathcal{F}(X) \quad \cong \quad X^*$$

(*cf.* the "lists of Xs" data sort) provide for parametric data sorts.

Such a functorial basis leads us, for free, to rich design algebras involving both objects (sets) and morphisms (functions). For instance, given $A \xrightarrow{f} B$ an arbitrary fragment of a data type specification, the following extension to such a specification,

$$A^\star \quad \xrightarrow{f^\star} \quad B^\star$$

where

$$f^\star(l) \quad \stackrel{\text{def}}{=} \quad <f(a) \mid a \leftarrow l>$$

becomes implicitly available, packaged with properties such as (recall Definition 3)

$$(f \circ g)^\star \quad = \quad f^\star \circ g^\star$$

which elegantly express well-known facts of conventional functional programming practice, in this case:

$$mapcar(\lambda x.f(g(x)), l) \quad = \quad mapcar(f, mapcar(g, l))$$

Examples of other well-known endo-functors in SETS are $\lambda X.2^X$ (the "finite subsets of X" data sort) and $\lambda X.A \hookrightarrow X$ (the "finite mappings from A to X" data sort). For instance, from the above morphism f we may now enrich our data type specification with

$$2^A \quad \xrightarrow{2^f} \quad 2^B$$

where

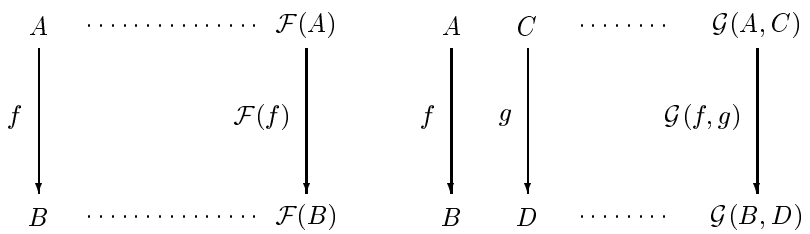
$$2^f(x) \quad \stackrel{\text{def}}{=} \quad \{f(a) \mid \ a \in x\}$$

knowing in advance that property

$$2^{(f \circ g)} \quad = \quad 2^f \circ 2^g$$

will hold, and so on.

All this is compositional. For instance, $\mathcal{F}(X) = (2^X)^\star$ will induce $(2^f)^\star$, *etc.*In general, the following diagrams can always be drawn for unary parametric data sorts ($\mathcal{F}$) or binary ones ($\mathcal{G}$):



(the obvious extension of $\mathcal{G}$ to n-ary parametric data sorts is of little practical interest.)

REMARK 11: Set-**functorial approach to recursive data structuring**

Set-functorial approach to data structuring extends naturally to the specification of recursive data sorts. These are described by recursive endo-functors, for instance

$$\mathcal{F}(X) \quad \cong \quad \mu Y(1 + X \times Y^2)$$

— that is, equation

$$Y \quad \cong \quad 1 + X \times Y^2$$

— specifies the "binary tree of Xs" data sort. The binary-recursive "mapcar-like" extension of f is, then

$$\mathcal{F}(f) \quad \stackrel{\text{def}}{=} \quad \mu y(1 + f \times y^2)$$

that is, function [a]

$$\mathcal{F}(f)(y) \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{rcl} y = NIL & \Rightarrow & NIL \\ y = \langle x, y_1, y_2 \rangle & \Rightarrow & \langle f(x), \mathcal{F}(f)(y_1), \mathcal{F}(f)(y_2) \rangle \end{array} \right.$$

and we have the following property,

$$\mathcal{F}(f \circ g) \quad = \quad \mathcal{F}(f) \circ \mathcal{F}(g)$$

stating that two consecutive binary-traversals of the same binary tree amount to a single, composite traversal (recall the "loop-intercombination" topic so dear to program transformation [Dar82]).
In summary, for a given "basic" data type specification (a collection of morphisms between data sets), for every data sort constructor (functor) $\mathcal{F}$, there is an immediate extension of such a specification which includes the $\mathcal{F}$-extension of all morphisms in the original collection. Moreover, every such an extension is packaged with the following generic (*i.e.* functorial) properties:

$$\begin{array}{rcl} \mathcal{F}(f \circ g) & = & \mathcal{F}(f) \circ \mathcal{F}(g) \\ \mathcal{F}(1_A) & = & 1_{\mathcal{F}(A)} \end{array}$$

(1_A denotes the identity function on data sort A).

Exercise 2 Prove or contradict each of the following equalities:

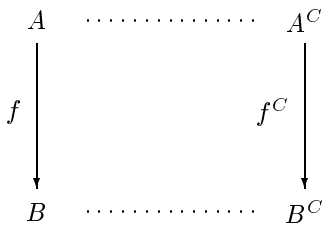
$$\begin{array}{rcll} (f \circ h) \times (g \circ i) & = & (f \times g) \circ (h \times i) & (26) \\ 1_{A \times B} & = & 1_A \times 1_B & (27) \\ [f, g] \circ h & = & [f \circ h, g \circ h] & (28) \\ \pi_1 \circ [f, g] & = & f & (29) \\ \pi_2 \circ [f, g] & = & g & (30) \\ (f \circ h) + (g \circ i) & = & (f + g) \circ (h + i) & (31) \\ 1_{A+B} & = & 1_A + 1_B & (32) \end{array}$$

□

[a] The injection functions strictly required by (12) have been dropped from the body of this function, for improved readability.

REMARK 12: **More about exponentiation and polynomial structures**

The functorial behaviour of exponentiation deserves a separate treatment of each argument A and C of the A^C construct.
Fixing C we obtain functor $\lambda X.X^C$, and $f^C(\sigma) = f \circ \sigma$, *cf.* diagram



This underpins the definition of a polynomial functor:

Definition 5 (Polynomial (Endo)functor) *A polynomial (endo) functor is a functor $\mathcal{F} : Set \longrightarrow Set$ which is either*

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

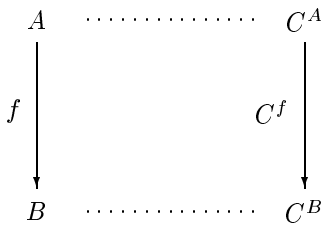
It is useful to know that every polynomial functor $\mathcal{F} : Set \longrightarrow Set$ may be put into the canonical form,

$$\begin{array}{rcl} \mathcal{F}(X) & \cong & C_0 + (C_1 \times X) + (C_2 \times X^2) + \cdots + (C_n \times X^n) \\ & = & \sum_{i=0}^{n} C_i \times X^i \end{array} \tag{33}$$

and that polynomial functors can be converted into canonical form via *Newton's binomial formula*:

$$(A + B)^n \cong \sum_{p=0}^{n} {}^nC_p \times A^{n-p} \times B^p \tag{34}$$

Fixing C in C^A yields functor $\lambda X.C^X$, *cf.*



which needs some extra care: f is required to be injective, and $C^f(\sigma) \stackrel{\text{def}}{=} \sigma \circ f^{-1}$ [a].

[a] This anomaly arises from an over-simplification of the current exposition. In fact, there will be no anomaly at all if we treat this functor as we should do: as a "contravariant" functor. This will entail the necessary distinction between $\mathcal{P}(A)$ and 2^A at morphism level but, for our purposes in this course, we do not need to go in this direction (provided we know what we are doing).

Transformational (Polymorphic) Operators

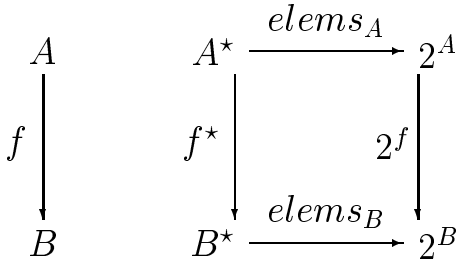
- **Quotation** (from *Theorems for free!*, by Philip Wadler [Wad89]):

  > *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 result which has a categorial nature — polymorphic functions can be regarded as *natural transformations* between appropriate functors in an appropriate category-theoretical setting, the theorems arising from the inherent commutative diagrams.

A well-known example of this kind of operator is:

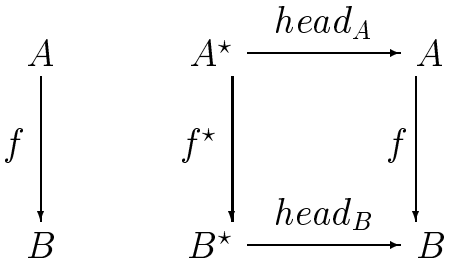


the corresponding *theorem for free* being (with a slight notation simplification):

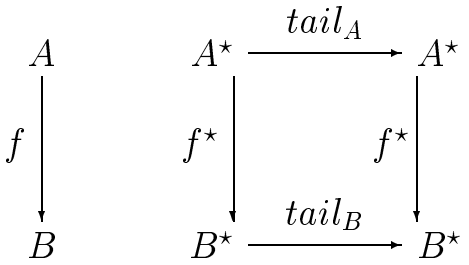
$$2^f \circ elems \;=\; elems \circ f^\star$$

See in the following examples how the usual VDM-like operators can be addressed in this way:

- Head and tail of a sequence:

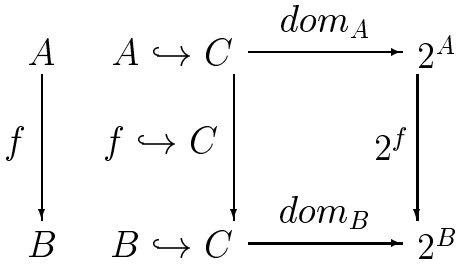


$$head \circ f^\star = f \circ head$$

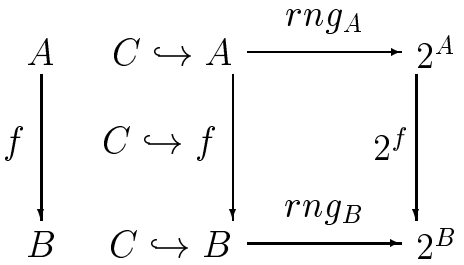


$$tail \circ f^\star = f^\star \circ tail$$

- Domain and range of a finite mapping:



$$dom \circ (f \hookrightarrow C) = 2^f \circ dom$$



$$rng \circ (C \hookrightarrow f) = 2^f \circ rng$$

- *etc. etc.*

Introduction to the Set Calculus

Questions

- What does it mean to say that two Set-expressions are the same?
- When do two data structures have the same expressive power?

Data Equivalence

We don't want to distinguish between two pieces of software whose data are "equivalent", *e.g.*

REMARK 13: **On the algebra of finite mappings**

The following properties hold for arbitrary finite mappings f, g e h [a]:

$$f \cup g = g \cup f \tag{35}$$
$$f \cup f = f \tag{36}$$
$$f \dagger f = f \tag{37}$$
$$f \cup (g \cup h) = (f \cup g) \cup h \tag{38}$$
$$f \dagger (g \dagger h) = (f \dagger g) \dagger h \tag{39}$$
$$f = f \cup (\ ) \tag{40}$$
$$f = (\ ) \cup f \tag{41}$$
$$f = f \dagger (\ ) \tag{42}$$
$$f = (\ ) \dagger f \tag{43}$$
$$dom(f \cup g) = dom(f) \cup dom(g) \tag{44}$$
$$dom(f \dagger g) = dom(f) \cup dom(g) \tag{45}$$
$$rng(f \dagger g) = rng(f) \cup rng(g) \tag{46}$$
$$rng(f \cup g) = rng(f) \cup rng(g) \tag{47}$$
$$f[dom(f)] = rng(f) \tag{48}$$
$$(f \cup g) \setminus S = (f \setminus S) \cup (g \setminus S) \tag{49}$$
$$(f \dagger g) \setminus S = (f \setminus S) \dagger (g \setminus S) \tag{50}$$
$$(f \setminus S) \setminus R = f \setminus (S \cup R) \tag{51}$$
$$f \circ (g \setminus S) = (f \circ g) \setminus S \tag{52}$$
$$f \mid (S \cup R) = (f \mid S) \cup (f \mid R) \tag{53}$$
$$(f \mid R) \mid S = (f \mid S) \mid R \tag{54}$$
$$(f \mid R) \setminus R = f \mid \emptyset \tag{55}$$
$$(f \dagger g) \mid S = (f \mid S) \dagger (g \mid S) \tag{56}$$
$$f \setminus S \cup f \mid S = f \tag{57}$$
$$(f \setminus R) \mid S = f \mid (S - R) \tag{58}$$
$$S \cap dom(f) = \emptyset \Rightarrow f \setminus S = f \tag{59}$$
$$dom(f \setminus S) = dom(f) - S \tag{60}$$
$$dom(f \mid S) = dom(f) \cap S \tag{61}$$
$$f \dagger g = f \setminus dom(g) \cup g \tag{62}$$
$$f \mid dom(f) = f \tag{63}$$
$$(f \dagger g) \mid S = (f \mid S) \dagger (g \mid S) \tag{64}$$
$$f = f \setminus S \cup f \mid S \tag{65}$$
$$f \setminus dom(f) = (\ ) \tag{66}$$
$$f \setminus (S \cap R) = f \setminus S \cup f \setminus R \tag{67}$$
$$(\ ) \setminus S = (\ ) \tag{68}$$
$$(\ ) \mid S = (\ ) \tag{69}$$

[a] The arguments of the mapping union operator ($\cup$), wherever this occurs, are furthermore assumed to be coherent mappings.

REMARK 14: **On the algebra of finite sequences**

Let s, r, t be finite sequences in $A^\star$, $f : A \longrightarrow B$ and $g : B \longrightarrow C$ be Set functions, and a an arbitrary element of A.
The following properties hold, where $s \frown r$ denotes the concatenation of s and r, and $cons(a, s) = <a> \frown s$:

$$head(cons(a, s)) = a \quad (70)$$
$$tail(cons(a, s)) = s \quad (71)$$
$$s \frown <> = s \quad (72)$$
$$<> \frown s = s \quad (73)$$
$$s \frown (r \frown t) = (s \frown r) \frown t \quad (74)$$
$$g^* \circ f^* = (g \circ f)^* \quad (75)$$
$$f^*(s \frown r) = f^*(s) \frown f^*(r) \quad (76)$$
$$<a> \frown s = cons(a, s) \quad (77)$$
$$length(<>) = 0 \quad (78)$$
$$length(cons(a, s)) = 1 + length(s) \quad (79)$$
$$length(s \frown r) = length(s) + length(r) \quad (80)$$
$$length \circ f^* = length \quad (81)$$
$$elems(<>) = \emptyset \quad (82)$$
$$elems(cons(a, s)) = \{a\} \cup elems(s) \quad (83)$$
$$elems(s \frown r) = elems(s) \cup elems(r) \quad (84)$$
$$elems \circ f^* = 2^f \circ elems \quad (85)$$
$$elems(<f(a) \mid a \leftarrow s \wedge p(a)>) \subseteq f[elems(s)] \quad (86)$$

For $s \neq <>$ and $1 \leq i \leq length(s)$, one has:

$$tail(s)(i) = s(i + 1) \quad (87)$$
$$tail(s \frown r) = tail(s) \frown r \quad (88)$$
$$head(s \frown r) = head(s) \quad (89)$$
$$tail \circ f^* = f^* \circ tail \quad (90)$$
$$elems(tail(s)) \subseteq elems(s) \quad (91)$$
$$length(tail(s)) = length(s) - 1 \quad (92)$$
$$cons(head(s), tail(s)) = s \quad (93)$$

```
record                                  record
    F: A;                                   F: record
    S: record                                      F: A;
           F: B;          equivalent to            S: B
           S: C;                                end;
       end                                  S: C;
end;                                    end;
```

that is, we want that enforce *associativity* of $\times$ (*i.e.* of `record` in PASCAL or `struct` in C):

$$A \times (B \times C) \;=\; (A \times B) \times C$$

This property is not valid in set-theory. But it will hold if we replace "$=$" by "$\cong$" (Set-isomorphism):

$$A \times (B \times C) \;\cong\; (A \times B) \times C$$

- Set is rich in laws such as the one above.
- Such laws arise from its categorial nature.
- Re-interpreting such laws in face of the analogies of the table of p.12 is a suggestive exercise.

The $\cong$-Subcalculus

Up to set-theoretical isomorphism, Set may be regarded as a commutative semiring under $\times$ and $+$:

$$A \times B \;\cong\; B \times A \tag{94}$$
$$A \times (B \times C) \;\cong\; (A \times B) \times C \tag{95}$$

$$A \times 1 \cong A \tag{96}$$
$$A + B \cong B + A \tag{97}$$
$$A + (B + C) \cong (A + B) + C \tag{98}$$
$$A + 0 \cong A \tag{99}$$
$$A \times 0 \cong 0 \tag{100}$$
$$A \times (B + C) \cong (A \times B) + (A \times C) \tag{101}$$

Therefore, it will make sense to write finitary products and sums:

$$A_1 \times \ldots \times A_n \quad that\ is \quad \Pi_{i=1}^{n} A_i$$
$$A_1 + \cdots + A_n \quad that\ is \quad \sum_{i=1}^{n} A_i$$

and we have

$$\underbrace{A \times \ldots \times A}_{n} \cong A^n \tag{102}$$
$$\underbrace{A + \ldots + A}_{n} \cong n \times A \tag{103}$$

Concerning exponentiation we have:

$$A^0 \cong 1 \tag{104}$$
$$A^1 \cong A \tag{105}$$
$$(B \times C)^A \cong B^A \times C^A \tag{106}$$
$$C^{A \times B} \cong (C^B)^A \tag{107}$$
$$A^{B+C} \cong A^B \times A^C \tag{108}$$
$$1^A \cong 1 \tag{109}$$

Relating exponentiation with finite mappings we have:

$$A \hookrightarrow B \cong (B + 1)^A \tag{110}$$
$$(A \hookrightarrow B)^C \cong (C \times A) \hookrightarrow B \tag{111}$$

Also note that

$$A^\star \cong \sum_{n \geq 0} A^n \tag{112}$$
$$A \hookrightarrow B \cong \sum_{K \subseteq A} B^K \tag{113}$$

since

$$A \neq B \;\Rightarrow\; X^A \cap X^B = \emptyset$$
$$A \cap B = \emptyset \;\Rightarrow\; A \cup B \cong A + B$$

Example of $\cong$-Reasoning

Re-write (5) as follows:

$$T \cong I \times (A \hookrightarrow T) \tag{114}$$

and instantiate $A = 2$. Then

$$\begin{aligned} I \times (A \hookrightarrow T) &\cong I \times (2 \hookrightarrow T) \\ &\cong I \times (T+1)^2 \\ &\cong I \times (T+1) \times (T+1) \end{aligned}$$

that is, (114) becomes

$$T \cong I \times (T+1) \times (T+1) \tag{115}$$

Finally, compare (115) with (6).

Conclusion

Genealogical diagrams have been shown to "be" particular cases of *decision trees*, that is, if a package implementing the latter is available, then it may be *re-used* to implement the former (*cf. re-usability, classification* etc.).

A Redundancy Ordering in Set

Question

Now that we already have a notion of Set-equivalence, how do we "order" data-structures according to their "size" or "expressive power"?

Starting Point

Set-theoretical cardinality:

$$A \preceq B \stackrel{\text{def}}{=} \exists B \xrightarrow{f} A : f \textit{ is surjective.}$$

Informal meanings of $A \preceq B$:

- A is "smaller" than B
- B is is able to "represent" A
- A is "abstracted" by B
- A is "implemented" by B
- B "refines" A

One will write $A \preceq_f B$ to record f (=abstraction map), the particular way in which B-values represent A-values, *e.g.*:

$$2^A \preceq_{elems} A^\star$$

Properties

It is easy to check that:

- $\preceq$ is reflexive,

$$A \preceq_{1_A} A$$

- $\preceq$ is transitive,

$$A \preceq_f B \wedge B \preceq_g C \Rightarrow A \preceq_{f \circ g} C$$

- $\preceq$-antisymmetry induces set-theoretical isomorphism:

$$A \preceq_f B \wedge B \preceq_g A \Rightarrow A \cong B$$

- for n $\preceq$-reasoning steps, we have

$$\underbrace{A \preceq_{f_1} A_1 \preceq_{f_2} \cdots \preceq_{f_n} A_n}_{f = f_1 \circ \cdots \circ f_n}$$

Question

Can we rely on $\preceq$ as a general *refinement* relation on data structure specifications?

No — we are ignoring *concrete invariants!*

(Counter)Example:

$$A \hookrightarrow B \preceq 2^{A \times B}$$

is not correct: every mapping "abstracts" some relation but not every relation "represents" some function.

In fact:

$$\begin{aligned} A \hookrightarrow B &\cong \{\rho \subseteq A \times B \mid \forall \langle a, b\rangle, \langle a', b'\rangle \in \rho : (a = a' \Rightarrow b = b')\} \\ &= \{\rho \in 2^{A \times B} \mid fdp(\rho)\} \end{aligned}$$

for

$$fdp(\rho) \stackrel{\text{def}}{=} \forall \langle a, b\rangle, \langle a', b'\rangle \in \rho : (a = a' \Rightarrow b = b')$$

and

$$mkf(\rho) \stackrel{\text{def}}{=} \begin{pmatrix} \pi_1(t) \\ \pi_2(t) \end{pmatrix}_{t \in \rho} \tag{116}$$

Therefore:

$$A \hookrightarrow B \trianglelefteq^{fdp}_{mkf} 2^{A \times B} \tag{117}$$

An Abstraction Invariant Ordering in Set

Because of concrete data type invariants we need to improve $\preceq$ into:

$$A \trianglelefteq B \stackrel{\text{def}}{=} \exists S \subseteq B : A \preceq_f S. \tag{118}$$

i.e.

$$A \trianglelefteq^{\phi}_{f} B \tag{119}$$

where ϕ denotes the characteristic function of S in B.

Generally speaking:

$$A \trianglelefteq_f^{\phi} B \stackrel{\text{def}}{=} \exists B \xrightarrow{f} A : \begin{array}{l} f \textit{ is partial and surjective } \wedge \\ \phi(b) \Leftrightarrow b \in dom(f) \end{array}$$

is the same as stating the following *Abstraction Invariant*:

$$\lambda(a,b).(a = f(b)) \wedge \phi(b)$$

Properties of $\trianglelefteq$:

- Reflexivity:

  $$A \trianglelefteq_{1_A}^{\lambda a.TRUE} A$$

  (NB: we will omit concrete invariant $\lambda a.TRUE$ hereafter)

- Symmetry:

  $$A \trianglelefteq_f^{\phi} B \wedge B \trianglelefteq_g^{\gamma} A \Leftrightarrow A \cong B$$

  So:

  $$A \cong B \;\Rightarrow\; A \trianglelefteq B$$

  that is, isomorphism is a special case of refinement.

- Transitivity:

  $$A \trianglelefteq_f^{\phi} B \wedge B \trianglelefteq_g^{\gamma} C \Rightarrow A \trianglelefteq_{f \circ g}^{\rho} C$$

  for

  $$\rho(c) = \gamma(c) \wedge \phi(g(c)) \tag{120}$$

  For a chain of n $\trianglelefteq$-steps,

  $$\trianglelefteq_i \left\{ \begin{array}{l} f_i \\ \phi_i \end{array} \right.$$

  we have

  $$\trianglelefteq \left\{ \begin{array}{l} f \stackrel{\text{def}}{=} \bigcirc_{i=1}^{n} f_i \\ \phi \stackrel{\text{def}}{=} \lambda x.\ \wedge_{i=n}^{1} \phi_i((\bigcirc_{j=i+1}^{n} f_j)(x)) \end{array} \right. \tag{121}$$

  Example ($n = 3$):

  $$\begin{array}{rcl} f & \stackrel{\text{def}}{=} & f_1 \circ f_2 \circ f_3 \\ \phi & \stackrel{\text{def}}{=} & \lambda x.\, \phi_3(x) \wedge \phi_2(f_3(x)) \wedge \phi_1(f_2(f_3(x))) \end{array}$$

The $\trianglelefteq$-Subcalculus

Most $\trianglelefteq$-results concern the reification of finite partial maps. Do laws (104) to (109) concerning exponentiation (total maps) hold for partial maps?

- (104) holds:

$$\begin{aligned} 0 \hookrightarrow A &\cong (A+1)^0 \\ &\cong 1 \end{aligned} \tag{122}$$

- (105) becomes:

$$\begin{aligned} 1 \hookrightarrow A &\cong (A+1)^1 \\ &\cong A+1 \end{aligned} \tag{123}$$

- (108) holds:

$$\begin{aligned} (B+C) \hookrightarrow A &\cong (A+1)^{B+C} \\ &\cong (A+1)^B \times (A+1)^C \\ &\cong (B \hookrightarrow A) \times (C \hookrightarrow A) \end{aligned} \tag{124}$$

- (109) becomes:

$$\begin{aligned} A \hookrightarrow 1 &\cong (1+1)^A \\ &\cong 2^A \end{aligned} \tag{125}$$

still in the $\cong$-subcalculus.

The remaining two laws lead to $\trianglelefteq$-results:

- (106) "holds" at $\hookrightarrow$ -level provided that $\trianglelefteq$ replaces $\cong$:

$$A \hookrightarrow B \times C \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C) \tag{126}$$

- (107) also "holds" at $\hookrightarrow$ -level, again provided that $\trianglelefteq$ replaces $\cong$:

$$(A \times B) \hookrightarrow C \trianglelefteq A \hookrightarrow (B \hookrightarrow C) \tag{127}$$

Instead of the uncurrying law "symmetrical" *w.r.t.* (127) we have

$$A \hookrightarrow (B \hookrightarrow C) \trianglelefteq 2^A \times ((A \times B) \hookrightarrow C) \tag{128}$$

which is a particular case of

$$A \hookrightarrow D \times (B \hookrightarrow C) \trianglelefteq (A \hookrightarrow D) \times ((A \times B) \hookrightarrow C) \tag{129}$$

(let $D = 1$ and apply basic properties of $\times$ and (125)).

The $\trianglelefteq$-subcalculus encompasses other $\hookrightarrow$ -laws which cannot be adapted from similar results concerning exponentiation. For instance,

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

and yet we have the following map-decomposition law:

$$A \hookrightarrow (B + C) \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C) \tag{130}$$

Another law of this kind is

$$A \hookrightarrow B \times C \ \trianglelefteq \ (A \times B) \hookrightarrow C \tag{131}$$

which induces concrete invariant $fdp \circ dom$ and generalizes (117).

Finally, recall law

$$2^A \ \trianglelefteq_{elems} \ A^\star \tag{132}$$

and consider the following laws recording basic refinements of derived constructs:

$$2^A \ \trianglelefteq_{ran} \ \mathbf{N} \hookrightarrow A \tag{133}$$

(refinement consists of indexing data)

$$A^\star \trianglelefteq \mathbf{N} \hookrightarrow A \tag{134}$$

(refinement consists of rendering sequence order explicit), and

$$A \hookrightarrow B \ \trianglelefteq_{finv} \ B \hookrightarrow 2^A_+ \tag{135}$$

REMARK 15: **A few side notes and some exercises**

- In order to save SETS-expressions from too many parentheses, the following operator priorities are assumed: $^\star > _^- > \times > \hookrightarrow > +$. For instance, expression $X \hookrightarrow A^2 \times B^\star + C$ will abbreviate $(X \hookrightarrow ((A^2) \times (B^\star))) + C$.
- Note that mkf (116) is partial and that its domain of definition is captured by fdp. It can be defined alternatively by

$$mkf(\rho) \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ the(\{b \in B \mid a\ \rho\ b\}) \end{array} \right)_{a \in 2^{\pi 1(\rho)}}$$

  where the is the operator

$$the : 2^A \longrightarrow A \tag{136}$$

  which yields the unique element of a singleton set — $the(\{a\}) = a$ — and is otherwise undefined.
- The $\wedge$-connective in (120) should be regarded as non-strict on its second argument (*e.g.* $F \wedge \bot = F$) due to the partiality of g.
- 2^A_+ in (135) means the non-empty elements of the powerset of A. Similarly, $(B \hookrightarrow C)_+$ will mean $B \hookrightarrow C$ minus the empty mapping.

Exercise 3 Show that, for $B \ncong 0$ and $C \ncong 0$, one has:

$$A \hookrightarrow (B \hookrightarrow C)_+ \cong A \times B \hookrightarrow C$$

Hint: play with the cardinalities of both sides of the equation.
□

Exercise 4 Consider the following Set-facts:

$$A^{B \cup X} \cong A^B \times A^X \quad \Leftarrow \quad B \cap X = \emptyset \tag{137}$$

$$A^n \quad \cong \quad A \times A^{n-1} \tag{138}$$

$$n \neq m \quad \Rightarrow \quad X^n \cap X^m = \emptyset \tag{139}$$

1. Which of these facts conveys our intuition that *no two sequences of different length can be the same sequence*?
2. Show that (137) can be drawn from previous Set-facts.
3. $cons$ is a function which comes to mind in order to inject the righthand side of (138) into its lefthand side. Which function would you suggest in the opposite direction? Justify.

□

Exercise 5 Wishing to prove the following fact, in Set,

$$A^\star \times \mathbb{N} \cong A^\star \times A^\star$$

somebody put forward the following function as a bijection, from right to left:

$$pack(l, l') \stackrel{\text{def}}{=} \langle l \frown l', length(l) + 1 \rangle$$

Show that $pack$ cannot be used for this purpose. **Hint**: show that $pack$ is not surjective.
□

Exercise 6 Show that (131) generalizes (117). **Hint**: let $C = 1$.
□

REMARK 16: **Details about laws (126), (127) and (134)**

Law (126) is justified by the following reasoning:

$$\begin{array}{lcl} A \hookrightarrow (B \times C) & \cong & \bigcup_{K \subseteq A} (B \times C)^K \\ & \cong & \bigcup_{K \subseteq A} (B^K) \times (C^K) \\ & \cong & \{\langle \sigma, \tau \rangle \mid \sigma \in B^K \wedge \tau \in C^K \wedge K \subseteq A\} \\ & = & \{\langle \sigma, \tau \rangle \mid \begin{array}{l} \sigma \in A \hookrightarrow B \wedge \tau \in A \hookrightarrow C \quad \wedge \\ dom(\sigma) = dom(\tau) \end{array} \} \end{array}$$

Defining predicate

$$eqd(\sigma, \tau) \stackrel{\text{def}}{=} dom(\sigma) = dom(\tau) \tag{140}$$

(eqd = equal domains) and

$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ \langle \sigma(a), \tau(a) \rangle \end{array} \right)_{a \in dom(\sigma)} \tag{141}$$

($\bowtie$ "joins" two finite mappings which obey to eqd), we will have:

$$A \hookrightarrow (B \times C) \trianglelefteq^{eqd}_{\bowtie} (A \hookrightarrow B) \times (A \hookrightarrow C)$$

establishing how $\hookrightarrow$ "distributes" over product $\times$.
Concerning law (127), the expected currying and uncurrying maps are, respectively, not surjective and partial — think of the mappings in $A \hookrightarrow (B \hookrightarrow C)$ containing $(\ )$ in their range. For abstraction function

$$uncurry(\sigma) \stackrel{\text{def}}{=} \left(\begin{array}{c} \langle a, b \rangle \\ (\sigma(a))(b) \end{array} \right)_{a \in dom(\sigma) \wedge b \in dom(\sigma(a))}$$

the concrete invariant is:

$$\phi(\sigma) \stackrel{\text{def}}{=} (\ ) \notin rng(\sigma)$$

$\mathbb{N} \hookrightarrow A$ in (134) is an adequate representation for $A^\star$ provided some total ordering criterion is assumed in $\mathbb{N}$, *e.g.* the $a < b$ ordering. The abstraction map will then be

$$f(\sigma) \stackrel{\text{def}}{=} \left\{ \begin{array}{rcll} \sigma = (\ ) & \Rightarrow & <> & \\ \neg(\sigma = (\ )) & \Rightarrow & let & m = Min(dom(\sigma)) \\ & & in & <\sigma(m)> \frown f(\sigma \setminus \{m\}) \end{array} \right. \tag{142}$$

where Min computes the least number in a non-empty set of numbers:

$$Min(S) \stackrel{\text{def}}{=} \left\{ \begin{array}{rcll} card(S) = 1 & \Rightarrow & the(S) & \\ \neg(card(S) = 1) & \Rightarrow & let & x \in dom(\sigma) \\ & & & m = Min(S - \{x\}) \\ & & in & \left\{ \begin{array}{rcl} x < m & \Rightarrow & x \\ \neg(x < m) & \Rightarrow & m \end{array} \right. \end{array} \right.$$

REMARK 17: **Law (129) and others**

The concrete invariant induced by (129) is

$$\phi(\sigma, \sigma') \stackrel{\text{def}}{=} 2^{\pi_1}(dom(\sigma')) \subseteq dom(\sigma) \tag{143}$$

This invariant recommends that all $a \in A$ found in $\sigma \in A \hookrightarrow D$ which cannot be found in $\sigma' \in A \times B \hookrightarrow C$ are exactly the entries, at abstract level, to be mapped to the empty map (in $B \hookrightarrow C$) — *cf.* the corresponding abstraction map, a kind of "nested join":

$$\bowtie (\sigma, \tau) \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ \langle \sigma(a), \left(\begin{array}{c} \pi_2(p) \\ \tau(p) \end{array} \right)_{p \in sel1(a,\tau)} \rangle \end{array} \right)_{a \in dom(\sigma)} \tag{144}$$

where the following auxiliary function can be found:

$$sel1(a, \tau) \stackrel{\text{def}}{=} \{\langle a', b\rangle \in dom(\tau) \mid\ a' = a\}$$

The abstraction invariant implicit in law (130) consists of retrieve function

$$\begin{array}{lcl} \stackrel{+}{\bowtie} (\sigma, \tau) & \stackrel{\text{def}}{=} & i_1 \circ \sigma \cup i_2 \circ \tau \\ & = & (A \hookrightarrow i_1)(\sigma) \cup (A \hookrightarrow i_2)(\tau) \end{array}$$

coupled with concrete invariant

$$djd(\sigma, \tau) \stackrel{\text{def}}{=} dom(\sigma) \cap dom(\tau) = \emptyset$$

Two other, rather elementary $\trianglelefteq$-results have to do with the explicit introduction of additive or multiplicative redundancy in data models:

$$A \trianglelefteq^{\lambda x.x=\langle 1,a\rangle}_{\lambda\langle i,a\rangle.a} A + B \tag{145}$$

$$B \trianglelefteq^{\lambda x.x=\langle 2,b\rangle}_{\lambda\langle i,b\rangle.b} A + B \tag{146}$$

(injecting A into a variant record structure), and

$$A \trianglelefteq_{\pi_1} A \times B \tag{147}$$

$$B \trianglelefteq_{\pi_2} A \times B \tag{148}$$

(coupling A with another data structure B).
For instance, what is the "difference" between

```
union X {
   int tag; /* 1,2 */
   A ifA;
   B ifB;
};
```

and

```
struct X {
   A *ifA;
   B *ifB;
};
```

— *i.e.* between $A + B$ and $(A + 1) \times (B + 1)$? Let us do some calculations:

$$\begin{array}{lcl} (A + 1) \times (B + 1) & \cong & A \times B + A \times 1 + 1 \times B + 1 \\ & \cong & A \times B + A + B + 1 \\ & = & (A + B) + (A \times B + 1) \end{array}$$

Therefore,

$$A + B \quad \trianglelefteq \quad (A + 1) \times (B + 1)$$

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

A SETS Refinement Target: the Relational Database Model

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

- relations — $2^{A_1 \times \times \ldots A_n}$ where every A_i is regarded as an “atomic type” in the target relational environment;
- mappings — $(A_1 \times \ldots \times A_n) \hookrightarrow (B_1 \times \ldots \times B_m)$ where every A_i and B_j is regarded as an “atomic type” in the target relational environment.

Then:

- Laws such as (126,128,129,130) and (134) 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,

  $$A \hookrightarrow B \trianglelefteq 2^{A \times B}$$

  our $\trianglelefteq$-subcalculus fragment above is useful in refining elaborate, finite-mapping/sequence based data models into relational database schemata.
- It has been proved elsewhere [Rod93] that 3NF is guaranteed by mere application of the above laws, among others.

Structural Calculation

Motivation: we want to refine $A \hookrightarrow 2^B$ (A-indexed families of B-subsets) and we know that $2^B \trianglelefteq B^\star$ holds.

Can we infer

$$A \hookrightarrow 2^B \trianglelefteq A \hookrightarrow B^\star?$$

Such an inference will be allowed provided functor $\mathcal{F}(X) = A \hookrightarrow X$ is $\trianglelefteq$-monotone.

In fact (see theorems 1 and 2) we may state that, for every functor $\mathcal{F}$ involving either polynomial functors (see below) or the powerset functor, if

$$A \trianglelefteq_f^\phi B$$

holds, then

$$\mathcal{F}(A) \trianglelefteq_{\mathcal{F}(f)}^{\mathcal{F}(\phi)} \mathcal{F}(B)$$

will hold.

Both $\mathcal{F}(f)$ and $\mathcal{F}(\phi)$ can be structurally synthesized according to the "functorial abstract invariant calculus" summarized in the following table:

$\mathcal{F}(X)$	$\mathcal{F}(f)$	$\mathcal{F}(\phi)$
X	f	ϕ
C	1_C	$\lambda c.TRUE$
2^X	$2^f = \lambda s.\{f(x) \mid x \in s\}$	$2^\phi = \lambda s.\forall b \in s : \phi(b)$
$X^\star$	$f^\star = \lambda l.<f(x) \mid x \leftarrow l>$	$\phi^\star = \lambda l.\forall 1 \leq i \leq length(l) : \phi(l(i))$
X^C	$f^C = \lambda\sigma.f \circ \sigma = \lambda\sigma.\begin{pmatrix} c \\ f(\sigma(c)) \end{pmatrix}_{c \in C}$	$\phi^C = \lambda\sigma.\forall c \in C : \phi(\sigma(c))$
$C \hookrightarrow X$	$C \hookrightarrow f = \lambda\sigma.f \circ \sigma = \lambda\sigma.\begin{pmatrix} c \\ f(\sigma(c)) \end{pmatrix}_{c \in dom(\sigma)}$	$C \hookrightarrow \phi = \lambda\sigma.\forall x \in rng(\sigma) : \phi(x)$
$\mathcal{G}(X) \times \mathcal{H}(X)$	$\mathcal{G}(f) \times \mathcal{H}(f)$	$\lambda(x,y).\mathcal{G}(\phi)(x) \wedge \mathcal{H}(\phi)(y)$
$\mathcal{G}(X) + \mathcal{H}(X)$	$\mathcal{G}(f) + \mathcal{H}(f)$	$\lambda x.\begin{cases} x = \langle 1, a\rangle & \Rightarrow \quad \mathcal{G}(\phi)(a) \\ x = \langle 2, b\rangle & \Rightarrow \quad \mathcal{H}(\phi)(b) \end{cases}$
$\mathcal{G}(X) + 1$	$\lambda x.\begin{cases} x = \langle 2, NIL\rangle & \Rightarrow \quad x \\ x = \langle 1, a\rangle & \Rightarrow \quad \langle 1, \mathcal{G}(f)(a)\rangle \end{cases}$	$\lambda x.\begin{cases} x = \langle 2, NIL\rangle & \Rightarrow \quad TRUE \\ x = \langle 1, a\rangle & \Rightarrow \quad \mathcal{G}(\phi)(a) \end{cases}$

REMARK 18: **Structural abstraction invariant calculation — main results**

The following theorem provides an elegant, functorial strategy for computing complex abstraction invariants throughout a reification process.

Theorem 1 *Let $\mathcal{F}$ be a polynomial (endo)functor in Set. If*

$$A \trianglelefteq_f^{\phi} B \tag{149}$$

then

$$\mathcal{F}(A) \trianglelefteq_{\mathcal{F}(f)}^{\mathcal{F}(\phi)} \mathcal{F}(B) \tag{150}$$

Proof*: It is performed by induction on the structure of polynomial functor $\mathcal{F}$. Sum and product (and exponentiation) are dealt with in theorem 2.*
The base case concerning the identity functor $\mathcal{F}(X) = X$ is trivial, since $\mathcal{F}(f) = f$ and $\mathcal{F}(\phi) = \phi$, i.e. *(150) reduces to (149). The other base case concerns the constant functor $\mathcal{F}(X) = C$. We have $\mathcal{F}(f) = 1_C$ and $\mathcal{F}(\phi) = \lambda b.TRUE$, (150) holding trivially.* □

The scope of application of this result is fairly wide:

- Although $\mathcal{F}(X) = 2^X$ is not polynomial [MA86], (150) still holds for this functor, for finite A and B [Jou92].
- Moreover, $\mathcal{F}(X) = X^\star$ can be shown to be canonically polynomial, up to isomorphism — *cf.* (112) — such that every $C_i = 1$, *cf.*

$$\begin{aligned} X^\star &\cong \sum_{i \geq 0} X^i \\ &\cong \sum_{i \geq 0} 1 \times X^i \end{aligned}$$

- Finally, $\mathcal{F}(X) = C \hookrightarrow X$ is also polynomial, up to isomorphism, for C a finite exponent — since $C \hookrightarrow X \cong (X+1)^C$ (110) and, via *Newton's binomial formula* (34),

$$(X+1)^{|C|} \cong \sum_{p=0}^{|C|} {}^{|C|}C_p \times X^{n-p} \times 1^p \cong \sum_{p=0}^{|C|} {}^{|C|}C_p \times X^{n-p}$$

  A similar result can be drawn from (113), whereby one has

$$C \hookrightarrow X \cong \sum_{K \subseteq C} X^K$$

Back to

$$A \hookrightarrow 2^B \trianglelefteq_f A \hookrightarrow B^\star$$

it is now straightforward to calculate, from $2^B \trianglelefteq_{elems} B^\star$ the implied abstraction function [a]:

$$\begin{aligned} f &= A \hookrightarrow elems \\ &= \lambda\sigma.(elems \circ \sigma) \\ &= \lambda\sigma. \left(\begin{array}{c} a \\ elems(\sigma(a)) \end{array} \right)_{a \in dom(\sigma)} \end{aligned}$$

[a] We are ignoring concrete invariants here because they are trivially the everywhere $TRUE$ predicate.

REMARK 19: **Monotonicity of Set-primitive constructs**

Theorem 2 ($\trianglelefteq$-Monotonicity of Set-constructs) *The Set-constructs $\times$ and $+$ are monotone* w.r.t. *the $\trianglelefteq$-ordering (118); exponentiation requires isomorphical exponents.*
That is, given Set-objects A, B, X, Y, M, N such that $A \trianglelefteq X$, $B \trianglelefteq Y$ and $M \cong N$, then facts

$$A \times B \trianglelefteq X \times Y \tag{151}$$
$$A + B \trianglelefteq X + Y \tag{152}$$
$$A^M \trianglelefteq X^N \tag{153}$$

hold.
Outline of Proof*: Let*

$$A \trianglelefteq_f^{\phi} X, \;\; B \trianglelefteq_g^{\varphi} Y, \;\; M \cong_h N \tag{154}$$

Then

1. *Equation (151):* $A \times B \trianglelefteq_{f \times g}^{\phi \times \varphi} X \times Y$
   where the product of two functions f and g is "parallel application" (9) and the product of two predicates ϕ and φ is "parallel conjunction"

   $$(\phi \times \varphi)\langle a, b\rangle = \phi(a) \wedge \varphi(b)$$

2. *Equation (152):* $A + B \trianglelefteq_{f+g}^{\phi+\varphi} X + Y$
   where the sum of two functions f and g is given by (12) and the sum of two predicates ϕ and φ is

   $$\begin{cases} (\phi + \varphi)\langle 1, a\rangle &= \phi(a) \\ (\phi + \varphi)\langle 2, b\rangle &= \varphi(b) \end{cases}$$

3. *Equation (153): Since M and N have the same cardinality, say m, then*

   $$A^M \cong A^m \cong \underbrace{A \times \ldots \times A}_{m}$$

   and

   $$X^N \cong X^m \cong \underbrace{X \times \ldots \times X}_{m}$$

   cf. *(102) below. That is, (153) can be reduced to a finitary version of (151) in which we have $A^m \trianglelefteq_{f^m}^{\phi^m} X^m$ where*

   $$f^m\langle a_1, \ldots, a_m\rangle = \langle f(a_1), \ldots, f(a_m)\rangle \tag{155}$$
   $$\phi^m\langle a_1, \ldots, a_m\rangle = \forall 1 \leq i \leq m : \phi(a_i)$$

   Alternatively, we may define the overall abstraction map t and invariant τ in

   $$A^M \trianglelefteq_t^{\tau} X^N$$

   as follows, for $\sigma \in X^N$ and h the isomorphism between M and N (154):

   $$t(\sigma) = f \circ \sigma \circ h^{-1}$$
   $$\tau(\sigma) = \forall n \in dom(\sigma) : \phi(\sigma(n))$$

It remains to be proved that all composite abstraction maps above are surjective, which is not hard work [Jou92].
□

Thus,

- the components of each data domain of a Set expression can be refined in isolation;
- this allows for the stepwise introduction of redundancy in formal models of software.

Examples of Structural Calculation in SETS

The "linked-list" implementation of $A^\star$

We know that...

$$\begin{aligned} A^\star &= \bigcup_{n=0}^{\infty} A^n \\ &\cong \sum_{n=0}^{\infty} A^n \\ &\cong 1 + A + A^2 + \ldots \end{aligned} \tag{171}$$

cf. (112), (104) and (105), respectively.
Let us introduce two auxiliary variables L and N,

$$\begin{aligned} N &= A + A^2 + \ldots \\ L &= 1 + N \end{aligned} \tag{172}$$

so that we have, by substitution,

$$A^\star \cong L \tag{173}$$

and

$$\begin{aligned} N &= A + A^2 + \ldots \\ &\cong A \times 1 + A \times A + A \times A^2 + \ldots \\ &\cong A \times (1 + A + A^2 + \ldots) \\ &\cong A \times L \end{aligned} \tag{174}$$

REMARK 20: **Abstraction invariant cookbook**

Abstraction invariant components f and ϕ in (119), which can be left out in "quick" SETS calculation, will be explicitly demanded by algorithm synthesis, as we shall see.

For this purpose, this and Remark 21 provide a quick reference list, for the basic laws of the calculus, of the relevant abstraction map f and concrete invariant ϕ. Equation numbers are used consistently in the identification of such functions and predicates: $f_{(n)}$ and $\phi_{(n)}$ denote respectively the abstraction map and concrete invariant associated with law (n).

Abstraction functions

$$
\begin{array}{rcl}
f_{(94)} & = & [\pi_2, \pi_1] \\
f_{(95)} & = & [\pi_1 \circ \pi_1, [\pi_2 \circ \pi_1, \pi_2]] \\
f_{(96)} & = & \pi_1 \\
f_{(97)} & = & i_2 \mid i_1 \\
f_{(98)} & = & (i_1 \mid i_2 \circ i_1) \mid (i_2 \circ i_2) \\
f_{(99)} & = & i_1 \\
f_{(100)} & = & 1_0 \\
f_{(101)} & = & [\pi_1, i_1 \circ \pi_2] \mid [\pi_1, i_2 \circ \pi_2] \\
f_{(102)} & = & \lambda\sigma.\langle \sigma(1), \ldots, \sigma(n) \rangle \\
f_{(103)} & = & \lambda x.x \\
f_{(104)} & = & \lambda x.\left(\quad \right) \\
f_{(105)} & = & \lambda a.\left(\begin{array}{c} NIL \\ a \end{array} \right) \\
f_{(106)} & = & \lambda\langle x, y \rangle.\left(\begin{array}{c} a \\ \langle x(a), y(a) \rangle \end{array} \right)_{a \in A} \\
f_{(107)} & = & uncurry \\
f_{(108)} & = & \lambda\langle \sigma_1, \sigma_2 \rangle.A^{i_1}(\sigma_1) \cup A^{i_2}(\sigma_2) \\
f_{(109)} & = & \lambda x.\left(\begin{array}{c} a \\ NIL \end{array} \right)_{a \in A} \\
f_{(110)} & = & \lambda\sigma.\left(\begin{array}{c} a \\ b \end{array} \right)_{a \in A \wedge \sigma(a) = i_1(b)} \\
f_{(111)} & = & \lambda\sigma.\left(\begin{array}{c} c \\ \left(\begin{array}{c} \pi_1(p) \\ \sigma(p) \end{array} \right)_{p \in dom(\sigma) \wedge \pi_2(p) = c} \end{array} \right)_{c \in C} \\
f_{(117)} & = & mkf \\
f_{(122)} & = & \lambda x.\left(\quad \right) \\
f_{(123)} & = & \lambda x.\left\{ \begin{array}{lcl} x = i_1(a) & \Rightarrow & \left(\begin{array}{c} NIL \\ a \end{array} \right) \\ \neg(x = i_1(a)) & \Rightarrow & \left(\quad \right) \end{array} \right. \\
f_{(124)} & = & \lambda\langle \sigma_1, \sigma_2 \rangle.(i_1 \hookrightarrow A)(\sigma_1) \cup (i_2 \hookrightarrow A)(\sigma_2) \\
f_{(125)} & = & dom \\
f_{(126)} & = & \bowtie \\
f_{(127)} & = & \lambda\sigma.\left(\begin{array}{c} \langle a, b \rangle \\ (\sigma(a))(b) \end{array} \right)_{a \in dom(\sigma) \wedge b \in dom(\sigma(a))} \\
f_{(129)} & = & \bowtie_{\not n} \\
f_{(130)} & = & \stackrel{+}{\bowtie}
\end{array}
$$

REMARK 21: **Abstraction invariant cookbook (continuation)**

$$
\begin{array}{lcl}
f_{(131)} & = & \lambda\sigma.\left(\begin{array}{c} a \\ \langle b, \sigma(\langle a, b\rangle)\rangle \end{array}\right)_{\langle a,b\rangle \in dom(\sigma)} \\
f_{(132)} & = & elems \\
f_{(135)} & = & finv \\
f_{(147)} & = & \pi_1 \\
f_{(148)} & = & \pi_2
\end{array}
$$

Concrete Invariants

$$
\begin{array}{lcl}
\phi_{(117)} & = & fdp \\
\phi_{(126)} & = & eqd \\
\phi_{(127)} & = & \lambda\sigma.\,(\quad) \notin rng(\sigma) \\
\phi_{(129)} & = & dpi \\
\phi_{(130)} & = & djd \\
\phi_{(131)} & = & fdp \circ dom \\
\phi_{(135)} & = & mdjfam
\end{array}
$$

Auxiliary Functions and Predicates

$$djd(\sigma,\tau) \stackrel{\text{def}}{=} dom(\sigma) \cap dom(\tau) = \emptyset \tag{156}$$

$$dpi(\sigma,\tau) \stackrel{\text{def}}{=} 2^{\pi_1}(dom(\tau)) \subseteq dom(\sigma) \tag{157}$$

$$eqd(\langle\sigma,\tau\rangle) \stackrel{\text{def}}{=} dom(\sigma) = dom(\tau) \tag{158}$$

$$elems(b) \stackrel{\text{def}}{=} \{b(i) \mid i \in length(b)\} \tag{159}$$

$$fdp(r) \stackrel{\text{def}}{=} \forall t, t' \in r : \pi_1(t) = \pi_1(t') \Rightarrow \pi_2(t) = \pi_2(t') \tag{160}$$

$$mdjfam(\sigma) \stackrel{\text{def}}{=} \forall b, b' \in dom(\sigma) : b \neq b' \Rightarrow \sigma(b) \cap \sigma(b') = \emptyset \tag{161}$$

$$i_1(a) \stackrel{\text{def}}{=} \langle 1, a\rangle \tag{162}$$

$$i_2(a) \stackrel{\text{def}}{=} \langle 2, a\rangle \tag{163}$$

$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ \langle \sigma(a), \tau(a)\rangle \end{array}\right)_{a \in dom(\sigma)} \tag{164}$$

$$\overset{+}{\bowtie}(\sigma,\tau) \stackrel{\text{def}}{=} i_1 \circ \sigma \cup i_2 \circ \tau \tag{165}$$

$$\underset{\pi}{\bowtie}(\sigma,\tau) \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ \langle \sigma(a), \left(\begin{array}{c} b \\ \tau(a,b) \end{array}\right)_{\langle a,b\rangle \in sel(a,\tau)} \rangle \end{array}\right)_{a \in dom(\sigma)} \tag{166}$$

$$mkf(\rho) \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ the(\{b \in B \mid a\ \rho\ b\}) \end{array}\right)_{a \in 2^{\pi_1}(\rho)} \tag{167}$$

$$sel(a,\tau) \stackrel{\text{def}}{=} \{\langle a', b\rangle \in dom(\tau) \mid a' = a\} \tag{168}$$

$$uncurry(t) \stackrel{\text{def}}{=} \left(\begin{array}{c} \langle b, c\rangle \\ t(b)(c) \end{array}\right)_{b \in B \wedge c \in C} \tag{169}$$

$$finv(\sigma) \stackrel{\text{def}}{=} \bigcup_{b \in dom(\sigma)} \left(\begin{array}{c} a \\ b \end{array}\right)_{a \in \sigma(b)} \tag{170}$$

where we resorted to laws (105), (138) e (104), to the finitary version of (101), and where step (174) was obtained by "folding" over step (171). In summary,

$$\begin{array}{lll} A^\star & \cong & L \\ & where & L = 1 + N \\ & & N \cong A \times L \end{array} \qquad (175)$$

Recalling the table of p.12, L (175) can be encoded thus, in *e.g.* PASCAL:

```
L = ^N;
N = record
      P: A;
      S: L
    end;
```

Note how recursion became explicit by calculation from $A^\star$, a well-known programming device for "pointer"-based representation of arbitrarily large collections of data.

Relational Implementation of BAMS (1)

Re-writing (1) as a Set-expression we obtain:

$$BAMS = AccNr \hookrightarrow (2^{AccHolder} \times Amount)$$

Calculating a relational implementation:

$$\begin{array}{rl} BAMS & = AccNr \hookrightarrow (2^{AccHolder} \times Amount) \\ (94) & \cong_1 AccNr \hookrightarrow (Amount \times 2^{AccHolder}) \\ (125) & \cong_2 AccNr \hookrightarrow (Amount \times (AccHolder \hookrightarrow 1)) \\ (129) & \trianglelefteq_3 (AccNr \hookrightarrow Amount) \times ((AccNr \times AccHolder) \hookrightarrow 1)) \\ (125) & \cong_4 (AccNr \hookrightarrow Amount) \times (2^{AccNr \times AccHolder}) \\ (117) & \trianglelefteq_5 2^{AccNr \times Amount} \times 2^{AccNr \times AccHolder} \\ & = BAMS_5 \end{array}$$

where:

$$
\begin{array}{ll}
\cong_1 & \left\{ \begin{array}{l} f_1 = AccNr \hookrightarrow [\pi_2, \pi_1] \\ \phi_1 = \lambda x \cdot TRUE \end{array} \right. \\
\cong_2 & \left\{ \begin{array}{l} f_2 = AccNr \hookrightarrow (id \times dom) \\ \phi_2 = \lambda x \cdot TRUE \end{array} \right. \\
\trianglelefteq_3 & \left\{ \begin{array}{l} f_3 = \bowtie_{\mathit{ñ}} \\ \phi_3 = dpi \end{array} \right. \\
\cong_4 & \left\{ \begin{array}{l} f_4 = id \times (\lambda r \cdot \left(\begin{array}{c} p \\ NIL \end{array} \right)_{p \in r}) \\ \phi_4 = \lambda x \cdot TRUE \end{array} \right. \\
\trianglelefteq_5 & \left\{ \begin{array}{l} f_5 = mkf \times id \\ \phi_5 = fdp \times (\lambda x \cdot TRUE) \end{array} \right.
\end{array}
$$

Calculating the overall abstraction invariant: abstraction function...

$$
\begin{array}{rcl}
f & = & f_1 \circ f_2 \circ f_3 \circ f_4 \circ f_5 \\
& = & (AccNr \hookrightarrow [\pi_2, \pi_1]) \circ (AccNr \hookrightarrow (id \times dom)) \circ \bowtie_{\mathit{ñ}} \circ \\
& & (id \times \lambda r \cdot \left(\begin{array}{c} p \\ NIL \end{array} \right)_{p \in r}) \circ (mkf \times id) \\
& = & (AccNr \hookrightarrow [\pi_2, \pi_1] \circ (id \times dom)) \circ \bowtie_{\mathit{ñ}} \circ (mkf \times \lambda r \cdot \left(\begin{array}{c} p \\ NIL \end{array} \right)_{p \in r})
\end{array}
$$

i.e.

$$
\begin{array}{rcl}
f(\langle \rho, \gamma \rangle) & = & let \quad x = \left(\begin{array}{c} a \\ \langle mkf(\rho)(a), \left(\begin{array}{c} \pi_2(p) \\ NIL \end{array} \right)_{p \in \gamma \wedge \pi_1(p) = a} \rangle \end{array} \right)_{a \in 2^{\pi_1}(\rho)} \\
& & in \quad (AccNr \hookrightarrow [\pi_2, \pi_1] \circ (id \times dom))(x) \\
& = & \left(\begin{array}{c} a \\ \langle \{\pi_2(p) \mid \ p \in \gamma \wedge \pi_1(p) = a\}, the(\{b \mid \ a \ \rho \ b\}) \rangle \end{array} \right)_{a \in 2^{\pi_1}(\rho)}
\end{array}
$$

... and concrete invariant:

$$
\begin{array}{rcl}
\phi(\langle \rho, \gamma \rangle) & = & \phi_5(\langle \rho, \gamma \rangle) \wedge \\
& & \phi_4(f_5(\langle \rho, \gamma \rangle)) \wedge \\
& & \phi_3(f_4(f_5(\langle \rho, \gamma \rangle))) \wedge
\end{array}
$$

$$
\begin{array}{rl}
& \phi_2(f_3(f_4(f_5(\langle\rho,\gamma\rangle)))) \wedge \\
& \phi_1(f_2(f_3(f_4(f_5(\langle\rho,\gamma\rangle))))) \\
= & \phi_5(\langle\rho,\gamma\rangle) \wedge \phi_3(f_4(f_5(\langle\rho,\gamma\rangle))) \\
= & fdp(\rho) \wedge \phi_3(f_4(\langle mkf(\rho),\gamma\rangle)) \\
= & fdp(\rho) \wedge dpi(\langle mkf(\rho), \left(\begin{array}{c} p \\ NIL \end{array} \right)_{p\in\gamma} \rangle) \\
= & fdp(\rho) \wedge 2^{\pi_1}(\gamma) \subseteq dom(mkf(\rho)) \\
= & fdp(\rho) \wedge 2^{\pi_1}(\gamma) \subseteq 2^{\pi_1}(\rho)
\end{array}
$$

. . . clearly:

- imposing a functional dependence on the "amounts table" ρ (=the balance of any account is not ambiguous),
- allowing for more than one holder per account (*cf.* "account-holder table" γ),
- guaranteeing that every account held by somebody is a valid account.

Note how much work of the alternative "invent & verify" refinement strategy was saved by calculation:

- invent $BAMS_5$, invent f and invent ϕ;
- prove that f is well defined over ϕ and that it is surjective.

Relational Implementation of a "Toy" Production Database

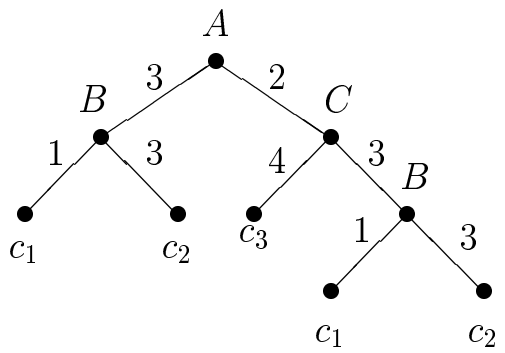
Consider the following data model for a "toy factory" production database:

$$
\begin{array}{rcl}
PPD & \cong & Stock \times Pricelist \times Equip \\
Stock & \cong & \#Unit \hookrightarrow \mathbf{N} \\
Pricelist & \cong & \#Comp \hookrightarrow \mathbf{N} \\
Equip & \cong & \#Equip \hookrightarrow (\#Unit \hookrightarrow \mathbf{N})_+ \\
\#Unit & \cong & \#Comp + \#Equip
\end{array}
$$

that is:

- this factory builds *equipments* whose production involves individual *components* obtained from a second source;
- each individual component has a *cost* and is stocked in some *quantity*;
- components participate, in fixed quantities, in the production of *equipments* (*e.g.* PC board ref. X has n integrated circuits of ref. Y);
- equipments may contain, in fixed quantities, other equipments as *sub-blocks* (*e.g.* personal computer ref. Z has m PC boards of ref. X).
- (sub)equipments are stocked along with components in order to be readily available for production.

In summary, a production tree must exist for each equipment which may involve individual components and/or other production sub-trees, for instance:



The Calculation

Let us derive a relational implementation of PPD by "on the fly" calculation (see remarks 22 and 23 for details):

— $Stock$ e $Pricelist$:

$$\begin{aligned} & Stock \times Pricelist \\ & \cong \ (\#Unit \hookrightarrow \mathbf{N}) \times (\#Comp \hookrightarrow \mathbf{N}) \\ & \cong \ ((\#Comp + \#Equip) \hookrightarrow \mathbf{N}) \times (\#Comp \hookrightarrow \mathbf{N}) \end{aligned}$$

$$
\begin{array}{lll}
& \cong & (\#Comp \xrightarrow{} \mathbf{N}) \times (\#Equip \xrightarrow{} \mathbf{N}) \times (\#Comp \xrightarrow{} \mathbf{N}) \\
& \cong & (\#Comp \xrightarrow{} \mathbf{N})^2 \times (\#Equip \xrightarrow{} \mathbf{N}) \\
& \trianglelefteq & 2^{2\times\#Comp\times\mathbf{N}} \times 2^{\#Equip\times\mathbf{N}}
\end{array}
$$

— $Equip$:

$$
\begin{array}{lll}
Equip & \cong & \#Equip \xrightarrow{} (\#Unit \xrightarrow{} \mathbf{N})_+ \\
& \cong & (\#Equip \times \#Unit) \xrightarrow{} \mathbf{N} \\
& \cong & (\#Equip \times \#Comp + \#Equip \times \#Equip) \xrightarrow{} \mathbf{N} \\
& \cong & (\#Equip \times \#Comp \xrightarrow{} \mathbf{N}) \times (\#Equip \times \#Equip \xrightarrow{} \mathbf{N}) \\
& \trianglelefteq & 2^{\#Equip\times\#Comp\times\mathbf{N}} \times 2^{\#Equip\times\#Equip\times\mathbf{N}}
\end{array}
$$

By compound reasoning we will then have:

$$PPD \trianglelefteq RPPD$$

(read $RPPD$ as "relational database refinement of PPD") where

$$
\begin{array}{lllll}
RPPD & \cong & CC: & 2^{Row1} & \times \quad \textit{/*component cost/stock table */} \\
& & SE: & 2^{Row2} & \times \quad \textit{/*equipment stock table */} \\
& & EC: & 2^{Row3} & \times \quad \textit{/*equipment-to-component */} \\
& & & & \quad \textit{/*decomposition table */} \\
& & EE: & 2^{Row4} & \quad \textit{/*equipment-to-(sub)equipment */} \\
& & & & \quad \textit{/*decomposition table */}
\end{array}
$$

and where

$$
\begin{array}{lllll}
Row1 & \cong & Co: & \#Comp & \times \\
& & QC: & \mathbf{N} & \times \quad \textit{/*amount or cost, according to ... */} \\
& & T: & 2 & \times \quad \textit{/*... this tag! */} \\
Row2 & \cong & Eq: & \#Equip & \times \\
& & St: & \mathbf{N} & \\
Row3 & \cong & Eq: & \#Equip & \times \\
& & Co: & \#Comp & \times \\
& & Nr: & \mathbf{N} & \\
Row4 & \cong & Eq: & \#Equip & \times \\
& & Se: & \#Equip & \times \\
& & Nr: & \mathbf{N} &
\end{array}
$$

REMARK 22: **Detailed calculation of $RPPD$ from PPD**

Calculation steps:

$$
\begin{array}{rll}
PPD & \cong & ((\#Comp + \#Equip) \hookrightarrow \mathbb{N}) \times \\
 & & (\#Comp \hookrightarrow \mathbb{N}) \times \\
 & & (\#Equip \hookrightarrow ((\#Comp + \#Equip) \hookrightarrow \mathbb{N})_{+}) \\
 & \cong_1 & ((\#Comp \hookrightarrow \mathbb{N}) \times (\#Equip \hookrightarrow \mathbb{N})) \times \\
 & & (\#Comp \hookrightarrow \mathbb{N}) \times \\
 & & ((\#Equip \times (\#Comp + \#Equip)) \hookrightarrow \mathbb{N})) \\
 & \cong_2 & ((\#Comp \hookrightarrow \mathbb{N}) \times (\#Equip \hookrightarrow \mathbb{N})) \times \\
 & & (\#Comp \hookrightarrow \mathbb{N}) \times \\
 & & ((\#Equip \times \#Comp) + (\#Equip \times \#Equip)) \hookrightarrow \mathbb{N})) \\
 & \cong_3 & ((\#Comp \hookrightarrow \mathbb{N}) \times (\#Equip \hookrightarrow \mathbb{N})) \times \\
 & & (\#Comp \hookrightarrow \mathbb{N}) \times \\
 & & (((\#Equip \times \#Comp) \hookrightarrow \mathbb{N}) \times ((\#Equip \times \#Equip) \hookrightarrow \mathbb{N})) \\
 & \cong_4 & ((\#Comp \hookrightarrow \mathbb{N}) \times (\#Comp \hookrightarrow \mathbb{N})) \times \\
 & & (\#Equip \hookrightarrow \mathbb{N}) \times \\
 & & ((\#Equip \times \#Comp) \hookrightarrow \mathbb{N}) \times \\
 & & ((\#Equip \times \#Equip) \hookrightarrow \mathbb{N}) \\
 & \trianglelefteq_5 & (\#Comp \hookrightarrow \mathbb{N})^2 \times \\
 & & 2^{\#Equip \times \mathbb{N}} \times \\
 & & 2^{(\#Equip \times \#Comp) \times \mathbb{N}} \times \\
 & & 2^{(\#Equip \times \#Equip) \times \mathbb{N}} \\
 & \trianglelefteq_6 & ((2 \times \#Comp) \hookrightarrow \mathbb{N}) \times \\
 & & 2^{\#Equip \times \mathbb{N}} \times \\
 & & 2^{\#Equip \times \#Comp \times \mathbb{N}} \times \\
 & & 2^{\#Equip \times \#Equip \times \mathbb{N}} \\
 & \trianglelefteq_7 & 2^{2 \times \#Comp \times \mathbb{N}} \times \\
 & & 2^{\#Equip \times \mathbb{N}} \times \\
 & & 2^{\#Equip \times \#Comp \times \mathbb{N}} \times \\
 & & 2^{\#Equip \times \#Equip \times \mathbb{N}}
\end{array}
$$

Step by step abstraction invariants [a]:

$$
\begin{array}{rl}
\cong_1 & \{ \ f_1 = f_{(124)} \times id \times h \\
\cong_2 & \{ \ f_2 = id \times id \times (f_{(101)} \hookrightarrow \mathbb{N}) \\
\cong_3 & \{ \ f_3 = id \times id \times f_{(124)} \\
\cong_4 & \{ \ f_4 = \lambda\langle\langle\rho_1, \rho_2\rangle, \rho_3, \rho_4, \rho_5\rangle \cdot \langle\langle\rho_1, \rho_3\rangle, \rho_2, \langle\rho_4, \rho_5\rangle\rangle \\
\trianglelefteq_5 & \left\{ \begin{array}{l} f_5 = id \times mkf \times mkf \times mkf \\ \phi_5 = (\lambda x \cdot TRUE) \times fdp \times fdp \times fdp \end{array} \right. \\
\cong_6 & \{ \ f_6 = f_{(111)} \times id \times 2^j \times 2^j \\
\trianglelefteq_7 & \left\{ \begin{array}{l} f_7 = (mkf \circ 2^j) \times id \times id \times id \\ \phi_7 = (fdp \circ 2^j) \times \Pi_{i=1}^{3} (\lambda x \cdot TRUE) \end{array} \right.
\end{array}
$$

[a] Omitting "everywhere $TRUE$" concrete invariants for the sake of brevity. Variable j abbreviates $\lambda\langle a, b, c\rangle.\langle\langle a, b\rangle, c\rangle$ and h is the abstraction function implicit in the result of Exercise 3, p.40 — see (177).

REMARK 23: **Detailed calculation of $RPPD$ from PPD (continuation)**

Overall abstraction invariant synthesis:

- Abstraction function:

$$\begin{array}{lcl} f & = & f_1 \circ f_2 \circ f_3 \circ f_4 \circ f_5 \circ f_6 \circ f_7 \\ & = & (f_{(124)} \times id \times (h \circ (i \hookrightarrow \mathbb{N}) \circ f_{(124)})) \circ f_4 \circ ((f_{(111)} \circ mkf \circ 2^j) \times id \times mkf \circ 2^j \times mkf \circ 2^j) \end{array}$$

- Concrete invariant:

$$\phi = (fdp \circ 2^j) \times fdp \times (fdp \circ 2^j) \times (fdp \circ 2^j) \tag{176}$$

for

$$h(\sigma) \stackrel{\text{def}}{=} \left(\begin{pmatrix} b \\ \sigma(a,b) \end{pmatrix}^{a}_{\langle a,b\rangle \in dom(\sigma)} \right)_{a \in 2^{\pi_1(dom(\sigma))}} \tag{177}$$

and j function

$$\lambda\langle a,b,c\rangle.\langle\langle a,b\rangle,c\rangle \tag{178}$$

as we have said.

Normally, there is no advantage in expanding complex abstraction functions and invariants, such as f and ϕ above, until explicitly demanded by algorithm calculation, a topic which will be addressed later on [a]. Nevertheless, it will not be difficult to expand, for instance, term $fdp \circ 2^j$ in (176) into:

$$\begin{array}{lcl} (fdp \circ 2^j)(r) & = & fdp(\{\langle\langle a,b\rangle,c\rangle \mid \langle a,b,c\rangle \in r\}) \\ & = & fdp(\{\langle\langle \pi_1(t),\pi_2(t)\rangle,\pi_3(t)\rangle \mid t \in r\}) \\ & = & \forall t,t' \in r : \pi_1(t) = \pi_1(t') \wedge \pi_2(t) = \pi_2(t') \Rightarrow \pi_3(t) = \pi_3(t') \end{array}$$

and so on.

Exercise 7 Consider the following data model for *priority queues*,

$$\Sigma \cong A \hookrightarrow B^+$$

where B^+ denotes $B^\star - \{<>\}$, as usual, for B an arbitrary data model of the objects to enqueue and A an arbitrary domain of priorities where some total order is assumed.

Suppose that somebody guessed the following relational implementation of Σ,

$$\Sigma' \cong 2^{A \times \mathbb{N} \times B}$$

subject to concrete invariant

$$\phi'(t) \stackrel{\text{def}}{=} \forall r,r' \in t : (\pi_1(r) = \pi_1(r') \Rightarrow \pi_2(r) \neq \pi_2(r'))$$

Show (by applying SETS abstract invariant functorial calculus) that a too strong concrete invariant ϕ' was guessed.
□

[a] It will be seen later that, in fact, many simplifications are likely to happen at algorithm calculation level which are independent of such an extra effort. This happens in particular wherever reasoning steps arise from the functorial or transformational properties of our Set operators, recall what we have said about the "category theoretic" operators of Set and the properties "hidden" in their diagrams.

Recursive Data Model Reification in SETS

Recursion provides "elegant" models for a wide range of problems, *e.g.*:

$$
\begin{aligned}
X &\cong 1 + A \times X && \text{/*finite lists on } A \text{ */} && (179)\\
X &\cong 1 + A \times X^2 && \text{/*binary trees on } A \text{ */} && (180)\\
X &\cong 1 + A \times X^\star && \text{/*generic trees on } A \text{ */} &&\\
X &\cong V + A \times X^\star && \text{/*formal terms on } V \text{ and } A \text{ */} &&
\end{aligned}
$$

etc.

What can we do about refining a recursive data model?

Motivation

Recall from [Fie80] the following VDM *representation* of abstract mappings:

$$
\begin{aligned}
&S1 = [Bt1]\\
&Bt1 :: S1\ Key\ Data\ S1
\end{aligned}
$$

that is, an instance of recursive definition (180) in Set, for $A = Key \times Data$:

$$
\begin{aligned}
X &\cong 1 + X \times (Key \times Data) \times X\\
&\cong 1 + (Key \times Data) \times X^2
\end{aligned}
$$

Fielding's *representation 2* "maps a binary tree onto linear storage":

$$
\begin{aligned}
&S2 :: ROOT : [Ptr]\ ARRAY : Ptr \xrightarrow{m} Node2\\
&Node2 :: [Ptr]\ Key\ Data\ [Ptr]
\end{aligned}
\qquad (181)
$$

that is,

$$
\begin{aligned}
S2 &\cong (Ptr + 1) \times ARRAY\\
ARRAY &\cong Ptr \hookrightarrow Key \times Data \times (Ptr + 1)^2
\end{aligned}
$$

for an abstract domain $Ptr \cong \mathbf{N}$ of "pointers".

Moreover, there is a need for a...

...concrete invariant over $S2$ ensuring: $\begin{cases} \text{pointer definedness !} \\ \text{absence of pointer loops !} \end{cases}$

Question

Can this "recursion removing" reification step be generalized to an arbitrary domain equation

$$X \cong \mathcal{F}(X)?$$

An elegant way of doing it is to regard $\mathcal{F}$ as a polynomial (endo)functor in Set — recall Remark 11.

A Result for Recursion Removal

Let

$$(X_0, \delta : \mathcal{F}(X_0) \to X_0)$$

be a fixpoint solution of

$$X \cong \mathcal{F}(X) \tag{182}$$

in Set.

Reification step:

$$X_0 \trianglelefteq_f^{\phi} K \times (K \hookrightarrow \mathcal{F}(K)) \tag{183}$$

for K a domain of "pointers" such that $K \cong \mathbf{N}$ (see Theorem 3).

Abstraction Function f

We need to define a surjection

$$f : K \times (K \hookrightarrow \mathcal{F}(K)) \longrightarrow X_0$$

total over $\{\langle k, \sigma \rangle \in K \times (K \hookrightarrow \mathcal{F}(K)) \mid \phi(k, \sigma)\}$.

REMARK 24: **On SETS's first recursion removal theorem**

The main theorem which underlies data model recursion removal in SETS is the following [Jou92] [a]:

Theorem 3 (First Recursion Removal Theorem)
Let $\mathcal{F}$ be a Set polynomial endofunctor and let $\langle X_0, \delta \rangle$ be a fixpoint of $\mathcal{F}$ such that X_0 is finite or denumerable, and such that the elements of X_0, regarded as tree-like structures, are finite.
Let K be denumerable and let $\phi : K \times (K \hookrightarrow \mathcal{F}(K)) \to 2$ be a predicate defined by [b]

$$\begin{array}{rcl} \phi(k_0, \sigma) \stackrel{\text{def}}{=} k_0 \in dom(\sigma) & \wedge & (k' <_{\mathcal{F}} k \Rightarrow k' \in dom(\sigma)) \\ & \wedge & (\nexists k \in K : k <^{+}_{\mathcal{F}} k) \\ & \wedge & dom(\sigma) \text{ is finite} \end{array}$$

Finally, let $f : (K \times (K \hookrightarrow \mathcal{F}(K)))_{\phi} \longrightarrow X_0$ be a function defined by

$$f(k_0, \sigma) \stackrel{\text{def}}{=} f_\sigma(k_0) \tag{184}$$

where $f_\sigma : dom(\sigma) \longrightarrow X_0$ is an auxiliary recursive function defined as follows:

$$f_\sigma(k) \stackrel{\text{def}}{=} \delta(\mathcal{F}(f_\sigma)(\sigma(k))) \tag{185}$$

Then

$$X_0 \trianglelefteq^{\phi}_{f} (K \times (K \hookrightarrow \mathcal{F}(K)))$$

holds.
Proof: See [Jou92]. □

Let us see an example of abstraction function f (184) synthesis concerning polynomial functor (179). $(A^\star, \delta)$ is a well-known fixpoint solution of this functor [MA86] for

$$\begin{array}{rcl} \delta & : & 1 + A \times A^\star \longrightarrow A^\star \\ \delta(x) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lcl} <> & \Leftarrow & x = i_1(NIL) \\ cons(a, l) & \Leftarrow & x = i_2(\langle a, l \rangle) \end{array} \right. \end{array} \tag{186}$$

For $\mathcal{F}(f_\sigma)$ we obtain

$$\begin{array}{rcl} \mathcal{F}(f_\sigma)(x) & = & (1 + A \times f_\sigma)(x) \\ & = & (1_1 + 1_A \times f_\sigma)(x) \\ & = & \left\{ \begin{array}{lcl} x & \Leftarrow & x = i_1(NIL) \\ i_2(\langle a, f_\sigma(l) \rangle) & \Leftarrow & x = i_2(\langle a, l \rangle) \end{array} \right. \end{array} \tag{187}$$

Composing (187) with (186) we obtain, following (185),

$$f(k, \sigma) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} <> & \Leftarrow & \sigma(k) = \langle 1, NIL \rangle \\ cons(a, f_\sigma(k')) & \Leftarrow & \sigma(k) = \langle 2, \langle a, k' \rangle \rangle \end{array} \right.$$

which may be written in a more "palatable" notation as follows:

$$\begin{array}{rcll} f(k, \sigma) & \stackrel{\text{def}}{=} & if \text{ is-NIL}(\sigma(k)) \; then <> & (188) \\ & & else \;\; let \;\; \sigma(k) = \langle a, k' \rangle & (189) \\ & & \quad in \;\; cons(a, f(k', \sigma)) & \end{array}$$

[a] Note that a least fixpoint solution to $\mathcal{F}$ (182) always exists wherever $\mathcal{F}$ is *co-continuous*. We will ignore the technical details here, knowing that every polynomial functor is co-continuous — see [MA86], p.262,270.

[b] See (191) and (192) for the definition of $<_{\mathcal{F}}$.

Temporarily assume that $\sigma \in K \hookrightarrow \mathcal{F}(K)$ is a total function, and draw

$$\begin{array}{c} K \\ \downarrow \sigma \\ \mathcal{F}(K) \end{array}$$

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:

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ & & \downarrow \sigma \\ & & \mathcal{F}(K) \end{array}$$

Since (X_0, δ) is a fixpoint of $\mathcal{F}$:

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ \delta \uparrow & & \downarrow \sigma \\ \mathcal{F}(X_0) & & \mathcal{F}(K) \end{array}$$

Finally, since $\mathcal{F}$ is a functor:

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ \delta \uparrow & & \downarrow \sigma \\ \mathcal{F}(X_0) & \xleftarrow{\mathcal{F}(f_\sigma)} & \mathcal{F}(K) \end{array}$$

that is,

$$f_\sigma = \delta \circ \mathcal{F}(f_\sigma) \circ \sigma$$

or

$$\begin{array}{rcl} f & : & K \times (K \hookrightarrow \mathcal{F}(K)) \longrightarrow X_0 \\ f(k, \sigma) & \stackrel{\text{def}}{=} & \delta(\mathcal{F}(f_\sigma)(\sigma(k))) \end{array} \tag{190}$$

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

Concrete Invariant ϕ

Face pointer undefinedness in case σ is partial — $\sigma(k)$ in (190) 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

$$k_1 <_{\mathcal{F}} k \stackrel{\text{def}}{=} k \in dom(\sigma) \wedge k_1 \in_{\mathcal{F}} \sigma(k) \tag{191}$$

where $\in_{\mathcal{F}}$ is structurally defined for polynomial $\mathcal{F}$:

$$\begin{array}{rcl} k \in_C x & \stackrel{\text{def}}{=} & FALSE \\ k \in_{\lambda X.X} x & \stackrel{\text{def}}{=} & k = x \\ k \in_{\mathcal{F}\times\mathcal{G}} \langle x, y\rangle & \stackrel{\text{def}}{=} & k \in_{\mathcal{F}} x \vee k \in_{\mathcal{G}} y \\ k \in_{\mathcal{F}+\mathcal{G}} x & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lcl} k \in_{\mathcal{F}} y & \Leftarrow & x = i_1(y) \\ k \in_{\mathcal{G}} z & \Leftarrow & x = i_2(z) \end{array} \right. \end{array} \tag{192}$$

that is

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

- Well-foundedness — Restrict to least fixpoints [MA86], *i.e.* guarantee that $f(k,\sigma)$ does not yield infinite results, forcing P to be well-founded *w.r.t.* $<_{\mathcal{F}}$:

$$\begin{array}{rcll} \phi(k,\sigma) & \stackrel{\text{def}}{=} & let & P = \{k\} \cup \{k' \in K \mid k' <^+_{\mathcal{F}} k\} \\ & & in & P \subseteq dom(\sigma) \wedge \\ & & & \forall \emptyset \subset C \subseteq P : \exists m \in C : \forall k' <_{\mathcal{F}} m : k' \notin C \end{array} \tag{193}$$

The proof of surjectiveness of f (190) for $(X_0, \delta : \mathcal{F}(X_0) \to X_0)$ a finite or denumerable fixpoint of $\mathcal{F}$, is a lengthy one! [Jou92]

Trivial Example

$(C, 1_C)$ is of course a fixpoint of constant functor $\mathcal{F}(X) = C$. Then

$$\begin{array}{rcl} f(k,\sigma) & \stackrel{\text{def}}{=} & \delta(\mathcal{F}(f_\sigma)(\sigma(k))) \\ & = & 1_C(C(f_\sigma)(\sigma(k))) \\ & = & 1_C(1_C(\sigma(k))) \\ & = & \sigma(k) \end{array}$$

Since $<^+_{\mathcal{F}}$ is empty, we have

$$\phi(k,\sigma) \stackrel{\text{def}}{=} k \in dom(\sigma) \tag{194}$$

as expected — *cf.* dynamic programming typical of *e.g.* C or PASCAL— instead of handling C data directly (statically stored), the program handles *dynamic* references to them.

REMARK 25: **More about the recursion removal concrete invariant**

Equation (193) is a generalization of some invariant definitions found in the VDM literature, *cf. e.g.* [Fie80, Wol88, Wol90]. It is less constrained, however, in the sense that only the set P of pointers in σ reachable from k is affected by non-circularity and definedness restrictions ([Fie80, Wol90] constrain $dom(\sigma)$ as a whole).
Some algorithmic flavour can be added to (193) by introducing an auxiliary function

$$reach(k,\sigma) \stackrel{\text{def}}{=} \{k\} \cup \{k' \in K \mid k' <^{+}_{\mathcal{F}} k\}$$

(that is, $P = reach(k,\sigma)$) subject to transformations as follows:

$$\begin{array}{lll} reach(k,\sigma) \stackrel{\text{def}}{=} & & \\ \quad \{k\} \cup \{k' \in K \mid k' <^{+}_{\mathcal{F}} k\} & & \\ = \quad \{k\} \cup \left\{ \begin{array}{ll} \emptyset & \Leftarrow \quad k \notin dom(\sigma) \\ \bigcup_{k' <_{\mathcal{F}} k} \{k'\} \cup \{k'' \in K \mid k'' <^{+}_{\mathcal{F}} k'\} & \Leftarrow \quad k \in dom(\sigma) \end{array} \right. & \\ = \quad \{k\} \cup \left\{ \begin{array}{ll} \emptyset & \Leftarrow \quad k \notin dom(\sigma) \\ \bigcup_{k' <_{\mathcal{F}} k} reach(k',\sigma) & \Leftarrow \quad k \in dom(\sigma) \end{array} \right. & \end{array}$$

The test for definedness may be encapsulated in a predicate

$$defined(k,\sigma) \stackrel{\text{def}}{=} P \subseteq dom(\sigma)$$

Since $P = reach(k,\sigma)$ we obtain, after the expected substitutions and transformations:

$$defined(k,\sigma) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} FALSE & \Leftarrow \quad k \notin dom(\sigma) \\ \forall k' <_{\mathcal{F}} k : defined(k',\sigma) & \Leftarrow \quad k \in dom(\sigma) \end{array} \right.$$

Note that both $reach$ and $defined$ do not compute the intended set-theoretical fixpoint implicit in the transitive closure of $<_{\mathcal{F}}$ wherever this relation is cyclic [a].
A $<_{\mathcal{F}}$-cycle is detected wherever we revisit the same $k \in K$. Since cycle detection matches with wellfoundedness testing, one may put everything together and write

$$\phi(k,\sigma) = \phi_{aux}(k,\sigma,\emptyset) \tag{195}$$

where

$$\begin{array}{l} \phi_{aux}(k,\sigma,C) \stackrel{\text{def}}{=} \\ \quad \left\{ \begin{array}{ll} FALSE & \Leftarrow \quad k \notin dom(\sigma) \vee k \in C \\ \forall k' <_{\mathcal{F}} k : \phi_{aux}(k',\sigma,C \cup \{k\}) & \Leftarrow \quad k \in dom(\sigma) \end{array} \right. \end{array} \tag{196}$$

Exercise 8 Show that, for $\mathcal{F}(X) = C \hookrightarrow \mathcal{G}(X)$, $\in_{\mathcal{F}}$ expands to:

$$k \in_{C \hookrightarrow \mathcal{G}} x \stackrel{\text{def}}{=} \exists c \in dom(x) : k \in_{\mathcal{G}} x(c) \tag{197}$$

□

[a] Note in passing that, similarly to (120), logical conjunction in (191) should once again regarded as non-strict.

REMARK 26: **Particular cases of recursive data model reification**

Further to the constant functor $\mathcal{F}(X) = C$ which lead us to (194), other particular cases are of interest concerning (183):

– Constant functor for $C = 0$. This case trivializes (182) to the definition $X \cong 0$ of an empty data domain; note that $K \times (K \hookrightarrow \mathcal{F}(K))$ becomes $K \times (K \hookrightarrow 0)$ which reduces to

$$\begin{aligned} K \times (K \hookrightarrow 0) &\cong K \times \bigcup_{X \subseteq K} 0^X \\ &\cong K \times 0^0 \end{aligned}$$

That is, $\sigma : 0 \to 0$ is bound to be completely undefined, and $dom(\sigma) = \emptyset$; then $k \in dom(\sigma) = FALSE$ and (194) reduces to

$$\phi(k, \sigma) \stackrel{\text{def}}{=} FALSE$$

As expected, we are led to the empty reification (every concrete-level datum is invalid).

– Identity functor $\mathcal{F}(X) = X$. This case trivializes (182) to $X \cong X$, which accepts any fixpoint $(X_0, \delta : X_0 \to X_0)$ for δ a bijection. Let us see how the least fixpoint interpretation implicit in (193) reduces this case to the previous one ($X_0 = 0$), by showing that definedness and well-foundedness cannot be met at the same time.

Assuming that P is well-founded *w.r.t.* $k_1 <_{\lambda X.X} k$ (which we abbreviate to $k_1 < k$ and is logically equivalent to $k \in dom(\sigma) \wedge k_1 = \sigma(k)$), we have, for $C = P$ in (193):

$$\exists m \in P : \sigma(m) \notin P \tag{198}$$

But if $m \in P$ then m is reachable from k ($m <^+ k$), $m \in dom(\sigma)$ and $\sigma(m) < m$. Then

$$\sigma(m) < m <^+ k$$

i.e. $\sigma(m) <^+ k$, which entails $\sigma(m) \in P$ and contradicts (198). Thus the conjunction of definedness and well-foundedness in (194) is impossible, and we obtain

$$\phi(k, \sigma) \stackrel{\text{def}}{=} FALSE$$

— something we would expect as refinement of the least fixpoint $X_0 = 0$ of the identity functor.

Exercise 9 The following recursive data model

$$BT \cong 1 + BT \times (A \times BT)^\star$$

which may be accepted as a simplistic formalization for B^+-trees, involves functor

$$\mathcal{F}(X) = 1 + X \times (A \times X)^\star$$

Find the corresponding $<_{\mathcal{F}}$ ordering. **Hint**: do it first for $\mathcal{F}(X) = X^\star$.
□

An Illustration of Calculated Reification

Recall

$$DecTree \cong What \times (Answer \xrightarrow{} DecTree)$$

Data-level Reification

First transformation is recursion removal according to (183)

$$DecTree \trianglelefteq_1 DecTree_1$$

leading to

$$DecTree_1 = K \times (K \xrightarrow{} What \times (Answer \xrightarrow{} K)) \quad (199)$$

What have we achieved? Two interpretations (at least) are admissible:

1. For K a domain of *pointers*,

   $$K \xrightarrow{} What \times (Answer \xrightarrow{} K) \quad (200)$$

   — that is:

   $$K \xrightarrow{} What \times (K + 1)^{Answer}$$

   — models a heap-segment of dynamic storage. In PASCAL (the heap is "hidden" in the run-time system):

   ```
   type DecTree1 = ^DecTree;
        DecTree =  record
                   Q: What;
                   R: array [Answer] of ^DecTree
                   end;
   ```

2. For K a domain of *object names* or "unique identifiers", (200) models the object database

   $$name \rightarrow object$$

   implicit in an object-oriented programming environment, *cf.* the following VDM "sugaring" of (199):

   $$\begin{array}{lll} DecTree_1 :: & ObjName : & K \\ & Archive : & ObjBase \end{array}$$

$$
\begin{array}{rcl}
ObjBase & = & K \xrightarrow{m} Attributes \\
Attributes & :: & Q : \ What \\
 & & SubObjs : \ Answer \xrightarrow{m} K
\end{array}
$$

Proceeding via law (129):

$$
\begin{array}{rcl}
DecTree_1 & = & K \times (K \hookrightarrow What \times (Answer \hookrightarrow K)) \\
 & \trianglelefteq_2 & K \times ((K \hookrightarrow What) \times ((K \times Answer) \hookrightarrow K)) \\
 & = & DecTree_2
\end{array}
\tag{201}
$$

For K a state descriptor, $DecTree_2$ accepts the following interpretation:

- $(K \times Answer) \hookrightarrow K$ = *state transition diagram* of a (deterministic) *finite state automaton* ($Answer$ = input stimuli);
- first factor $k \in K$ in (201) = current state of the automaton;
- $K \hookrightarrow What$ = semantic table assigning a meaning to each state.

Carrying on our reasoning,

$$
\begin{array}{rcl}
DecTree_2 & = & K \times ((K \hookrightarrow What) \times ((K \times Answer) \hookrightarrow K)) \\
 & \trianglelefteq_3 & K \times (2^{K \times What} \times 2^{(K \times Answer) \times K}) \\
 & = & DecTree_3 \\
 & \cong_4 & K \times (2^{K \times What} \times 2^{K \times Answer \times K}) \\
 & = & DecTree_4
\end{array}
$$

we obtain a final model,

$$
DecTree_4 = K \times (2^{K \times What} \times 2^{K \times Answer \times K}) \tag{202}
$$

which is a relational database schema implementing $DecTree$ in terms of two database files where K plays the rôle of a domain of *keys*.

In summary,

$$
DecTree \trianglelefteq_1 DecTree_1 \trianglelefteq_2 DecTree_2 \trianglelefteq_3 DecTree_3 \cong_4 DecTree_4
$$

Abstraction Invariant Synthesis

(In brief: see remarks 27 and 28 for details.)
The synthesis of the overall abstraction map via (121), *i.e.*

$$f(k, \langle t, t' \rangle) = f_1(f_2(f_3(f_4(k, \langle t, t' \rangle))))$$

will lead to

$$\begin{array}{rcll} f(k, \langle t, t' \rangle) & \stackrel{\text{def}}{=} & let & w = the(\{\pi_2(r) \mid r \in t \wedge \pi_1(r) = k\}) \\ & & in & \langle w, \left(\begin{array}{c} a \\ f(k', \langle t, t' \rangle) \end{array} \right)_{\langle k'', a, k' \rangle \in t' \wedge k'' = k} \rangle \end{array} \quad (203)$$

The overall concrete invariant can be calculated in a similar way,

$$\begin{array}{rcl} \phi(k, \langle t, t' \rangle) & \stackrel{\text{def}}{=} & \phi_3(f_4(k, \langle t, t' \rangle)) \wedge \phi_2(f_3(f_4(k, \langle t, t' \rangle))) \wedge \\ & & \phi_1(f_2(f_3(f_4(k, \langle t, t' \rangle)))) \end{array} \quad (204)$$

leading to

$$\begin{array}{l} \phi(k, \langle t, t' \rangle) = \\ \quad (\forall \langle k, w \rangle, \langle k', w' \rangle \in t : k = k' \Rightarrow w = w') \wedge \\ \quad (\forall \langle k_1, a, k_1' \rangle, \langle k_2, b, k_2' \rangle \in t' : (k_1 = k_2 \wedge a = b) \Rightarrow k_1' = k_2') \wedge \\ \quad 2^{\pi_1}(t') \subseteq 2^{\pi_1}(t) \wedge \\ \quad aux(k, t, t', \emptyset) \end{array}$$

where

$$aux(k, t, t', C) \stackrel{\text{def}}{=}$$
$$\left\{ \begin{array}{lcl} FALSE & \Leftarrow & k \notin 2^{\pi_1}(t) \vee \\ & & k \in C \\ \forall r \in \{\pi_1(s) = k \mid s \in t'\} : aux(\pi_3(r), t, t', C \cup \{k\}) & \Leftarrow & k \in 2^{\pi_1}(t) \end{array} \right.$$

Another Example of Recursion Removal

Recall (6),

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

in which we find the "pedigree functor":

$$\mathcal{F}(X) \;=\; I \times (X + 1)^2$$

— see also (115).

REMARK 27: **Calculation details for (203)**

Overall abstraction function f is obtained by functorial composition of abstraction maps f_1 to f_4, as follows: f_4 is given by

$$f_4 = 1_K \times (1_{2^{K \times What}} \times 2^j)$$

where j is isomorphism

$$g(\langle a, b, c\rangle) \quad = \quad \langle\langle a, b\rangle, c\rangle$$

— recall (178) — concerning

$$(A \times B) \times C \cong_j A \times B \times C$$

That is to say,

$$f_4(\langle k, \langle t, t'\rangle\rangle) \quad = \quad \langle k, \langle t, \{\langle\langle k, a\rangle, k'\rangle \mid \langle k, a, k'\rangle \in t'\}\rangle\rangle$$

Similarly, f_3 is based on mkf (116), which abstracts mappings from relations (wherever possible),

$$f_3 \quad = \quad 1_K \times (mkf \times mkf)$$

while f_2 simply is

$$f_2 = 1_K \times \bowtie$$

recall (144). Finally,

$$\begin{array}{rcll} f_1(k, \sigma) & = & f_\sigma(k) & \\ f_\sigma(k) & \stackrel{\text{def}}{=} & let & \sigma(k) = \langle w, \sigma'\rangle \\ & & in & \langle w, \left(\begin{array}{c} a \\ f_\sigma(\sigma'(a)) \end{array} \right)_{a \in dom(\sigma')} \rangle \end{array} \tag{205}$$

cf. (190) and (9), and the $C \hookrightarrow f$ entry in the table of page p.44. Then

$$\begin{array}{rcl} f_3(f_4(\langle k, \langle t, t'\rangle\rangle)) & = & \langle k, \langle mkf(t), mkf(\{\langle\langle k, a\rangle, k'\rangle \mid \langle k, a, k'\rangle \in t'\})\rangle\rangle \\ & = & \langle k, \langle \left(\begin{array}{c} k \\ w \end{array} \right)_{\langle k, w\rangle \in t}, \left(\begin{array}{c} \langle k, a\rangle \\ k' \end{array} \right)_{\langle k, a, k'\rangle \in t'} \rangle\rangle \end{array}$$

and

$$\begin{array}{l} f_2(f_3(f_4(\langle k, \langle t, t'\rangle\rangle))) = \\ \quad \langle k, \left(\begin{array}{c} \\ \langle w, \left(\begin{array}{c} a \\ k' \end{array} \right)_{\langle k'', a, k'\rangle \in t' \wedge k'' = x} \rangle \end{array} \right)^{x}_{\langle x, w\rangle \in t} \rangle \end{array} \tag{206}$$

Composing (206) with (205) one obtains, after some simplifications,

$$\begin{array}{rcll} f(k, \langle t, t'\rangle) & \stackrel{\text{def}}{=} & let & w = the(\{\pi_2(r) \mid r \in t \wedge \pi_1(r) = k\}) \\ & & in & \langle w, \left(\begin{array}{c} a \\ f(k', \langle t, t'\rangle) \end{array} \right)_{\langle k'', a, k'\rangle \in t' \wedge k'' = k} \rangle \end{array}$$

that is, (203).

REMARK 28: **Calculation details for concrete invariant (204)**

Let us now address the calculation of (204), which should stem from:

$$\phi(k, \langle t, t' \rangle) = \phi_3(f_4(k, \langle t, t' \rangle)) \wedge \phi_2(f_3(f_4(k, \langle t, t' \rangle))) \wedge \phi_1(f_2(f_3(f_4(k, \langle t, t' \rangle))))$$

since ϕ_4 is the everywhere $TRUE$ predicate. ϕ_3 involves fdp (160),

$$\phi_3 = (\lambda k.T) \times fdp \times fdp$$

and ϕ_2 is nothing but dpi (157). It can be checked that $\phi_2(f_3(f_4(k, \langle t, t' \rangle)))$ reduces to $2^{\pi_1}(t') \subseteq 2^{\pi_1}(t)$. Then

$$\begin{array}{rcl} \phi_3(f_4(k, \langle t, t' \rangle)) & = & TRUE \wedge fdp(t) \wedge fdp(\{\langle \langle k, a \rangle, k' \rangle \mid \langle k, a, k' \rangle \in t'\}) \\ & = & fdp(t) \wedge fdp(\{\langle \langle k, a \rangle, k' \rangle \mid \langle k, a, k' \rangle \in t'\}) \end{array} \quad (207)$$

By calculating $fdp(\{\langle \langle k, a \rangle, k' \rangle \mid \langle k, a, k' \rangle \in t'\})$ further in (207) we will have

$$\forall \langle k_1, a, k_1' \rangle, \langle k_2, b, k_2' \rangle \in t' : (k_1 = k_2 \wedge a = b) \Rightarrow k_1' = k_2'$$

Finally, ϕ_1 (193) is based upon the ordering

$$\begin{array}{rcl} k_1 < k & \stackrel{\text{def}}{=} & k \in dom(\sigma) \wedge \\ & & (FALSE \vee \exists a \in dom(\pi_2(\sigma(k))) : k_1 = \pi_2(\sigma(k))(a)) \\ & \Leftrightarrow & k \in dom(\sigma) \wedge \ let \quad \sigma' = \pi_2(\sigma(k)) \\ & & \qquad\qquad\quad in \quad \exists a \in dom(\sigma') : k_1 = \sigma'(a) \end{array}$$

cf. (197). Recalling (195) and (196) above,

$$\phi_1(f_2(f_3(f_4(k, \langle t, t' \rangle))))$$

will reduce to $aux(k, t, t', \emptyset)$ where

$$aux(k, t, t', C) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} FALSE & \Leftarrow & k \notin 2^{\pi_1}(t) \vee \\ & & k \in C \\ \forall r \in \{\pi_1(s) = k \mid s \in t'\} : aux(\pi_3(r), t, t', C \cup \{k\}) & \Leftarrow & k \in 2^{\pi_1}(t) \end{array} \right. \quad (208)$$

Putting everything together, we finally obtain the following (non-trivial) concrete invariant over (202):

$$\begin{array}{l} \phi(k, \langle t, t' \rangle) = \\ \quad (\forall \langle k, w \rangle, \langle k', w' \rangle \in t : k = k' \Rightarrow w = w') \wedge \\ \quad (\forall \langle k_1, a, k_1' \rangle, \langle k_2, b, k_2' \rangle \in t' : (k_1 = k_2 \wedge a = b) \Rightarrow k_1' = k_2') \wedge \\ \quad 2^{\pi_1}(t') \subseteq 2^{\pi_1}(t) \wedge \\ \quad aux(k, t, t', \emptyset) \end{array} \quad (209)$$

where aux is the recursive auxiliary predicate given by (208).
The reader who finds (209) a too complex concrete invariant should note the following:

- it is the "weakest" invariant which guarantees, according the adopted calculation path, the correct relational implementation of $DecTree$.
- it provides some evidence about how "unsafe" and risky the manipulation of such an implementation is likely to become if performed in an informal, careless way.

How do we build a relational database recording such family pedigrees?

By calculation! First, apply our main result for recursion removal, obtaining

$$K \times (K \hookrightarrow I \times (K+1)^2)$$

Next, Newton's *binomial* formula (34) will be of some help,

$$\begin{array}{lcl} K \hookrightarrow I \times (K+1)^2 & \cong & K \hookrightarrow I \times (K^2 + 2 \times K + 1) \\ & \cong & K \hookrightarrow (I \times K^2 + I \times 2 \times K + I) \\ & \trianglelefteq & (K \hookrightarrow I \times K \times K) \times \\ & & (K \hookrightarrow I \times 2 \times K) \times \\ & & (K \hookrightarrow I) \end{array}$$

— a reasoning chain in which law (130) also plays an important rôle.

The relational structure we have obtained by calculation confirms our intuition about the pedigree view of family relationship:

> *"About an individual either both father and mother are known ($I \times K^2$), or one of either father or mother is known ($I \times 2 \times K$), or both father and mother are unknown (I)"*

See Remark 29 for some more results available in SETS for recursion removal.

Handling 'In loco' Data-type Invariants in SETS

Our main purpose in this section is to describe how the SETS method deals with so-called *local* ("in loco") data type invariants.

REMARK 29: **On SETS's second recursion removal theorem**

Some languages support recursion directly (*e.g.* LISP) but may others do not (*e.g.* FORTRAN). Languages such as C or PASCAL are "half-way through" — recursion is supported in an indirect way by providing *pointers* to *dynamic storage* ("heaps"). Programming with pointers is error-prone (*cf.* pointer undefinedness, non-termination caused by cyclic referencing *etc.*) and tends to produce "tricky" code (pointers are data-structuring counterparts of "gotos" [Wir76]). A way of overcoming this situation is, as we have seen, to think of generic rules for refining recursive data structures into safe non-recursive ones, thus ruling out non-systematic use of pointers [a].

So far, we have regarded K (183) as an arbitrary, denumerable set. We said that some $k \in K$ might be regarded as a name, a key or a pointer, depending upon our implementation target. But there is some difference between a pointer as understood in languages such as C or PASCAL and, for instance, a name or key in a relational database system: pointers can have value NIL. So, a pointer data type is better captured by $1 + K$ (or $K + 1$, obviously), where $1 \cong \{NIL\}$ and K is a set big enough to contain all memory cell addresses of a computer (in theory, $K \cong \mathbb{N}$) — recall the analogy in the table of p.12.

Pointers have particularly to do with recursive patterns of the shape $\mathcal{F}(X) = 1 + \mathcal{G}(X)$. Although such patterns fall well into Theorem 3, there is an alternative result which matches better conventional pointer-oriented computer programming techniques. We present it as a theorem but, for space economy, we ignore a few details of the presentation and omit its proof. A complete presentation of this result can be found in [Jou92].

Theorem 4 (Second Recursion Removal Theorem)

Let $\mathcal{G}$ be a polynomial functor in Set and let $(X_0, \mathcal{F}(X_0) \xrightarrow{\delta} X_0)$ be a fixpoint solution of equation

$$X \cong 1 + \mathcal{G}(X)$$

such that X_0 is finite or denumerable, and such that the elements of X_0, regarded as tree-like structures, are finite. Let K be denumerable. Then the following reification holds,

$$X_0 \trianglelefteq_f^{\phi} ((1 + K) \times (K \hookrightarrow \mathcal{G}(1 + K)))$$

where f is given by

$$f(k, \sigma) \stackrel{\text{def}}{=} f_\sigma(k)$$

and where

$$\begin{array}{rcl} f_\sigma & \stackrel{\text{def}}{=} & \delta \circ (1 + \mathcal{G}(f_\sigma)) \circ (1 + \sigma) \\ & & \delta \circ (1 + \mathcal{G}(f_\sigma) \circ \sigma) \end{array}$$

both partial over concrete invariant

$$\begin{array}{l} \phi(\langle i, x\rangle, \sigma) \stackrel{\text{def}}{=} \\ \quad ((x \neq NIL \Rightarrow x \in dom(\sigma)) \quad \wedge \quad (k' <_\sigma k \Rightarrow k' \in dom(\sigma)) \\ \qquad\qquad\qquad\qquad\quad \wedge \quad (\nexists k \in K : k <_\sigma^+ k) \\ \qquad\qquad\qquad\qquad\quad \wedge \quad dom(\sigma) \text{ is finite}) \end{array}$$

Proof: See [Jou92]. □

Exercise 10 Recalculate (189) by applying Theorem 4 instead of Theorem 3 to polynomial functor (179). **Hint**: have a look at Remark 36.

□

[a] Much in the same way that one can do "Structured Programming with Goto Statements" *cf.* [Knu74].

REMARK 30: **Recursive data structures and their refinement: epilogue**

The results we have seen so far enable us to refine recursive structures into non-recursive ones. The implementation of recursive structures by other recursive structures is also of practical interest. For instance, binary search-trees (180) intuitively implement finite lists (179) — just stick to totally unbalanced trees or simply ignore the left (or right) branch of the tree (consistently). The latter situation is supported by the following theorem, which extends $\trianglelefteq$-monotonicity towards recursively defined data domains.

Theorem 5 ($\trianglelefteq$-Monotonicity of Set-Recursion) *Let $\mathcal{F}$ and $\mathcal{G}$ be two Set-functors, built by composition of the $\trianglelefteq$-monotone operators of Theorem 2. If*

$$\mathcal{F}(X) \trianglelefteq \mathcal{G}(X) \tag{210}$$

for any X, then a solution to domain equation

$$X \cong \mathcal{G}(X) \tag{211}$$

is a $\trianglelefteq$-refinement of the least solution to

$$X \cong \mathcal{F}(X) \tag{212}$$

Proof: We will prove that any fixpoint solution $X_{\mathcal{G}}$ to equation (211) is a $\trianglelefteq$-refinement of $\mu\mathcal{F}$, the least fixpoint solution to equation (212). Firstly, if $X_{\mathcal{G}}$ is a solution of $\mathcal{G}$, then $\mathcal{G}(X_{\mathcal{G}}) \cong X_{\mathcal{G}}$, *i.e.*, $\mathcal{G}(X_{\mathcal{G}}) \trianglelefteq X_{\mathcal{G}}$ simply because isomorphism is a special case of refinement. Then

$$\mathcal{F}(X_{\mathcal{G}}) \trianglelefteq X_{\mathcal{G}} \tag{213}$$

by equation (210) and $\trianglelefteq$-transitivity. Since $\mathcal{F}$ involves only monotone operators, $\mathcal{F}$ is also monotone [Man74]. Then we may regard equation (213) as the antecedent of a fixpoint induction argument [Man74], whose consequent is

$$\mu\mathcal{F} \trianglelefteq X_{\mathcal{G}}$$

and this completes the proof. □

Let $\mathcal{F}(X) = 1 + A \times X$ and $\mathcal{G}(X) = 1 + A \times X^2$, for instance. Then $\mathcal{F}(X) \trianglelefteq \mathcal{G}(X)$ holds — under abstraction map $1 + A \times \pi_1$, for instance, a functorial consequence of (147) — and the above theorem will confirm that, as a matter of fact, $A^\star$ can be implemented by any solution to the binary tree equation (*e.g.* the set of all 2-branching S-expressions).

We finish this remark about recursive structures and their refinement with two relevant comments:

- We don't consider infinite solutions to recursive equations. This is apparent in theorems 3 and 4, where constraint "$dom(\sigma)$ is finite" is explicitly quoted in the concrete invariant (note that $K \cong \mathbb{N}$). This also happens implicitly in other situations where $\mathbb{N}$ is allowed to occur in the exponent of an exponentiation. In (117), for instance, a non-finite B (allowed in $A \hookrightarrow B$ provided A is finite) jumps to the exponent of $2^{A \times B}$ on the righthand side. But fdp (160), for a finite A, restores finiteness. In law (133) finiteness is guaranteed by a concrete invariant imposing injectivity (otherwise indexing becomes ambiguous). In law (134) the concrete invariant imposes a finite domain on the righthand side, otherwise an infinite list is retrieved.

- In general, $X \cong \mathcal{F}(X)$ does not always have solutions in $Sets$ because exponentiation is allowed. A well known counter-example, due to Scott & Strachey, is $X \cong A + X^X$. [MA86] give a thorough discussion of this problem, which leads beyond sets to *domains*. However, functors $\mathcal{F}(X)$ arbitrarily involving X in the exponent are unusual in data-type specification, and are of theoretical interest only. As pointed out by [MA86], one may stay with $Sets$ in data-type definition, resorting to domains only in program (algorithmic) specification.

Exercise 11 For A a nonempty set, prove the following trivial refinement law

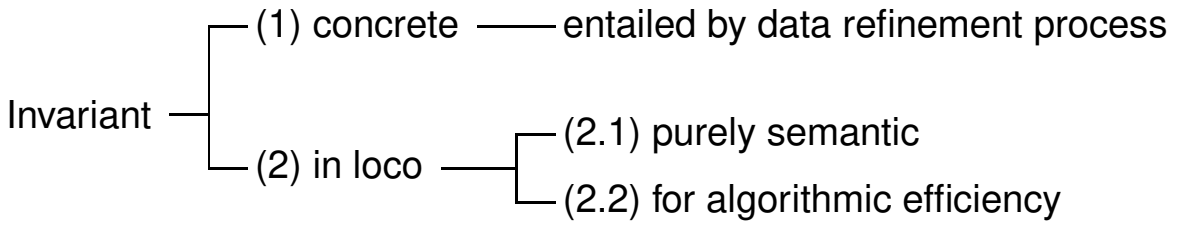
$$1 \quad \trianglelefteq \quad A \tag{214}$$

and use this result and Theorem 5 to show that $\mathbb{N}$, the least solution to $X \cong 1 + X$, is refined by $A^\star$, the least solution to (179). What is the intuition behind this (highly redundant) reification? Given a particular sequence of As, what is the natural number it represents?

□

Local invariants are constraints which, although not implied by the reification process, may be enforced into an implementation to obtain extra advantages — typically, efficiency.

A Data-type Invariant Taxonomy



COMMENTS:

- *concrete invariants are "minimal" constraints entailed by the data refinement process* — the SETS calculus ensures these as weak as necessary in order to guarantee faithful data representation;
- *purely semantic local invariants* — arbitrarily stated at the highest specification level, these invariants mirror real-life conventions, rules, pieces of legislation, norms or natural constraints;
- *local invariants for algorithmic efficiency* — arbitrarily stated at intermediate or final reification levels, these invariants exploit the target machine architecture for efficiency.

In the refinement literature, the term "concrete invariant" normally refers to a combination of (1) and (2.2) in the taxonomy above — see [Mor90], for instance.

Examples

Case (2.1)

A well-known example of a purely semantic local invariant concerns data-type

$$Date \cong 31 \times 12 \times \mathbb{N}$$

which requires (in the Gregorian calendar) a nontrivial 'in loco' invariant:

$$\begin{array}{l} dateOk : Date \longrightarrow 2 \\ dateOk(d,m,y) \stackrel{\text{def}}{=} \\ \quad \left\{ \begin{array}{lcl} m \in \{1,3,5,7,8,10,12\} & \Rightarrow & d \leq 31 \wedge \\ & & (\neg(y = 1582 \wedge m = 10) \\ & & \vee(d < 5) \vee (14 < d)) \\ m \in \{4,6,9,11\} & \Rightarrow & d \leq 30 \\ m = 2 \wedge leapYear(y) & \Rightarrow & d \leq 29 \\ m = 2 \wedge \neg leapYear(y) & \Rightarrow & d \leq 28 \end{array} \right. \end{array} \tag{215}$$

where

$$\begin{array}{rcl} leapYear & : & \mathbb{N} \longrightarrow 2 \\ leapYear(y) & \stackrel{\text{def}}{=} & rem(y, \left\{ \begin{array}{lcl} 1700 \leq y \wedge rem(y,100) = 0 & \Rightarrow & 400 \\ 1700 > y \vee rem(y,100) \neq 0 & \Rightarrow & 4 \end{array} \right.) = 0 \end{array}$$

Case (2.2)

A well-known example of a local invariant for algorithmic efficiency is, in the context of

$$2^A \trianglelefteq_{elems} A^\star$$

to restrict $A^\star$ to sequences with no duplicates and linearly ordered by some ordering on A.

Question

How do we handle local invariants in SETS?

Case (2.1)

Given $\alpha : A \longrightarrow 2$, let A_α denote the set

$$\{a \in A \mid \alpha(a)\}$$

that is, the result of imposing local invariant α upon A, for example,

$$Date_{dateOk}$$

REMARK 31: **About 'in loco' invariants**

For programmers and program users it is important to know not only which properties hold for the programs they deal with, but also *why* and *when* they arise in the development process.

Particularly relevant properties of programs are those which hold forever along programs lifetime — these are called *invariant properties*. Other such "eternal" properties may hold as consequences of the underlying invariants.

Data-type invariants are a burden for the software developer because formal arguments have to be produced ensuring that they are preserved by all the operations which create the corresponding data. So, in terms of design effort, it is important to have as weak overall data type invariants as possible.

Traditional engineering pragmatics dealing with continuous domains (*e.g.* space, time, temperature, voltage) normally impose so-called "safety overheads" (*e.g.* "10% stronger than calculated") in order to guarantee design implementation, in face of instrumental inaccuracy, for instance. The same "culture" applied to informal software design tends to overstrengthen presumed properties (invariants) of software artifacts. But a contrast can be found here: discreteness of data (supported by inductive arguments) should make software (theoretically) 100% accurate!

As we have seen, the process of data refinement normally entails the introduction of data-type invariants required for the implementation to be faithful to the specification. As described in the main text, the SETS calculus is based on a so-called *redundancy ordering* which guarantees data representation whilst saving as many data-type invariants as technically possible.

This strategy has the advantage of preventing from implementations "stronger" than necessary, but ignores other aspects of data refinement which are relevant in practice, namely,

- the proper starting specification of a design normally involves data-types affected by 'in loco' invariants which formalize real-life conventions, laws, rules, norms or natural constraints. $dateOk$ (215) is a good illustration of the *ad hoc* character of most constraints formally imposed to specifications. It arises from a natural, cosmological fact — the number of days of the year is not a natural number — and from several historical attempts to devise a norm able to finitely approximate such a number [a].

- some rules of SETS are "context-sensitive" in the sense that they can be applied to a data-type only if this is affected by some invariant [Oli90];

- 'in loco' invariants may be introduced throughout refinement which have nothing to do with representation faithfulness (*representativeness*), but rather with other facets of refinement such as *e.g.* seeking *efficiency*.

  For instance, a basic reification fact in SETS is (132), for A a finite set, meaning that finite lists implement finite sets. However, for efficiency we may be interested in implementing 2^A by lists in $A^\star$ which are ordered by some total order on A, which is irrelevant *w.r.t.* the abstraction function $elems$ (159) but reduces the time-complexity of searching for some element in such lists.

In practice, both the need to bring *legislation* into many computer applications and the overall interest in efficient applications justify our study of local invariants and their impact in the SETS methodology.

Exercise 12 Concerning the PPD data model specification of p.52, specify as local invariants the following desirable properties of the toy-factory database:

- no equipment is made of unknown (sub)equipments;
- equipments can be stocked only if their production tree is known;
- if a component is involved in the production of at least one equipment, than its price is known;
- no equipment can recursively be a sub-block of itself.

□

[a] The situation is actually more complex, because this invariant is "geography" dependent. In fact, the 1582 discontinuity (10 days omitted in October of that year) of the Gregorian calendar (215) was adopted much later in several countries — *e.g.* Poland (1586), Hungary (1587), Germany (1700), England (1752) *etc.* [Gof84]. (A world-wide "date" data-type taking into account current and past dating systems, *e.g.* Chinese, Indian, *etc.*, would be overwhelming in complexity.)

What is the *least* impact of α on a given refinement of A?

Theorem 6 *Should $A \trianglelefteq_f^{\phi} B$ hold and $\alpha : A \longrightarrow 2$ be an arbitrary invariant on A, then α can be "pushed" downto B as follows:*

$$A_{\alpha} \trianglelefteq_f^{\phi} B_{\phi \wedge \alpha \circ f} \tag{216}$$

where $\phi \wedge \alpha \circ f$ is a shorthand for $\lambda b.\phi(b) \wedge \alpha(f(b))$.
□

Case (2.2)

Let

$$A \trianglelefteq_f^{\phi} B \tag{217}$$

hold, and $\beta : B \longrightarrow 2$ be an extra invariant we would like to impose at concrete level B.

Should the β-restriction of f,

$$\begin{array}{rcl} f|\beta & : & B_{\phi \wedge \beta} \longrightarrow A \\ b & \rightsquigarrow & f(b) \end{array}$$

be surjective, then "stronger" refinement

$$A \trianglelefteq_{f|\beta}^{\phi \wedge \beta} B \tag{218}$$

will be valid and can replace (217) (see Theorem 7).

Predicate β can be as strong as we wish provided that the cardinality of $B_{\phi \wedge \beta}$ is greater or equal to the cardinality of A.

LIMIT CASE:

$$A \cong B_{\phi \wedge \beta} \tag{219}$$

for instance:

$$2^A \quad \cong_{elems|linear} \quad (A^{\star})_{linear} \tag{220}$$

where $linear$ forces every sequence $s \in A^{\star}$ to be strictly ordered by a given linear order in A.

In summary, we have seen two results for local invariant handling, one in which the local invariant is introduced on the lefthand side of a $\trianglelefteq$-law (Theorem 6) and another in which it is introduced on the righthand side (Theorem 7).

Finally, the following result (Theorem 8) will help in reasoning about subscripted data domains:

$$\mathcal{F}(A_\phi) \cong \mathcal{F}(A)_{\mathcal{F}(\phi)}$$

for most $\mathcal{F}$.

Example of Local Invariant Impact Calculation

Consider that

- the following requirement on $BAMS$ (p.50) is added to the specification:

  No account's balance can ever be lower than $\$m_0$.

- this happens after our $BAMS_5$ refinement calculation.

What is the impact of this late constraint in our calculation?

It is a local invariant of type (2.1) which can be easily handled. First, its SETS specification is:

$$\alpha \;=\; AccNr \hookrightarrow (min \circ \pi_2)$$

for $min(a) \stackrel{\text{def}}{=} a \geq \$m_0$.

Following (216), the extra work amounts to calculating $\alpha \circ f$:

$$(AccNr \hookrightarrow (min \circ \pi_2)) \circ$$
$$(AccNr \hookrightarrow [\pi_2, \pi_1] \circ (id \times dom)) \circ \bowtie \circ (mkf \times \lambda r \cdot \left(\begin{array}{c} p \\ NIL \end{array} \right)_{p \in r})$$

REMARK 32: **Main results concerning 'in loco' invariant handling in** SETS

The proof of Theorem 6 can be found in [Jou92]. The two other theorems follow.

Theorem 7 *Let $A \trianglelefteq_f^{\phi} B$ hold and $\beta : B \longrightarrow 2$ be a predicate such that the restriction of f by β,*

$$f|\phi : B_{\phi \wedge \beta} \longrightarrow A$$

is still a surjection. Then

$$A \trianglelefteq_{f|\phi}^{\phi \wedge \beta} B$$

holds.

Proof: Obvious, from the definition of the $\trianglelefteq$-order [Oli90]. □

Theorem 8 *For $\mathcal{F}$ a polynomial (endo)functor in* SETS *or the powerset functor, A a finite set and $\phi : A \longrightarrow 2$ an invariant on A, one has*

$$\mathcal{F}(A_\phi) \cong \mathcal{F}(A)_{\mathcal{F}(\phi)}$$

Proof: By induction on the structure of a polynomial functor plus the case of the powerset functor.
Concerning the constant functor, one has $C(A_\phi) = C$ which is the same as $C(A)_{C(\phi)} = C_{\lambda x.TRUE}$.
Concerning products we have:

$$\begin{array}{rcl} A_\phi \times B_\varphi & = & \{\langle a, b\rangle \mid a \in A_\phi \wedge b \in B_\varphi\} \\ & = & \{\langle a, b\rangle \mid a \in A \wedge b \in B \wedge \phi(a) \wedge \varphi(b)\} \\ & = & \{\langle a, b\rangle \mid \langle a, b\rangle \in A \times B \wedge (\phi \times \varphi)(a, b)\} \\ & = & (A \times B)_{\phi \times \varphi} \end{array}$$

Concerning co-products we have:

$$\begin{array}{rcl} A_\phi + B_\varphi & = & \{\langle 1, a\rangle \mid a \in A \wedge \phi(a)\} \cup \{\langle 2, b\rangle \mid b \in B \wedge \phi(b)\} \\ & = & \{x \mid x = \langle 1, a\rangle \wedge a \in A \wedge \phi(a) \vee x = \langle 2, b\rangle \wedge b \in B \wedge \phi(b)\} \\ & = & \{x \mid \left\{ \begin{array}{lcl} x = \langle 1, a\rangle \wedge a \in A & \Rightarrow & \phi(a) \\ x = \langle 2, b\rangle \wedge b \in B & \Rightarrow & \phi(b) \end{array} \right. \} \\ & = & (A + B)_{\phi + \varphi} \end{array}$$

Concerning finite exponentiation we have:

$$\begin{array}{rcl} (A_\phi)^C & = & \{f : C \to A \mid \forall c \in C : \phi(f(a))\} \\ & = & \{f : C \to A \mid \phi^C(f)\} \\ & = & (A^C)_{\phi^C} \end{array}$$

Finally, concerning powersets we have:

$$\begin{array}{rcl} 2^{A_\phi} & = & \{S \mid S \subseteq A_\phi\} \\ & = & \{S \subseteq A \mid \forall a \in S : \phi(a)\} \\ & = & (2^A)_{2\phi} \end{array}$$

□

Note that Theorem 8 extends to recursive functors which have a solution in Set. For instance, let A be $\mathbb{N}$ and let ϕ be the "is even" predicate $even(a) \stackrel{\text{def}}{=} \exists k \in \mathbb{N} : a = 2 \times k$. Then $X \cong 1 + \mathbb{N}_{even} \times X$ — the data-type whose values are lists of even numbers — is, by Theorem 8, is the "same" as data-type $X \cong 1 + \mathbb{N} \times X$ restricted by (recursive) local invariant

$$\psi \stackrel{\text{def}}{=} 1 + even \times \psi$$

that is:

$$\psi(l) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} l = i_1(NIL) & \Rightarrow & TRUE \\ l = i_2(\langle a, l'\rangle) & \Rightarrow & even(a) \wedge \psi(l') \end{array} \right.$$

We handle the first line composition in the first place:

$$\begin{array}{rl} & (AccNr \hookrightarrow (min \circ \pi_2)) \circ (AccNr \hookrightarrow [\pi_2, \pi_1] \circ (id \times dom)) \\ = & AccNr \hookrightarrow (min \circ \pi_2 \circ [\pi_2, \pi_1] \circ (id \times dom)) \\ = & AccNr \hookrightarrow (min \circ \pi_1 \circ (id \times dom)) \\ = & AccNr \hookrightarrow (min \circ \pi_1) \end{array}$$

Now we compose this with the remaining expression:

$$\begin{array}{rl} & (AccNr \hookrightarrow (min \circ \pi_1)) \circ \bowtie_{\hat{n}} \circ (mkf \times \lambda r \cdot \left(\begin{array}{c} p \\ NIL \end{array} \right)_{p \in r}) \\ = & (AccNr \hookrightarrow min) \circ \pi_1 \circ (mkf \times \lambda r \cdot \left(\begin{array}{c} p \\ NIL \end{array} \right)_{p \in r}) \\ = & (AccNr \hookrightarrow min) \circ mkf \circ \pi_1 \\ = & 2^{min \circ \pi_2} \circ \pi_1 \end{array}$$

that is,

$$\lambda \langle \rho, \gamma \rangle . \forall t \in \rho : \pi_2(t) \geq \$m_0$$

(see Exercise 14 for two laws used in this reasoning.)

So the "least strengthened" concrete invariant over $BAMS_5$ which accommodates such a requirement is (*cf.* p.52):

$$fdp(\rho) \wedge 2^{\pi_1}(\gamma) \subseteq 2^{\pi_1}(\rho) \wedge \forall t \in \rho : \pi_2(t) \geq \$m_0$$

CONCLUSION

Local invariants can be introduced at any stage in a SETS refinement, that is, we may ignore them in the first calculation and introduce them later, in a stepwise way.

The Hash Table Case Study

Let A be a finite domain of *data* which we want to record and maintain in a *database* file. We will regard

$$Database \;\cong\; 2^A \tag{221}$$

as a convenient (however primitive!) model for the database file itself:

$$
\begin{array}{rcl}
initialDb & : & \longrightarrow Database \\
initialDb & \stackrel{\text{def}}{=} & \emptyset \\
\hline
insert & : & A \times Database \longrightarrow Database \\
insert(a, \sigma) & \stackrel{\text{def}}{=} & \sigma \cup \{a\} \\
\hline
find & : & A \times Database \longrightarrow 2 \\
find(a, \sigma) & \stackrel{\text{def}}{=} & (a \in \sigma) \\
\hline
remove & : & A \times Database \longrightarrow Database \\
remove(a, \sigma) & \stackrel{\text{def}}{=} & \sigma - \{a\}
\end{array}
\tag{222}
$$

We want to calculate the hash-table implementation of (221). We concentrate on $Database$'s reification, the refinement of the its operators (222) being provided later.

Let $n \in \mathbf{N}$ be the number of entries of our intended hash table, and

$$H : A \longrightarrow n$$

be the selected hash function. (Usually, n is much smaller than the cardinal of A, that is, H is clearly non-injective.)

Our target is to use $\trianglelefteq$-reasoning in order to convert

$$2^A \trianglelefteq \ldots$$

into something we can accept as a model of hash tables.

Details of abstract invariant calculation concerning the steps which follow are provided in Remark 34.

REMARK 33: **Exercises**

Exercise 13 Prove the following equalities:

$$[f,h]\circ i = [f\circ i, h\circ i] \quad (223)$$
$$(j\times k)\circ[f,h] = [j\circ f, k\circ h] \quad (224)$$
$$A\hookrightarrow(f\circ g) = (A\hookrightarrow f)\circ(A\hookrightarrow g) \quad (225)$$
$$rng\circ(A\hookrightarrow f) = 2^f\circ rng \quad (226)$$
$$dom\circ(A\hookrightarrow f) = dom \quad (227)$$
$$rng\circ(i\hookrightarrow B) = rng \quad (228)$$
$$(A\hookrightarrow f)(\sigma\mid S) = ((A\hookrightarrow f)(\sigma))\mid S \quad (229)$$
$$(A\hookrightarrow f)(\sigma\cup\tau) = (A\hookrightarrow f)(\sigma)\cup(A\hookrightarrow f)(\tau) \quad (230)$$
$$(A\hookrightarrow f)(\sigma)\dagger(A\hookrightarrow h)(\tau) = (A\hookrightarrow \lambda b.\left\{\begin{array}{lcl} a\in dom(\tau) & \Rightarrow & h(b)\\ a\in dom(\sigma) & \Rightarrow & f(b)\end{array}\right.)(\sigma\dagger\tau) \quad (231)$$

□

Exercise 14 Prove that the following properties hold with respect to mkf (116,167), $nJoin$ (144) and an arbitrary (but suitably well-typed) predicate α:

$$(A\hookrightarrow(\alpha\circ\pi_1))\circ\bowtie_n = (A\hookrightarrow\alpha)\circ\pi_1 \quad (232)$$
$$(A\hookrightarrow\alpha)\circ mkf = 2^{\alpha\circ\pi_2} \quad (233)$$

□

Exercise 15 Recall invariant inv-$BAMS$ in Remark 3 and

- write it as a Set-expression;
- "push it" downto the calculation of $BAMS_5$.

□

Exercise 16 Prove that the following properties hold with respect to mkf (116,167), $nJoin$ (144) and an arbitrary (but suitably well-typed) morphism f:

$$(A\hookrightarrow(f\circ\pi_1))\circ\bowtie_n = (A\hookrightarrow f)\circ\pi_1 \quad (234)$$
$$rng\circ(A\hookrightarrow f)\circ mkf = 2^{f\circ\pi_2} \quad (235)$$

□

Exercise 17 Let

$$mkr : (A\hookrightarrow B)\longrightarrow 2^{A\times B}$$
$$mkr(\sigma) \stackrel{\text{def}}{=} \{\langle a,\sigma(a)\rangle \mid a\in dom(\sigma)\}$$

be a function which converts finite mappings into relational form. Prove or contradict the following facts about mkr:

$$mkf\circ mkr = id \quad (236)$$
$$rng\circ(A\hookrightarrow f) = 2^{f\circ\pi_2}\circ mkr \quad (237)$$

□

We start from (148), a law enabling a data domain to be refined by adding extra information to it. In our case,

$$A \trianglelefteq_{\pi_2} n \times A \tag{238}$$

where we want to impose the following local invariant on $n \times A$:

$$\phi(i, a) \stackrel{\text{def}}{=} i = H(a) \tag{239}$$

forcing every datum $a \in A$ to be coupled with its hash-index $H(a)$. Since the restriction of π_2 to $(n \times A)_\phi$ is surjective on A, that is,

$$2^{\pi_2}(\{(i, a) \in n \times A \mid \ i = H(a)\}) = A$$

we have

$$A \trianglelefteq_{\pi_2|\phi} (n \times A)_\phi \tag{240}$$

by Theorem 7. By applying the powerset functor $\lambda X.2^X$ to (240) we obtain

$$2^A \trianglelefteq_{f_1} (2^{n \times A})_{\Phi_1} \tag{241}$$

for

$$\begin{aligned} \Phi_1 &= \lambda s.(\forall (i, a) \in s : i = H(a)) \\ f_1 &= \lambda s.\{a \in A \mid \ (i, a) \in s\} \end{aligned}$$

We now resort to the following instance of the "Curry/uncurry" law of exponentiation (107), for $A = 2$:

$$2^{n \times A} \cong (2^A)^n \tag{242}$$

The result of "pushing" Φ_1 along step (242) is concrete invariant Φ_2 in

$$(2^{n \times A})_{\Phi_1} \cong_{f_2} ((2^A)^n)_{\Phi_2} \tag{243}$$

which can be calculated via Theorem 6:

$$\Phi_2 \ = \ \lambda t.\forall i \in n : (\forall a \in t(i) : i = H(a)) \tag{244}$$

We also have

$$f_2 \;=\; \lambda t.\{\langle i, a\rangle \mid\; i \in n \wedge a \in t(i)\}$$

It is easy to see that $((2^A)^n)_{\Phi_2}$ above — *cf.*

$$\texttt{array}[n]\ \texttt{of}\ 2^A$$

is in fact a model for hash tables, *provided that* Φ_2 holds.

According to Φ_2, every set in the range of an array of such a data-type contains *synonyms* [HS78]. All such sets are mutually disjoint (see Lemma 1). Therefore, every $db \in 2^A$ is partitioned into n-many, disjoint "collision segments", each one addressed by the relevant hash index, which is computed by H.

As expected, the overall abstraction function retrieves the abstract database by re-assembling all disjoint collision sets:

$$\begin{array}{rcl} f_1(f_2(t)) & = & \{a \in A \mid\; (i,a) \in \{(i',a') \mid\; i' \in n \wedge a' \in t(i')\}\} \\ & = & \{a \in A \mid\; i \in n \wedge a \in t(i)\} \\ & = & \{a \in A \mid\; \exists i \in n : a \in t(i)\} \\ & = & \{a \in A \mid\; \bigvee_{i \in n} a \in t(i)\} \\ & = & \bigcup_{i \in n} \{a \in A \mid\; a \in t(i)\} \\ & = & \bigcup_{i \in n} t(i) \end{array} \tag{245}$$

The intended "hashing effect" has thus already been achieved.

Collision Handling

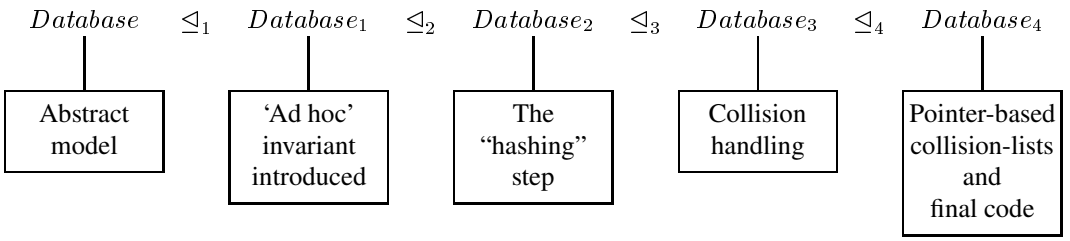
The task of reifying 2^A "recurs" in the co-domain of $n \to 2^A$ — a kind of "fractal" effect on this type of data structuring.

COMMENTS

- This problem is identified in the literature under the headings *overflow handling* or *collision handling* [HS78].
- See remarks 35 to 37 for a calculational approach to this problem which will lead us to the classical approach (quoting [HS78]):

  > *Many of the comparisons being made could be saved if we maintained lists of identifiers, one list per bucket, each list containing all the synonyms for that bucket (...) Since the size of these lists is not known in advance, the best way to maintain them is as linked chains.*

The following diagram provides a summary of the overall reasoning:



(246)

Operation Refinement: Merging SETS with Other Calculi

Quotation from [Mor90], chapter 17, pp. 173–174:

> *The laws of Section 17.4 dealt with a very general case of data refinement, in which the coupling invariant linking the abstract and concrete states could be anything whatever.* [...]
>
> *This is actually a fairly rare occurrence in everyday program development however: it is much more common for the coupling invariant to be functional from concrete to abstract.* [...]
>
> *The general form for such coupling invariants, called* functional abstractions *is*

$$a = \mathsf{af}\ c \wedge \mathsf{dti}\ c\ , \qquad (17.1)$$

REMARK 34: **Details of proposed hash table calculation**

Step (242) is obtained via the "Curry/uncurry" law of exponentiation (107), established by the $uncurry$ bijection (169). For $A = 2$ we get the following version of $uncurry$:

$$uncurry(t) \stackrel{\text{def}}{=} \{\langle b, c\rangle \mid b \in B \wedge c \in t(b)\} \tag{247}$$

The first step of the abstraction invariant calculation is as follows:

$$\begin{array}{rcl} \Phi_1 & = & 2^{\phi} \\ & = & \lambda s.(\forall x \in s : \phi(x)) \\ & = & \lambda s.(\forall (i,a) \in s : i = H(a)) \end{array}$$

and

$$\begin{array}{rcl} f_1 & = & 2^{(\pi_2 | \phi)} \\ & = & \lambda s.\{\pi_2(x) \mid x \in s\} \\ & = & \lambda s.\{a \in A \mid (i,a) \in s\} \end{array}$$

From the second step we infer f_2 and Φ_2 as follows:

$$\begin{array}{rcl} f_2 & = & \lambda t.uncurry(t) \\ & = & \lambda t.\{\langle i, a\rangle \mid i \in n \wedge a \in t(i)\} \end{array}$$

and

$$\begin{array}{rcl} \Phi_2 & = & \Phi_1 \circ uncurry \\ & = & \lambda t.\Phi_1(\{(i,a) \mid i \in n \wedge a \in t(i)\}) \\ & = & \lambda t.\forall i \in n : (\forall a \in t(i) : i = H(a)) \end{array} \tag{248}$$

Concrete invariant Φ_2 underpins the following lemma:

Lemma 1 *For every hash table $t \in (n \to 2^A)_{\Phi_2}$ we have*

$$\forall i \neq j \in n : t(i) \cap t(j) = \emptyset$$

Proof: By contradiction, suppose that, for particular $i \neq j$ and $a \in A$, we have

$$a \in t(i) \wedge a \in t(j)$$

Then, by Φ_2 (248) one has

$$i = H(a) \wedge j = H(a)$$

that is, $i = j$ thus contradicting the starting assumption $i \neq j$. □

Exercise 18 Derive the "set-theoretical" version of $uncurry$ which is given by (247). **Hint**: resort to the following isomorphisms explicitly:

$$2^A \xrightarrow{\lambda p.\{a \in A \mid p(a)\}} \mathcal{P}(A) \qquad \text{and} \qquad \mathcal{P}(A) \xrightarrow{\lambda s.(\lambda a.a \in s)} 2^A \tag{249}$$

□

Exercise 19 Recall Exercise 12, in which four local invariants were added to the PPD data model specification of p.52. Resort to Theorem 6 to calculate the weakest impact of these invariants, one at a time, into $RPPD$ (the final relational data model obtained on p.54).

□

REMARK 35: **Hash-table collision handling: a detailed calculation (1)**

Since refinement steps in SETS are compositional, and bearing in mind that we are trying to refine sets into hash tables, what about doing the same for each collision set?
This solution is known as *rehashing* and leads to something like

$$n \to (m \to 2^A) \tag{250}$$

under a more elaborate invariant involving n possibly different sub-hashing functions H_i $(i = 1, \ldots, n)$:

$$\Phi_3 \quad = \quad \lambda t. \forall i \in n : (\forall j \in m : (\forall a \in t(i)(j) : i = H(a) \wedge j = H_i(a))) \tag{251}$$

But, strictly speaking, *rehashing* is nothing but multiplying the address space of a given hashing table by some factor (m), since

$$((2^A)^m)^n \cong (2^A)^{n \times m}$$

— *cf.* law (107) — where the hash function is

$$H'(a) = \langle H(a), H_{H(a)}(a) \rangle$$

Typically $m < n$, and collision sets get smaller, but 2^A still recurs in (250).
For a more effective solution to the problem we need to analyse the reification theory of 2^A, and many options are available. The most immediate is perhaps given by law (132), turning hash tables into arrays of sequences of collisions:

$$(n \to A^\star)_{\Phi_3}$$

The corresponding invariant Φ_3 is obtained via Theorem 1, whereby

$$2^A \trianglelefteq_{elems} A^\star \quad \Rightarrow \quad (2^A)^n \trianglelefteq_{elems^n} (A^\star)^n$$

for

$$elems^n(t) = \lambda i. elems(t(i))$$

cf. table of page p.44. By Theorem 6 one has,

$$(2^A)^n \trianglelefteq_{elems^n} (A^\star)^n \quad \Rightarrow \quad ((2^A)^n)_{\Phi_2} \trianglelefteq_{elems^n} ((A^\star)^n)_{\underbrace{\Phi_2 \circ elems^n}_{\Phi_3}} \tag{252}$$

that is,

$$\begin{array}{lcl} \Phi_3 & = & \Phi_2 \circ elems^n \\ & = & \lambda t. \forall i \in n : (\forall a \in elems(t(i)) : i = H(a)) \end{array}$$

$A^\star$ is not readily available in programming languages like C or PASCAL. Its reification towards such programming environments has been studied earlier on — recall remarks 24 and 29 — as an illustration of refining recursive data types: first, recursion is *introduced* by realizing that $A^\star$ is a well-known fixpoint solution of the functorial equation

$$X \quad \cong \quad 1 + A \times X \quad \textit{/*finite lists on } A \textit{ */} \tag{253}$$

Then recursion is *removed* by reifying the recursive pattern of (253) in terms of references or pointers $(K + 1)$ which are either NIL or point to a location in a *heap* or *object store*, addressed by K, a natural-number isomorphic domain of "keys":

$$A^\star \quad \trianglelefteq \quad \underbrace{(K \hookrightarrow A \times (K+1))}_{\text{heap}} \times \underbrace{(K+1)}_{\text{pointer}} \tag{254}$$

Calculation proceeds in the remark which follows.

REMARK 36: **Hash-table collision handling: a detailed calculation (2)**

The abstraction function at (254) is:

$$f_{(254)}(\sigma, x) \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{rcll} x = i_2(NIL) & \Rightarrow & <> & \\ x = i_1(k) & \Rightarrow & let & a = \pi_1(\sigma(k)) \\ & & & x' = \pi_2(\sigma(k)) \\ & & in & <a> \frown f_{(254)}(\sigma, x')) \end{array} \right.$$

which is coupled to the following low-level invariant, ensuring acyclic pointer definedness,

$$\phi_{(254)}(\sigma, x) \quad \stackrel{\text{def}}{=} \quad \gamma(x, \sigma, \emptyset)$$

where:

$$\gamma(x, \sigma, C) \stackrel{\text{def}}{=} \left\{ \begin{array}{rcl} x = i_2(NIL) & \Rightarrow & TRUE \\ x = i_1(k) \wedge k \notin dom(\sigma)) \vee k \in C & \Rightarrow & FALSE \\ x = i_1(k) \wedge k \in dom(\sigma)) & \Rightarrow & \gamma(\pi_2(\sigma(k)), \sigma, C \cup \{k\}) \end{array} \right. \tag{255}$$

Concerning $(A^\star)^n$ we will have:

$$(A^\star)^n \quad \trianglelefteq \quad (K \hookrightarrow A \times (K+1)) \times (K+1)^n \tag{256}$$

(leading to an array of pointers to linked-lists sharing the same heap structure) where

$$f_{(256)}(\sigma, t) \quad \stackrel{\text{def}}{=} \quad \left(\begin{array}{c} i \\ f_{(254)}(\sigma, t(i)) \end{array} \right)_{i \in n}$$

and

$$\phi_{(256)}(\sigma, t) \quad \stackrel{\text{def}}{=} \quad \forall i \in n : \phi_{(254)}(\sigma, t(i))$$

The overall invariant at this point,

$$\Phi_4 \quad = \quad \phi_{(256)} \wedge \Phi_3 \circ f_{(256)}$$

can be systematically calculated. First, we have:

$$\begin{array}{ll} \Phi_4(\sigma, t) = & \\ \quad \forall i \in n : \phi_{(254)}(\sigma, t(i)) \wedge \Phi_3 \circ f_{(256)} & \\ = \quad (\forall i \in n : \gamma(t(i), \sigma, \emptyset)) \wedge \forall i \in n : (\forall a \in elems(f_{(254)}(\sigma, t(i))) : i = H(a)) & \\ = \quad \forall i \in n : \gamma(t(i), \sigma, \emptyset) \wedge (\forall a \in elems(f_{(254)}(\sigma, t(i))) : i = H(a)) & \\ = \quad \forall i \in n : \gamma(t(i), \sigma, \emptyset) \wedge \varphi(t(i), \sigma, i) & \end{array}$$

where φ is easily shown to be the following predicate:

$$\varphi(k, \sigma, i) \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{rcll} k = NIL & \Rightarrow & TRUE & \\ \neg(k = NIL) & \Rightarrow & let & a' = \pi_1(\sigma(k)) \\ & & & k' = \pi_2(\sigma(k)) \\ & & in & i = H(a) \wedge \varphi(k', \sigma, i) \end{array} \right.$$

The remaining reasoning is presented in the remark which follows.

REMARK 37: **Hash-table collision handling: a detailed calculation (3)**

Further "fold-unfold" reasoning will lead to

$$\Phi_4(\sigma, t) \quad = \quad \forall i \in n : \Phi_{4aux}(t(i), \sigma, i, \{\}) \tag{257}$$

where Φ_{4aux} is an auxiliary predicate merging γ and φ, which realizes the expected linked-list downwards traversal checking for simultaneous hash-index consistency and pointer definedness and acyclicness:

$$\Phi_{4aux}(k, \sigma, i, C) \stackrel{\text{def}}{=}$$
$$\begin{cases} k = NIL & \Rightarrow \quad TRUE \\ k \neq NIL \\ \quad \wedge \\ k \notin dom(\sigma) & \Rightarrow \quad FALSE \\ \quad \vee \\ \quad k \in C \\ k \in dom(\sigma) & \Rightarrow \quad let \quad a' = \pi_1(\sigma(k)) \\ & \qquad\quad\; k' = \pi_2(\sigma(k)) \\ & \quad\; in \quad\; i = H(a) \wedge \Phi_{4aux}(k', \sigma, i, C \cup \{k\}) \end{cases}$$

Putting everything together, we have calculated the following data structure:

$$((K \hookrightarrow A \times (K+1)) \times (K+1)^n)_{\Phi_4} \tag{258}$$

The final step would be to encode (258) (with some extra "syntactic sugar") in a commercial programming language, *e.g.* C

```
typedef  struct Collision {
         A Synonym;
         struct Collision *Next;
         };
typedef  struct Collision *Bucket;
typedef  Bucket Database4[n]; /* hash table */
```
(259)

or PASCAL

```
type  Collision  =  record
                       Synonym:  A;
                       Next:  ^ Collision
                    end;
      Bucket     =  ^ Collision;
      Database4  =  (* hash table *)
                    array[1..n] of Bucket;
```
(260)

in the documentation of which Φ_4 (257) should be recorded, at conceptual level.

Exercise 20 Suppose that law (133) was used for the purpose of handling hash-table collision sets, that is, one has:

$$(\mathbb{N} \hookrightarrow A)^n$$

instead of $(A^\star)^n$. Calculate abstraction invariant components f_3, Φ_3 for this situation.
□

Exercise 21 A constant function is an obvious "bad" choice for hash function, *e.g.* $H(a) \stackrel{\text{def}}{=} i_0$, for $i_0 \in n$. Show that, in this case, Φ_2 (248) reduces to

$$\lambda t. \forall i \in n : i = i_0 \vee t(i) = \emptyset$$

□

> *where* af *is a function, called the* abstraction function *and* dti *is a predicate, in which* a *does not appear, called the* data-type invariant. [. . .]
>
> *The reason for our interest in the special cases of data refinement is that when the augmentation and diminution laws are specialised to coupling invariants of the form (17.1) they become very much simpler, and the augmentation and diminution may be done together in one step.*

This quotation is self-explanatory concerning the close link between SETS and the OXFORD REFINEMENT CALCULUS, while preserving their relative orthogonality:

- The REFINEMENT CALCULUS is a *weakest pre-condition* [Dij76] (algorithmic) calculus in which *change of representation* is handled by *chosing* coupling invariants.
- SETS' main purpose is that of *calculating* (instead of chosing) such coupling invariants.

Therefore, for the operation-level follow up the SETS calculations performed in this course — concrete operation synthesis — please "commute" to the REFINEMENT CALCULUS [Mor90].

However, this is not the only method available for concrete operation realization.

We will provide details below about another blend of SETS: that with the (perhaps) less known program-transformation style — the FOLD/UNFOLD CALCULUS [BD77, Dar82].

Program Synthesis by Solving Reification Diagrams

Transformational algorithmic refinement (concrete operation calculation) involves the following steps:

1. Calculation of a concrete-level simulation of the high-level operation (or "get rid of the abstraction function").

2. Calculation of an efficient version of 1. (*e.g.* "get rid of recursion" *etc.*).
3. Encoding of 2. (or "get rid of mathematics altogether!").

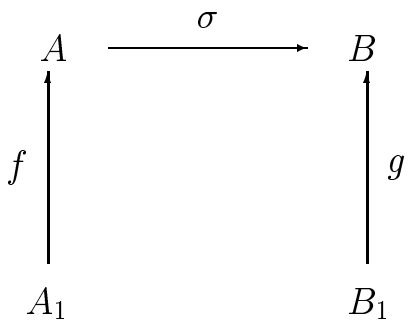
In general, let

$$\sigma : A \longrightarrow B$$

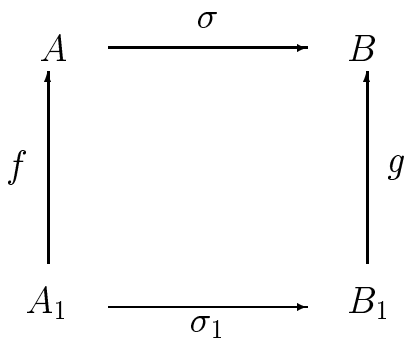
be a Set-morphism specifying some operation involving data-types A and B. (See later a generalization of σ to pre/post-condition operational specification.)

Step 1:

Suppose we have already used SETS to refine A and B:



The idea is to "close" the above diagram,



so that σ_1 "simulates" the behaviour of σ at concrete-level,

$$\forall a_1 \in A_1 : g(\sigma_1(a_1)) = \sigma(f(a_1)) \tag{261}$$

that is, the diagram commutes.

We will regard σ_1 as the "unknown" of equation (261).

Note that there may be many solutions for σ_1 in equation (261) — the typical "one-abstract to many-concrete" relationship of (functional) refinement.

"Simulation" amounts to exhibiting the same observational behaviour:

$$\begin{array}{rcl} \sigma(a) & \stackrel{\text{def}}{=} & let \quad a_1 \in \{a' \in A_1 \mid \ f(a') = a\} \\ & & in \quad g(\sigma_1(a_1)) \end{array}$$

An injective range abstraction map g will lead to unique solutions because g^{-1} exists and diagram also commutes in this way:

$$\sigma_1(a_1) = g^{-1}(\sigma(f(a_1)))$$

Then the idea is to calculate...

$$\begin{array}{rcl} \sigma_1(a_1)) & = & g^{-1}(\sigma(f(a_1))) \\ & = & \ldots \\ & \vdots & \\ & = & \ldots \sigma_1 \ldots a_1 \ldots \quad \textit{/*expression free of } f \textit{ and } g \textit{ */} \end{array}$$

For non-injective g there may exist more than one solution and idea is to transform refinement equation (261) so that f eventually gives place to g and g is eventually "pulled" up to the outermost place on the righthand side...

$$\begin{array}{rcl} g(\sigma_1(a_1)) & = & \sigma(f(a_1)) \\ & = & \ldots \\ & \vdots & \\ & = & \ldots g \ldots \sigma_1 \ldots a_1 \ldots \\ & \vdots & \\ & = & g(E(\ldots \sigma_1 \ldots a_1 \ldots)) \end{array}$$

so that abstraction map g can be factored out from both sides of the obtained equation:

$$\sigma_1(a_1) \stackrel{\text{def}}{=} \ldots E(\ldots \sigma_1 \ldots a_1 \ldots)$$

(Of course, there may exist another E' such that $g(E'(\ldots \sigma_1 \ldots a_1 \ldots))$ can also be obtained by alternative calculation.)

Step 2:

- "Efficiency" can be obtained in many ways which are dependent on the target machine architecture.
- Concrete version of overall local invariant can be brought in (if not *already* considered in step 1) if it helps in reducing time/space complexity.
- In the case of a target sequential state-based machine (programming language), recursion-removal transformations are very often used to synthesize `for/while`-loops from recursive equations.

See examples later on.

Step 3:

Gode generation consists of carrying "program" transformation even further,

$$\begin{array}{rcl} & \vdots & \\ \sigma_1(a_1) & = & \ldots E \ldots a_1 \ldots \\ & = & \ldots \\ & \vdots & \\ & = & [\![\mathtt{P}]\!] \end{array}$$

until the formal semantics $[\![\mathtt{P}]\!]$ of some executable piece of code `P` (in the target programming language) is found.

Concrete Invariants in Refinement Equations

In general, f and g have associated concrete invariants, say ϕ and ψ, respectively.

Refinement equation (261) becomes

$$\forall a_1 \in A_1 : \phi(a_1) \Rightarrow \psi(\sigma_1(a_1)) \wedge g(\sigma_1(a_1)) = \sigma(f(a_1))$$

What About σ?

The equations above regard σ as a (total) Set-morphism, that is, a total function. Although a simplistic assumption, all other situations (but one, see below) resort to it in a way or another.

Partial Operations

Wherever σ is partial, pre-σ (its pre-condition) has to be added to the refinement equation:

$$\forall a_1 \in A_1 : \phi(a_1) \wedge \text{pre-}\sigma(f(a_1)) \Rightarrow \psi(\sigma_1(a_1)) \wedge g(\sigma_1(a_1)) = \sigma(f(a_1))$$

Deterministic State Transformers

The typical constructive specification model involves operations which may deliver a result while changing the model's internal state. This can only be captured by a post-condition predicate

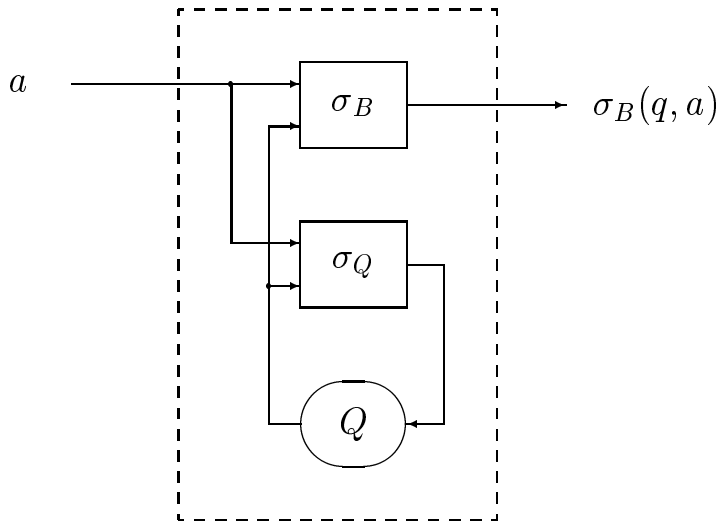
$$\text{post-}\sigma \ : \ Q \times A \times Q \times B \longrightarrow 2$$

(Q = internal state data-type).

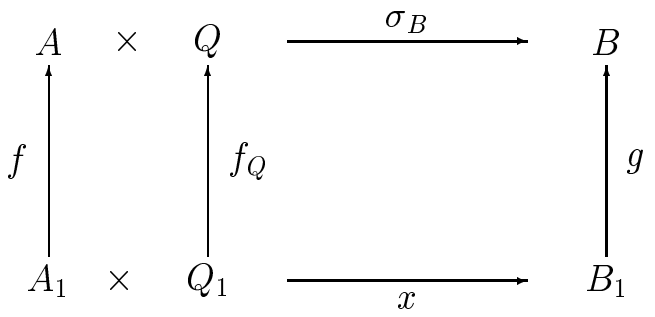
In many situations the behaviour of σ is deterministic in the sense that it can be "factored" into two functions: a result function σ_B and a state transforming function σ_Q:

$$\text{post-}\sigma(q, a, q', b') \ \stackrel{\text{def}}{=} \ q' = \sigma_Q(q, a) \wedge b' = \sigma_B(q, a)$$

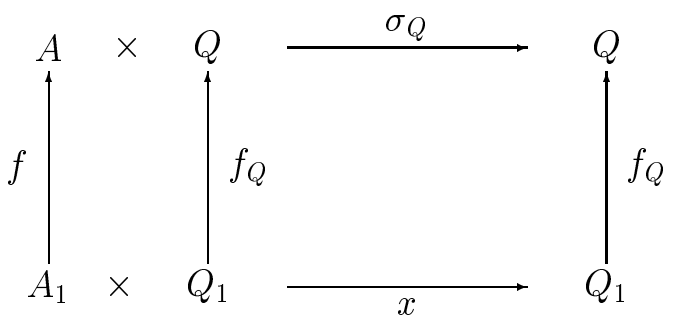
In a picture:



So we just have to refine functions σ_Q and σ_B, that is, solve for x the equations implicit in diagrams



and



but please note:

- In many situations A and B stay the same ($f = 1_A$ and $g = 1_B$) and only Q is refined.

- Both σ_Q and σ_B are partial, in general.

NON-DETERMINISTIC STATE TRANSFORMERS

In this case post-$\sigma(q, a, q', b')$ is an arbitrary predicate. We may still regard it as a Boolean-function and adapt refinement equation (261) to this situation, for f_Q the state abstraction function.

Leaving invariants and pre-conditions aside, for the moment, we will have:

$$\begin{array}{l} \forall q_1, q_1' \in Q_1, a_1 \in A_1, b_1' \in B_1 : \\ \quad \text{post-}\sigma_1(q_1, a_1, q_1', b_1') = \\ \quad \text{post-}\sigma(f_Q(q_1'), f(a_1), f_Q(q'), g(b_1')) \end{array} \tag{262}$$

But logical equality ($=$) in (262) is too strong: it should be relaxed to implication ($\Rightarrow$),

$$\begin{array}{l} \forall q_1, q_1' \in Q_1, a_1 \in A_1, b_1' \in B_1 : \\ \quad \text{post-}\sigma_1(q_1, a_1, q_1', b_1') \Rightarrow \\ \quad \text{post-}\sigma(f_Q(q_1'), f(a_1), f_Q(q'), g(b_1')) \end{array} \tag{263}$$

since increasing determinism is a valid refinement decision (recall the *strengthen postcondition* law of [Mor90]).

Last but not least...

If an operation is known to preserve an invariant α, then any of its proper reifications, as entailed by the SETS calculus, will preserve it as well (see Theorem 9).

Examples of Operation Refinement

Example 1 — Calculating a Typical Database Update Operation

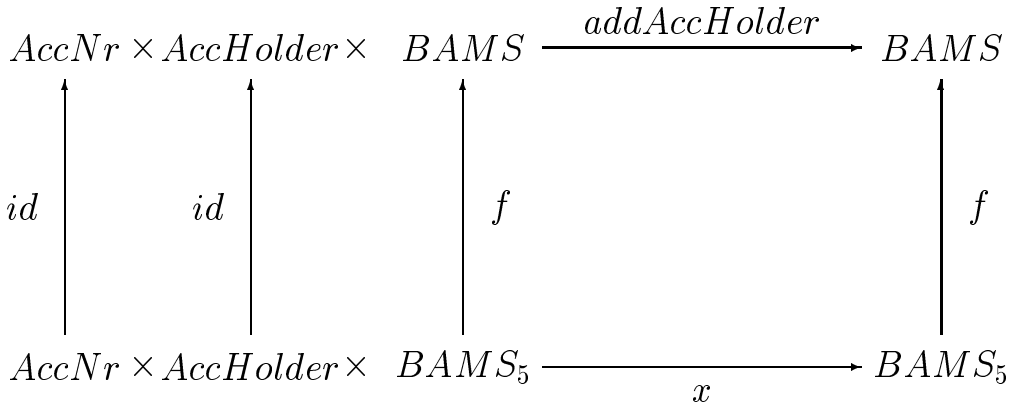
Consider the following operation on $BAMS$,

$$
\begin{array}{l}
addAccHolder : AccNr \times AccHolder \times BAMS \longrightarrow BAMS \\
addAccHolder(k, t, \sigma) \stackrel{\text{def}}{=} \\
\left\{ \begin{array}{lcll}
k \in dom(\sigma) & \Rightarrow & let & x = \sigma(k) \\
 & & & t_k = \pi_1(x) \\
 & & & q_k = \pi_2(x) \\
 & & in & \sigma \dagger \left(\begin{array}{c} k \\ \langle t_k \cup \{t\}, q_k \rangle \end{array} \right) \\
\neg(k \in dom(\sigma)) & \Rightarrow & \sigma &
\end{array} \right.
\end{array}
\tag{264}
$$

which specifies the "add a new account holder to given account" functionality.

We want to calculate its refinement at level $BAMS_5$ (see p.50).

Refinement diagram:



Refinement equation:

$$
f(x(k, t, \langle \rho, \gamma \rangle)) \;=\; addAccHolder(k, t, f(\langle \rho, \gamma \rangle)) \tag{265}
$$

where x is the unknown and f has been calculated before.

Let us start by replacing σ in $addAccHolder$'s body by $f(\langle \rho, \gamma \rangle)$. Such a substitution entails several others, as follows:

$$
\begin{array}{rcl}
\sigma & \longrightarrow & f(\langle \rho, \gamma \rangle) \\
k \in dom(\sigma) & \longrightarrow & k \in 2^{\pi_1}(\rho)
\end{array}
$$

$$\begin{array}{rcl} \sigma(k) & \longrightarrow & \langle \{\pi_2(p) \mid\ p \in \gamma \wedge \pi_1(p) = k\}, the(\{b \mid\ k\ \rho\ b\}) \rangle \\ t_k \cup \{t\} & \longrightarrow & \{\pi_2(p) \mid\ p \in \gamma \wedge \pi_1(p) = k\} \cup \{t\}, \\ q_k & \longrightarrow & the(\{b \mid\ k\ \rho\ b\}) \end{array}$$

Clearly,

$$\{\pi_2(p) \mid\ p \in \gamma \wedge \pi_1(p) = k\} \cup \{t\} \ = \ \{\pi_2(p) \mid\ p \in \gamma \cup \{\langle k, t \rangle\} \wedge \pi_1(p) = k\}$$

Then expression

$$\langle \{\pi_2(p) \mid\ p \in \gamma \cup \{\langle k, t \rangle\} \wedge \pi_1(p) = k\}, the(\{b \mid\ k\ \rho\ b\}) \rangle$$

reduces to

$$f(\langle \rho, \gamma \cup \{\langle k, t \rangle\} \rangle)(k)$$

and, so,

$$\begin{array}{rcl} & & \sigma \dagger \begin{pmatrix} k \\ \langle t_k \cup \{t\}, q_k \rangle \end{pmatrix} \\ & \longrightarrow & f(\langle \rho, \gamma \rangle) \dagger \begin{pmatrix} k \\ f(\rho, \langle \gamma \cup \{\langle k, t \rangle\} \rangle)(k) \end{pmatrix} \qquad (266) \\ & = & f(\rho, \langle \gamma \cup \{\langle k, t \rangle\} \rangle) \qquad (267) \end{array}$$

Thus equation (265) re-writes into

$$\begin{array}{rcl} f(x(k, t, \langle \rho, \gamma \rangle)) & = & \left\{ \begin{array}{rcl} k \in 2^{\pi_1}(\rho) & \Rightarrow & f(\langle \rho, \gamma \cup \{\langle k, t \rangle\} \rangle) \\ \neg(k \in 2^{\pi_1}(\rho)) & \Rightarrow & f(\langle \rho, \gamma \rangle) \end{array} \right. \\ & = & f(\left\{ \begin{array}{rcl} k \in 2^{\pi_1}(\rho) & \Rightarrow & \langle \rho, \gamma \cup \{\langle k, t \rangle\} \rangle \\ \neg(k \in 2^{\pi_1}(\rho)) & \Rightarrow & \langle \rho, \gamma \rangle \end{array} \right.) \end{array}$$

or, removing f from both sides and simplifying, into:

$$x(k, t, \langle \rho, \gamma \rangle) \ = \ \langle \rho, \left\{ \begin{array}{rcl} k \in 2^{\pi_1}(\rho) & \Rightarrow & \gamma \cup \{\langle k, t \rangle\} \\ \neg(k \in 2^{\pi_1}(\rho)) & \Rightarrow & \gamma \end{array} \right. \rangle$$

This concludes step 1, and we don't bother about step 2 in this case because we can already spot executable SQL code at this stage:

$$\begin{array}{l} x(k, t) \stackrel{\text{def}}{=} \\ \quad \lambda\langle \rho, \gamma \rangle.\ let \quad b = k \in [\![\texttt{SELECT ALL } AccNr \texttt{ FROM } \rho]\!] \\ \qquad\qquad in \quad \langle \rho, \left\{ \begin{array}{rcl} b & \Rightarrow & [\![\texttt{INSERT INTO } \gamma \texttt{ VALUES (}k\texttt{,}t\texttt{)}]\!] \\ \neg(b) & \Rightarrow & \gamma \end{array} \right. \rangle \end{array}$$

Final decisions may include the encoding of the outermost conditional expression as an "if-then-else" in say C, which will then call the embedded SQL code — a typical ORACLE Pro-C/SQL solution.

COMMENTS:

- A formal semantics of SQL, which has to become explicit at this stage, has been defined by [Rod93] in the context of SETS.
- Many constructs of SQL have a "functorial"-like semantics (of the 2^f pattern), so it is easy to "spot" pieces of mathematics which can be replaced by $[\![\ldots]\!]$ embraced code.
- Strictly speaking, the formal semantics of the C-embedding process above should also be taken into account.

Example 2 — A Local Invariant Dependent Calculation

Our example concerning local invariants will be the refinement of the $find$ operation (222) at hash-table level, that is, under abstraction function f (245) and concrete invariant Φ_2 (248).

Refinement diagram is:

$$\begin{array}{ccc} A \times 2^A & \xrightarrow{\quad find \quad} & 2 \\ \uparrow{\scriptstyle 1_A \times (f_1 \circ f_2)} & & \uparrow{\scriptstyle 1_2} \\ A \times ((2^A)^n)_{\Phi_2} & \xrightarrow[\quad find_2 \quad]{} & 2 \end{array}$$

leading to refinement equation:

$$find_2(a, t) = find(a, f_1(f_2(t)))$$

Reasoning is carried out as follows:

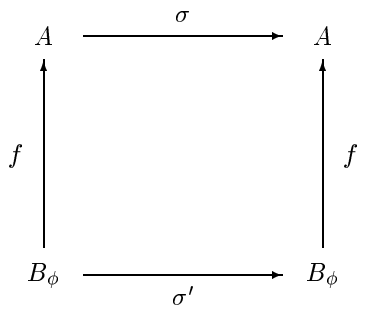
$$\begin{array}{rcl} find_2(a, t) & = & find(a, f_1(f_2(t))) \\ & = & a \in f_1(f_2(t)) \end{array}$$

REMARK 38: **A theorem concerning local invariant preservation**

Theorem 9 *(This theorem is stated for unary σ with no loss in generality: the result will be valid still for any extension of the arity of σ: $\sigma : \ldots \times A \times \ldots \to A$.) Let $\sigma : A \longrightarrow A$ be the specification of a function transforming data of sort A while preserving some ad hoc invariant α on A, that is,*

$$\forall a \in A : \alpha(a) \Rightarrow \alpha(\sigma(a)) \tag{268}$$

Then the reification σ' of σ implied by reification step (119),



will still preserve α at B-level.

Proof: σ' is any function satisfying

$$\forall b \in B_\phi : f(\sigma'(b)) = \sigma(f(b)) \tag{269}$$

σ' will preserve α at B-level iff

$$\forall b \in B_\phi : \alpha(f(b)) \Rightarrow \alpha(f(\sigma'(b)))$$

By contradiction, α will not be preserved if

$$\exists b \in B_\phi : \alpha(f(b)) \wedge \neg\alpha(f(\sigma'(b)))$$

that is,

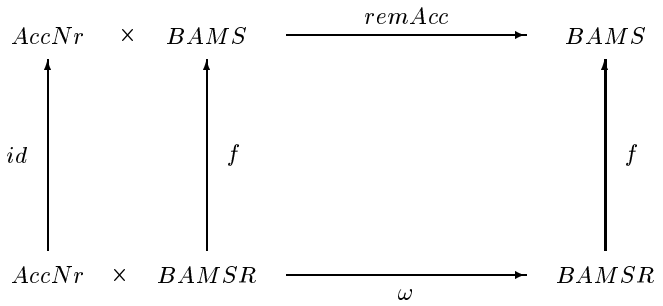
$$\exists b \in B_\phi : \alpha(f(b)) \wedge \neg\alpha(\sigma(f(b)))$$

via (269). Because f is surjective, $f(b) = a$ for some $a \in A$. Thus we will have

$$\exists a \in A : \alpha(a) \wedge \neg\alpha(\sigma(a))$$

thus contradicting our hypothesis (268). □

Exercise 22 Calculate the relational implementation, at $BAMS_5$ level (see p.50), of the "remove account" operator over its ancestor, $BAMS$, that is, solve for ω the following refinement diagram:



where

$$remAcc(k, \sigma) \stackrel{\text{def}}{=} \sigma \setminus \{k\} \tag{270}$$

Hint: It may help to consider the following relational operator for tuple-deletion,

$$del(\langle i, a\rangle, t) \stackrel{\text{def}}{=} \{r \in t \mid a \neq \pi_i(r)\}$$

which after all has the opposite behaviour of sel (17).

□

$$
\begin{aligned}
&= a \in \bigcup_{i \in n} t(i) \\
&= \bigvee_{i \in n} a \in t(i) \\
&= \exists i \in n : a \in t(i)
\end{aligned}
$$

Now, taking invariant Φ_2 into account, one has:

$$
\begin{aligned}
\bigvee_{i \in n} a \in t(i) &= a \in t(H(a)) \vee \bigvee_{i \in n \wedge i \neq H(a)} a \in t(i) \\
&= (a \in t(H(a)) \vee \bigvee_{i \in n \wedge i \neq H(a)} FALSE \\
&= a \in t(H(a)) \\
&= find(a, t(H(a)))
\end{aligned}
$$

In summary,

$$find_2(a, t) = find(a, t(H(a)))$$

It can be easily observed that $find_2$ is nothing but $find$ tuned to the relevant collision bucket. Since the cardinality of $t(H(a))$ is smaller than that of $\cup_{i \in n} t(i)$, the efficiency gain is obvious.

See remarks 39, 40 and 41 for the calculation of the other operations specified in (222).
It is interesting to observe that the "fractal"-like nature of hash-table data-level refinement turns up in every operation refinement — every operation ends up behaving like its ancestor but its effect is confined to the relevant hash-index collision bucket, a special kind of *promotion* as understood in the Z literature [Woo92].

Example 3 — `for/while` Loop Calculation

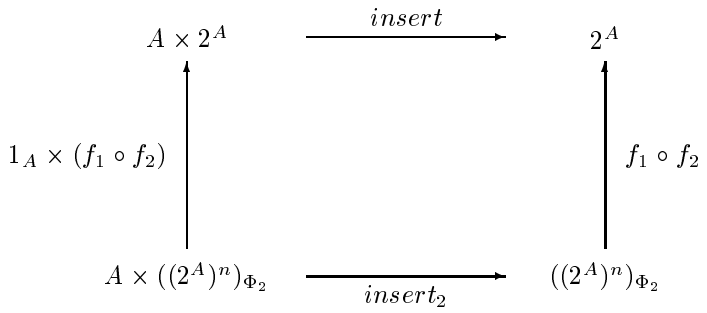
We will start here from the fairly academic refinement equation

$$belongs(x, y) = x \in elems(y) \tag{271}$$

relative to the diagram of the $find$ operation in the "sets replaced by lists" refinement:

REMARK 39: **Hash-table operation synthesis: the $insert$ function**

Refinement diagram:



Refinement equation:

$$f_1(f_2(insert_2(a,t))) = insert(a, f_1(f_2(t)))$$

Reasoning:

$$\begin{aligned}
f_1(f_2(insert_2(a,t))) &= insert(a, f_1(f_2(t))) \\
&= \{a\} \cup f_1(f_2(t)) \\
&= \{a\} \cup \bigcup_{i \in n} t(i) \\
&= \{a\} \cup t(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} t(i) \\
&= t'(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} t(i)
\end{aligned}$$

where t' is the one-entry mapping

$$t' = \begin{pmatrix} H(a) \\ \{a\} \cup t(H(a)) \end{pmatrix}$$

Then,

$$\begin{aligned}
t'(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} t(i) &= \bigcup_{i \in n} \begin{cases} i = H(a) & \Rightarrow \quad t'(i) \\ \neg(i = H(a)) & \Rightarrow \quad t(i) \end{cases} \\
&= \bigcup_{i \in n} \begin{cases} i \in dom(t') & \Rightarrow \quad t'(i) \\ \neg(i \in dom(t')) & \Rightarrow \quad t(i) \end{cases} \\
&= \bigcup_{i \in n} (t \dagger t')(i) \\
&= f_1(f_2(t \dagger t'))
\end{aligned}$$

In summary,

$$f_1(f_2(insert_2(a,t))) = f_1(f_2(t \dagger \begin{pmatrix} H(a) \\ \{a\} \cup t(H(a)) \end{pmatrix}))$$

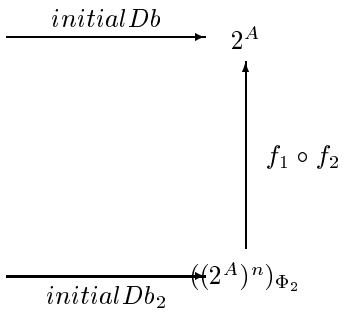
or, by removing $f_1 \circ f_2$ from both sides of the equality,

$$insert_2(a,t) = t \dagger \begin{pmatrix} H(a) \\ insert(a, t(H(a)) \end{pmatrix}$$

It can be observed that $insert_2$ is nothing but $insert$ confined to the relevant collision bucket, taking advantage of its smaller cardinality in subsequent refinements such as depicted in (246).

REMARK 40: **Hash-table operation synthesis: the $initialDb$ and $remove$ functions**

Refinement diagram for $initialDb$:



Refinement equation:

$$f_1(f_2(initialDb_2)) = initialDb$$

Reasoning: since

$$\bigcup_{i \in n} initialDb_2(i) \quad = \quad \emptyset$$

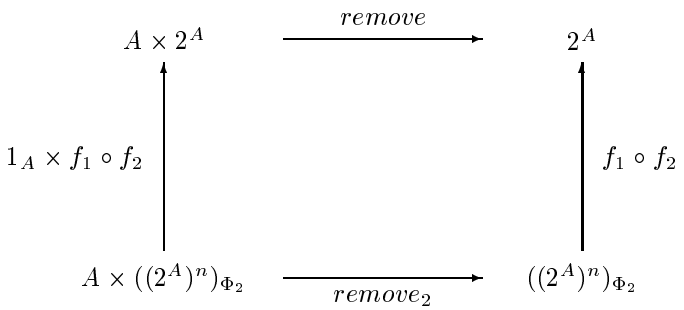
one straightforwardly draws

$$initialDb_2 \quad = \quad \left(\begin{array}{c} i \\ \emptyset \end{array} \right)_{i \in n}$$

We get the expected "for loop" ($i = 1, n$) initialization of the hash table setting every bucket to the empty set. Switching to $remove$, refinement equation is

$$f_1(f_2(remove_2(a,t))) = remove(a, f_1(f_2(t)))$$

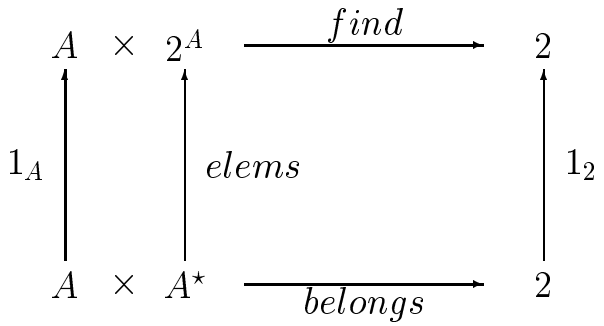
relative to refinement diagram:



Reasoning:

$$\begin{array}{rcl} f_1(f_2(remove_2(a,t))) & = & remove(a, f_1(f_2(t))) = f_1(f_2(t)) - \{a\} \\ & = & \bigcup_{i \in n} t(i) - \{a\} = (\bigcup_{i \in n} t(i)) \cap \overline{\{a\}} \\ & = & \bigcup_{i \in n} (t(i) \cap \overline{\{a\}}) = \bigcup_{i \in n} (t(i) - \{a\}) \\ & = & (t(H(a)) - \{a\}) \cup \bigcup_{i \in n - \{H(a)\}} (t(i) - \{a\}) = t'(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} \underbrace{t(i) - \{a\}}_{(*)} \end{array}$$

(to be continued in Remark 41)



where

$$find(a, x) \stackrel{\text{def}}{=} a \in x$$

We take

$$elems(x) \stackrel{\text{def}}{=} \begin{cases} x = <> & \Rightarrow \quad \emptyset \\ x \neq <> & \Rightarrow \quad \{head(x)\} \cup elems(tail(x)) \end{cases} \tag{272}$$

as the abstraction function.

By substitution — that is, unfolding of (272) in (271) — we get

$$belongs(x, y) = x \in \begin{cases} y = <> & \Rightarrow \quad \emptyset \\ y \neq <> & \Rightarrow \quad \{head(y)\} \cup elems(tail(y)) \end{cases}$$

which we re-write into

$$belongs(x, y) = \begin{cases} y = <> & \Rightarrow \quad x \in \emptyset \\ y \neq <> & \Rightarrow \quad x \in (\{head(y)\} \cup elems(tail(y))) \end{cases}$$

via (304). After some basic reductions we get

$$belongs(x, y) = \begin{cases} y = <> & \Rightarrow \quad FALSE \\ y \neq <> & \Rightarrow \quad x = head(y) \vee x \in elems(tail(y)) \end{cases} \tag{273}$$

Our target at this stage is to “delete” abstraction function $elems$ from (273).

Recalling the refinement equation,

$$belongs(x, y) = x \in elems(y)$$

let us equate it for y replaced by $tail(y)$:

$$belongs(x, tail(y)) = x \in elems(tail(y))$$

REMARK 41: **Hash-table operation synthesis: the $remove$ function (conclusion) and final comments**

(Continued from Remark 40)
Note that t' above is the one-entry mapping

$$t' \quad = \quad \left(\begin{array}{c} H(a) \\ t(H(a)) - \{a\} \end{array} \right)$$

At this point one resorts to invariant Φ_2 (248), from which we infer

$$\begin{array}{c} \forall i \in n : (\forall a \in t(i) : i = H(a)) \\ \updownarrow \\ \forall i \in n, a \in A : a \in t(i) \Rightarrow i = H(a) \\ \updownarrow \\ \forall i \in n, a \in A : i \neq H(a) \Rightarrow a \notin t(i) \\ \updownarrow \\ \forall i \in n, a \in A : i \neq H(a) \Rightarrow t(i) - \{a\} = t(i) \end{array}$$

which is enough to reduce $(*)$ in (271) above to $t(i)$. The reasoning here-upon is similar to that carried out for $insert$, leading to

$$f_1(f_2(remove_2(a,t))) \quad = \quad f_1(f_2(t \dagger \left(\begin{array}{c} H(a) \\ t(H(a)) - \{a\} \cup t(H(a)) \end{array} \right)))$$

i.e. (by removing $f_1 \circ f_2$ from both sides of the equality)

$$remove_2(a,t) \quad = \quad t \dagger \left(\begin{array}{c} H(a) \\ remove(a, t(H(a)) \end{array} \right)$$

Similarly to $insert_2$, it can be observed that $remove_2$ is nothing but $remove$ confined to the relevant collision bucket. Like $find_2$ but unlike $insert_2$, $remove_2$ resorts to the low-level ad hoc invariant in its refinement process.

Finally, it should be noted that the algebra of finite subsets of A, 2^A (221), is far too simple a model for database files. A model closer to reality is

$$Database' = A \hookrightarrow B \tag{274}$$

where A plays the rôle of a domain of keys and B is the data of interest.
Note that $2^A \cong_{dom} A \hookrightarrow 1$ (125). So it seems that the new model amounts to generalizing 1 in functor $\lambda X.X \hookrightarrow 1$ to any finite B. And indeed each operation σ over (221) can be generalized to some σ' over (274) provided that σ' "preserves σ along the dom morphism", that is, if σ is such that

$$dom(\sigma'(x, \ldots)) = \sigma(dom(x), \ldots)$$

With respect to data refinement, we know that functor $\lambda X.X \hookrightarrow B$ behaves well provided that it is applied to injective functions, see p.22.
Reference [Oli94] includes a thorough treatment of this extension and a generalization of the "hash effect" to arbitrary Set-functors which obey to the following property:

$$\mathcal{G}(n \times A) \quad \trianglelefteq \quad (\mathcal{G}(A))^n \tag{275}$$

for $n > 0$ and a given hash function $H : A \longrightarrow n$. This opens the hashing optimization to many other data-structures. However, for $G(X) = X^*$ we apparently have

$$(n \times A)^* \quad \cong \quad (A^*)^n \tag{276}$$

since both sides of (276) have the same cardinality ($\aleph_0$). Yet intuition may tell us that X^* is an "un-hashable" data-structure, for it seems impossible to reconstruct, from the n sequences on the right-hand side of (276), the original interleaving of such sequences. See [Oli94] for further discussion on this topic.

and move the lefthand side of this into (273):

$$belongs(x, y) = \begin{cases} y = <> & \Rightarrow & FALSE \\ y \neq <> & \Rightarrow & x = head(y) \vee belongs(x, tail(y)) \end{cases} \quad (277)$$

Abstraction function $elems$ has vanished! — so we have completed step 1 of the proposed method.

Next, we will be concerned with efficiency.

First of all, and for obvious reasons, in (277) we will prefer "lazy $\vee$" (309) to "eager $\vee$":

$$belongs(x, y) = \begin{cases} y = <> & \Rightarrow & FALSE \\ y \neq <> & \Rightarrow & \begin{cases} x = head(y) & \Rightarrow & TRUE \\ x \neq head(y) & \Rightarrow & belongs(x, tail(y)) \end{cases} \end{cases} \quad (278)$$

Second, and again for efficiency, we want to get rid of recursion in (278). Let $beloop$ be any function satisfying the following axiom:

$$beloop(x, y, b) = b \vee belongs(x, y) \quad (279)$$

It is easy to check the following instantiation of (279),

$$beloop(x, y, FALSE) = belongs(x, y) \quad (280)$$

since we know that algebraic structure $\langle \{TRUE, FALSE\}; \vee, FALSE \rangle$ is a monoid. Moreover, for $y = <>$, we have

$$\begin{array}{rcl} beloop(x, <>, b) & = & b \vee belongs(x, <>) \\ & = & b \vee FALSE \\ & = & b \end{array} \quad (281)$$

as well as, for $y \neq <>$,

$$\begin{array}{rcl} beloop(x, y, b) & = & b \vee (x = head(y) \vee belongs(x, tail(y))) \\ & = & (b \vee (x = head(y)) \vee belongs(x, tail(y)) \\ & = & beloop(x, tail(y), (b \vee (x = head(y)))) \end{array} \quad (282)$$

cf. (279). Putting (281) and (282) together, we obtain

$$\begin{array}{l} beloop(x,y,b) \stackrel{\text{def}}{=} \\ \quad \left\{ \begin{array}{lcl} y = <> & \Rightarrow & b \\ y \neq <> & \Rightarrow & beloop(x, tail(y), b \vee (x = head(y))) \end{array} \right. \end{array} \tag{283}$$

Furthermore, we know that $TRUE$ is the "zero" of $\vee$:

$$b \vee TRUE = TRUE \vee b = TRUE$$

So, via (279),

$$beloop(x, y, TRUE) = TRUE$$

i.e.

$$(b = TRUE) \Rightarrow beloop(x, y, b) = b$$

We take advantage of this in the follow up of (283) via (305):

$$\begin{array}{l} beloop(x,y,b) \stackrel{\text{def}}{=} \\ \quad \left\{ \begin{array}{lcl} y = <> & \Rightarrow & \left\{ \begin{array}{lcl} b & \Rightarrow & b \\ \neg b & \Rightarrow & b \end{array} \right. \\ y \neq <> & \Rightarrow & \left\{ \begin{array}{lcl} b & \Rightarrow & b \\ \neg b & \Rightarrow & beloop(x, tail(y), b \vee (x = head(y))) \end{array} \right. \end{array} \right. \end{array}$$

that is,

$$\begin{array}{l} beloop(x,y,b) \stackrel{\text{def}}{=} \\ \quad \left\{ \begin{array}{lcl} y = <> \wedge b & \Rightarrow & b \\ y = <> \wedge \neg b & \Rightarrow & b \\ y \neq <> \wedge b & \Rightarrow & b \\ y \neq <> \wedge \neg b & \Rightarrow & beloop(x, tail(y), b \vee (x = head(y))) \end{array} \right. \end{array}$$

or, at last,

$$\begin{array}{l} beloop(x,y,b) \stackrel{\text{def}}{=} \\ \quad \left\{ \begin{array}{lcl} y = <> \vee b & \Rightarrow & b \\ y \neq <> \wedge \neg b & \Rightarrow & beloop(x, tail(y), (x = head(y))) \end{array} \right. \end{array} \tag{284}$$

cf. (306).

Finally, we put (280) together with (284) to obtain the denotational semantics of, respectively, the initialization and body of the following `while`-loop,

```
{ bool  found = 0;
  list  p;
  {     p = y;
        while   ((p != <>) && ¬ found)
                { found = (x == head(p));
                  p = tail(p)
                };
  }
}
```
(285)

encoded here in a kind of 'ad hoc' imperative C-like syntax. (At this level one may prefer `found` to `b` for easier perception of the meaning of the loop.)

COMMENTS:

- Programming variables such as `found` are sometimes regarded as programming tricks produced by the intuition of able programmers. From the reasoning above we see that, rather than tricky, they have a sound mathematical basis, which is a consequence of the formal properties of the operators involved in their specification and calculation processes.
- The FOLD/UNFOLD CALCULUS encompasses other rules and schemata of this kind, useful to eliminate undesired recursion [BD77, Dar82].
- In a distributed/parallel environment, rules will go in the opposite way: the more recursive the better (*e.g.* double factorial instead of linear factorial).
- This apparently academic exercise can be generalized to a vast class of algorithm specifications (in a sense, every programming loop "has" a hidden mathematical structure of this kind), as is shown next.

Generalization

In the general case, let

$$\begin{array}{rcl} f & : & \ldots \times Y \times \ldots \longrightarrow M \\ f(_, y, _) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{rcl} p(_, y, _) & \Rightarrow & u \\ \neg(p(_, y, _)) & \Rightarrow & d(_, y, _)\ \theta\ f(_, e(y), _) \end{array} \right. \end{array} \tag{286}$$

be an arbitrary function of which we ignore the algorithmically irrelevant parameters and where the following abstract operators occur,

$$\begin{array}{rcl} p & : & \ldots \times Y \times \ldots \longrightarrow 2 \\ e & : & Y \longrightarrow Y \\ d & : & \ldots \times Y \times \ldots \longrightarrow M \\ \theta & : & M \times M \longrightarrow M \\ u & : & \longrightarrow M \end{array}$$

such that $\langle M; \theta, u \rangle$ is a monoid.

Patterns such as (286) are called *program schemata*. Schema (286) is called the *linear monadic schema*.

Wishing to generalize our calculation of $belongs$ to f, we introduce a function

$$floop(_, y, _, r) \stackrel{\text{def}}{=} r\ \theta\ f(_, y, _) \tag{287}$$

which, because $\langle M; \theta, u \rangle$ is a monoid, will satisfy:

$$f(_, y, _) = floop(_, y, _, u) \tag{288}$$

The following reasoning is a consequence of (287), of the substitution property, of rule (304) and of the monoidal properties of $\langle M; \theta, u \rangle$:

$$\begin{array}{rcl} floop(_, y, _, r) & = & r\ \theta\ f(_, y, _) \\ & = & r\ \theta \left\{ \begin{array}{rcl} p(_, y, _) & \Rightarrow & u \\ \neg(p(_, y, _)) & \Rightarrow & d(_, y, _)\ \theta\ f(_, e(y), _) \end{array} \right. \\ & = & \left\{ \begin{array}{rcl} p(_, y, _) & \Rightarrow & r\ \theta\ u \\ \neg p(_, y, _) & \Rightarrow & r\ \theta\ (d(_, y, _)\ \theta\ f(_, e(y), _)) \end{array} \right. \end{array}$$

$$= \left\{ \begin{array}{lcl} p(_, y, _) & \Rightarrow & r \\ \neg p(_, y, _) & \Rightarrow & \underbrace{(r\ \theta\ d(_, y, _))\ \theta\ f(_, e(y), _)}_{A} \end{array} \right.$$

By instantiation of (287) we get

$$A = floop(_, e(y), _, r\ \theta\ d(_, y, _))$$

In summary, we have

$$floop(_, y, _, r) = \left\{ \begin{array}{lcl} p(_, y, _) & \Rightarrow & r \\ \neg p(_, y, _) & \Rightarrow & floop(_, e(y), _, r\ \theta\ d(_, y, _)) \end{array} \right.$$

which, together with the "loop initialization" given by (288), matches the semantics of the following "`while`-loop schema",

```
{ M      r = u;
  Y      y' = y;
  while  (¬ p(_,y',_))
         { r = r θ d(_,y',_);
           y' = e(y')
         };
}
```

Should M have a "zero", that is some $a \in M$ such that

$$a\ \theta\ m = m\ \theta\ a = a$$

holds, further specialization of $floop$ becomes available:

$$\begin{array}{rcl} floop(_, y, _, r) & = & \left\{ \begin{array}{lcl} p(_, y, _) & \Rightarrow & r \\ \neg p(_, y, _) & \Rightarrow & \left\{ \begin{array}{lcl} r = a & \Rightarrow & a \\ r \neq a & \Rightarrow & floop(_, e(y), _, r\ \theta\ d(_, y, _)) \end{array} \right. \end{array} \right. \\ & = & \left\{ \begin{array}{lcl} p(_, y, _) \vee r = a & \Rightarrow & r \\ \neg p(_, y, _) \wedge r \neq a & \Rightarrow & floop(_, e(y), _, r\ \theta\ d(_, y, _)) \end{array} \right. \end{array}$$

leading to something we can identify as the semantics of the following

"`while`-loop schema",

```
{  M      r = u;
   Y      y' = y;
   while  (¬ p(_,y',_) && r!= a)
          { r = r θ d(_,y',_);
            y' = e(y')
          };
}
```

of which that of $belongs$ (285) is an obvious instance.

Examples 3.1 — Set and List Browsing

Recall Zermelo-Frænkl set-comprehension formula,

$$\{f(x) \mid x \in X \wedge p(x)\} \tag{289}$$

and list comprehension,

$$<f(x) \mid x \leftarrow L \wedge p(x)> \tag{290}$$

given $f : A \to B$, $X \subseteq A$, $L \in A^*$ and "filter" $p : A \to 2$.

It can be proved (*e.g.* by induction) that

- recursive function

$$\begin{array}{l} \varphi : 2^A \to 2^B \\ \quad \varphi(y) \stackrel{\text{def}}{=} \\ \quad\quad \left\{ \begin{array}{lcl} y = \emptyset & \Rightarrow & \emptyset \\ y \neq \emptyset & \Rightarrow & let \quad e \in y \\ & & \qquad a = \left\{ \begin{array}{lcl} p(e) & \Rightarrow & \{f(e)\} \\ \neg p(e) & \Rightarrow & \emptyset \end{array} \right. \\ & & in \quad a \cup \varphi(y - \{e\}) \end{array} \right. \end{array}$$

  when applied to X, yields the same set as ZF-formula above;

- recursive function

$$\phi(y) \stackrel{\text{def}}{=} \begin{cases} y = <> & \Rightarrow & <> \\ y \neq <> & \Rightarrow & let \quad x = head(()y) \\ & & \quad a = \begin{cases} p(x) & \Rightarrow & <f(x)> \\ \neg(p(x)) & \Rightarrow & <> \end{cases} \\ & & in \quad a \frown \phi(tail(y)) \end{cases}$$

  when applied to L, yields the same list as formula (290) above.

Moreover, both recursive functions above are instances of linear-monadic scheme (286).

So. . . they “are” loops!

Simple instantiation of (286) will show that ϕ, for instance, is realized by the following abstract “`while`-loop”,

```
{  B* r = <>;
   A* y' = y;
   A x;
   while (y' != <>)
       { x = head(y');
         y'= tail(y');
         if p(x) r = conc(r,f(x));
       }
}
```

or, being more “C-oriented” in parameterizing the function in terms of `stdin` and `stdout` and assuming suitable ‘i/o’ library functions `getA` and `putB`, by

```
{
   A x;
   while ((x = getA(stdin)) != EOF)
         { if p(x) putB(f(x)); }
}
```

Of course, a similar `while`-loop schema can be developed for (289).

Note how “fine-grained” these schemata look like when compared to relational model information browsing, and how they render “one-at-a-time” data browsing explicit.

COMMENTS:

- A lot of programming practice can be captured by schema results such as above.
- Experience in this kind of calculation develops the ability to spot while loops and to write them straight away — one just has to search for the closest linear monadic schema around...
- Recursive schemata which are not linear monadic may actually be converted into that form by program calculation. Our final example belongs to this category of operation refinement.

Example 4 — Calculation of PPD “Explode” or How to Deal with Complex Operations

One of the most interesting operations which can be defined over the PPD data model specification of p.52 is that which computes the *part explosion* of one equipment according to its production tree:

$$
\begin{array}{rcl}
explode & : & \#Equip \times PPD \longrightarrow (\#Comp \hookrightarrow \mathbf{N}) \\
explode(\#e, \sigma) & \stackrel{\text{def}}{=} & let \quad \tau = apply(\sigma, \#e) \\
 & & in \quad \oplus_{x \in dom(\tau)} \left\{ \begin{array}{lcl} x = i_1(k) & \Rightarrow & \left(\begin{array}{c} k \\ \tau(x) \end{array} \right) \\ x = i_2(k) & \Rightarrow & \tau(x) \otimes explode(k, \sigma) \end{array} \right.
\end{array}
$$

where

- $\oplus$ is multiset ("bag") union:

$$\begin{array}{rcl} x \oplus y & \stackrel{\text{def}}{=} & x \setminus dom(y) \\ & \cup & y \setminus dom(x) \\ & \cup & \left(\begin{array}{c} a \\ x(a) + y(a) \end{array} \right)_{a \in dom(x) \cap dom(y)} \end{array} \tag{291}$$

- $\otimes$ is called multiset external multiplication (by analogy with vector spaces, see Exercise 23):

$$\begin{array}{rcl} \otimes & : & \mathbf{N} \times (A \hookrightarrow \mathbf{N}) \longrightarrow (A \hookrightarrow \mathbf{N}) \\ n \otimes \sigma & \stackrel{\text{def}}{=} & A \hookrightarrow (\lambda a.n \times a)(\sigma) \end{array} \tag{292}$$

- $apply$ is the following auxiliary function:

$$\begin{array}{rcl} apply & : & PPD \times \#Equip \longrightarrow (\#Unid \hookrightarrow \mathbf{N}) \\ apply(\tau, k) & \stackrel{\text{def}}{=} & let \quad \sigma = \pi_3(\tau) \\ & & in \quad \left\{ \begin{array}{rcl} k \in dom(\sigma) & \Rightarrow & \sigma(k) \\ k \notin dom(\sigma) & \Rightarrow & (\ ) \end{array} \right. \end{array}$$

Eager to implement $explode$ at $RPPD$-level, suppose that from its refinement diagram we have already arrived at

$$\begin{array}{rcl} explode_R & : & \#Equip \times RPPD \longrightarrow (\#Comp \hookrightarrow \mathbf{N}) \\ explode_R(\#e, \gamma) & \stackrel{\text{def}}{=} & let \quad \tau = apply_R(\gamma, \#e) \\ & & in \quad \oplus_{x \in dom(\tau)} \left\{ \begin{array}{rcl} x = i_1(k) & \Rightarrow & \left(\begin{array}{c} k \\ \tau(x) \end{array} \right) \\ x = i_2(k) & \Rightarrow & \tau(x) \otimes explode_R(k, \gamma) \end{array} \right. \end{array} \tag{293}$$

where γ is a 4-tuple $\langle \gamma_1, \gamma_2, \gamma_3, \gamma_4 \rangle$, recall p.54 and see Exercise 25.

Therefore, only PPD was refined, *cf.* "state-oriented" data refinement.

QUESTION

How do we implement $explode_R$? How do we implement recursion on top the the underlying relational model?

CALCULATION

Our first move is to realize that

$$explode_R(\#e, \gamma)) \;=\; aux(apply_R(\gamma, \#e), \gamma) \tag{294}$$

holds, where

$$aux(\tau, \gamma) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcll} \tau = (\;) & \Rightarrow & (\;) & \\ \tau \neq (\;) & \Rightarrow & let & x \in dom(\tau) \\ & & & y = \left\{ \begin{array}{lcl} x = i_1(k) & \Rightarrow & \begin{pmatrix} k \\ \tau(x) \end{pmatrix} \\ x = i_2(k) & \Rightarrow & \tau(x) \otimes explode_R(k, \gamma) \end{array} \right. \\ & & in & y \oplus aux(\tau \setminus \{x\}, \gamma) \end{array} \right.$$

is the linear monadic version of $\oplus_{x \in \ldots} \ldots$ — see (297) and (298) in a remark to follow, for $\varphi = \oplus$.

However, because of the occurrence of $explode_R$ in the schema, we are still far away from iteration.

It is easy to check that the following properties

$$n \otimes aux(\tau, \gamma) \;=\; aux(n \otimes \tau, \gamma) \tag{295}$$

$$aux(\tau_1, \gamma) \oplus aux(\tau_2, \gamma) \;=\; aux(\tau_1 \oplus \tau_2, \gamma) \tag{296}$$

follow from the underlying multiset algebra (see Exercise 23).

Then, from (294) and (295) we draw

$$\begin{array}{rcl} \tau(x) \otimes explode_R(k, \gamma)) & = & \tau(x) \otimes aux(apply_R(\gamma, k), \gamma) \\ & = & aux(\tau(x) \otimes apply_R(\gamma, k), \gamma) \end{array}$$

and so we may re-write aux's body into:

$$\left\{ \begin{array}{lcl} x = i_1(k) & \Rightarrow & \begin{pmatrix} k \\ \tau(x) \end{pmatrix} \oplus aux(\tau \setminus \{x\}, \gamma) \\ x = i_2(k) & \Rightarrow & aux(\tau(x) \otimes apply_R(\gamma, k), \gamma) \oplus aux(\tau \setminus \{x\}, \gamma) \end{array} \right.$$

Now apply (296) to what we have just obtained:

$$\left\{\begin{array}{lcl} x = i_1(k) & \Rightarrow & \left(\begin{array}{c} k \\ \tau(x) \end{array}\right) \oplus aux(\tau \setminus \{x\}, \gamma) \\ x = i_2(k) & \Rightarrow & aux(\tau(x) \otimes apply_R(\gamma, k) \oplus \tau \setminus \{x\}, \gamma) \end{array}\right.$$

which leads to

$$\left\{\begin{array}{lcl} x = i_1(k) & \Rightarrow & \left(\begin{array}{c} k \\ \tau(x) \end{array}\right) \oplus \\ x = i_2(k) & \Rightarrow & (\ ) \end{array}\right.$$
$$\underbrace{\left\{\begin{array}{lcl} x = i_1(k) & \Rightarrow & aux(\tau \setminus \{x\}, \gamma) \\ x = i_2(k) & \Rightarrow & aux(\tau(x) \otimes apply_R(\gamma, k) \oplus \tau \setminus \{x\}, \gamma) \end{array}\right.}_{X}$$

simply because $(\ )$ is the unit of $\oplus$ and according to the following rule extending (304):

$$f(\left\{\begin{array}{lcl} p & \Rightarrow & q \\ \neg(p) & \Rightarrow & r \end{array}\right., \left\{\begin{array}{lcl} p & \Rightarrow & s \\ \neg(p) & \Rightarrow & t \end{array}\right.) = \left\{\begin{array}{lcl} p & \Rightarrow & f(q, s) \\ \neg(p) & \Rightarrow & f(r, t) \end{array}\right.$$

Now, X re-writes into

$$aux(\tau \setminus \{x\} \oplus \left\{\begin{array}{lcl} x = i_1(k) & \Rightarrow & (\ ) \\ x = i_2(k) & \Rightarrow & \tau(x) \otimes apply_R(\gamma, k) \end{array}\right., \gamma)$$

exploiting the unit and commutativity of $\oplus$, and via (304). Eventually, we find that aux instantiates the linear monadic schema (286) as follows:

$f(_, y, _)$	$aux(y, \gamma)$
$p(_, y, _)$	$y = (\ )$
u	$(\ )$
$d(_, y, _)$	$\left\{\begin{array}{lcl} x = i_1(k) & \Rightarrow & \left(\begin{array}{c} k \\ y(x) \end{array}\right) \\ x = i_2(k) & \Rightarrow & (\ ) \end{array}\right.$ /*for $x \in dom(y)$ */
θ	$\oplus$
$e(y)$	$\left\{\begin{array}{lcl} x = i_1(k) & \Rightarrow & (\ ) \\ x = i_2(k) & \Rightarrow & y(x) \otimes apply_R(\gamma, k) \end{array}\right. \oplus y \setminus \{x\}$

Because we don't find a zero relative to $\oplus$ in finite $\#Comp \hookrightarrow \mathbb{N}$, we stop at this "pseudo-C" formulation of our target `while`-loop,

```
{ Parts     r = ();
  Structure y' = y;
  while     (y' != ())
            { Unit x = choice(dom(y'));
              Quantity n = y'(x);
              r = r ⊕ { x = i1(k) ⇒ ( k )
                                   ( n )   ;
                       x = i2(k) ⇒ ( )
              y' = y' \{x} ⊕ { x = i1(k) ⇒ ( )
                             { x = i2(k) ⇒ n ⊗ apply_R(γ, k)   ;
            };
}
```

where data-types `Parts` and `Structure` correspond to $\#Comp \hookrightarrow \mathbb{N}$ and $\#Unit \hookrightarrow \mathbb{N}$, respectively.

Check that in the above step we took advantage from the unit of $\oplus$ and from the formal semantics of the imperative `if ...then ...else` construct.

Putting things together — recall (294) — we (finally!) obtain the following loop realizing $explode_R(k, \gamma)$:

```
{ Parts r = ();
  Structure ff = apply_R(γ, k);
  Quantity n ;
  Unit x;
  while (ff != ())
     { x = choice(dom(ff)) ;
       n = ff(x);
       ff = ff\{x};
       if x == (1,k) r = r ⊕ ( k
                               n ) ;
       if x == (2,k) ff = ff ⊕ (n ⊗ apply_R(γ,k));

     }
}
```

A full relational implementation will require a relational realization of `Structure` and operations $apply_R$, $\oplus$ and $\otimes$.

COMMENTS:

- This result can be generalized and converted into a program schema.
- Such a schema encompasses other (apparently recursive) algorithms which "run on top of" monoidal structures and thus become `while`-loops (*e.g.* transitive closure of an acyclic graph [Oli96]).
- Acyclicness can be lifted leading to a more general schema provided that the original specification takes that into account.

REMARK 42: **Monoids as a basis for recursion introduction and elimination**

Let $(Y\,;\theta,\epsilon)$ be an Abelian monoid, that is, θ is associative and commutative and ϵ is the unit of θ. Then it will make sense to write

$$\Theta_{e\in x}\varphi(e) \tag{297}$$

a construction which obeys to the following properties:

$$\begin{array}{lcl}\Theta_{e\in\emptyset}\varphi(e) & = & \epsilon \\ \Theta_{e\in x}\varphi(e) & = & \varphi(e_0)\theta\Theta_{e\in x-\{e_0\}}\varphi(e) \qquad \text{for } e_0\in x\end{array}$$

These properties lead to the following recursive schema,

$$\begin{array}{rcl} f_\varphi : 2^X & \rightarrow & Y \\ f_\varphi(x) & \stackrel{\text{def}}{=} & \left\{\begin{array}{lcl} x=\emptyset & \Rightarrow & \epsilon \\ x\neq\emptyset & \Rightarrow & let \quad e\in x \\ & & in \quad \varphi(e)\theta f_\varphi(x-\{e\}) \end{array}\right. \end{array} \tag{298}$$

where $\varphi : X \rightarrow Y$. This is called the θ-reduction algorithmic scheme and it can be regarded as a recursive implementation of (297). But, because it is an instance of the linear monadic scheme (286), recursion can be eliminated and thus an iterative implementation of the schema be obtained.

Exercise 23 A multiset (vulg. "bag") is an "intermediate" notion between sets (2^A) and sequences (A^*). In a multiset, a given element $a\in A$ may occur more than once, its number of occurrences being called its *multiplicity*:

$$MSet(A) \quad \cong \quad A \hookrightarrow \mathbb{N} \tag{299}$$

Therefore, the empty mapping $\left(\quad\right)$ represents the empty multiset and set-union generalizes to multiplicity addition, that is, given two multisets x and y in $MSet(A)$, their union, denoted by $x\oplus y$, is the mapping given by

$$\begin{array}{rcl} x\oplus y & \stackrel{\text{def}}{=} & x\setminus dom(y) \\ & \cup & y\setminus dom(x) \\ & \cup & \left(\begin{array}{c} a \\ x(a)+y(a) \end{array}\right)_{a\in dom(x)\cap dom(y)} \end{array} \tag{300}$$

1. Show that $\oplus$ can be alternatively defined by:

$$x\oplus y \quad \stackrel{\text{def}}{=} \quad x\dagger y\dagger\left(\begin{array}{c} a \\ x(a)+y(a) \end{array}\right)_{a\in dom(x|(dom(y)))} \tag{301}$$

2. Show that $Mset(A)$ together with $\oplus$ and the empty mapping form an Abelian (commutative) monoid.
3. Show that multiset external multiplication (292) obeys the following properties

$$n\otimes(m\otimes\sigma) \quad = \quad (n\times m)\otimes\sigma \tag{302}$$

$$n\otimes(\sigma\oplus\tau) \quad = \quad (n\otimes\sigma)\oplus(n\otimes\tau) \tag{303}$$

□

Exercise 24 Build f_φ (298) and its iterative transformation for the following instances of θ (guess the monoid structure in each case): (1) logical conjunction, (2) logical disjunction, (3) set union, (4) addition in $\mathbb{N}_0$, (5) multiplication in $\mathbb{N}$ and (6) multiset union ($\oplus$).

□

Exercise 25 Recalling the refinement of PPD into $RPPD$ (p.54), calculate $apply_R$ (293).

□

REMARK 43: **A quick reference guide to program transformation**

The FOLD/UNFOLD calculus arises from the λ-calculus and the theory of recursive functions [Man74]. The reader is referred to references [BD77, Dar82, Dar84] for seminal work in the area. The idea of an **algebra of programs** supporting program reasoning was pioneered by Backus [Bac78].

We start with a few properties involving conditionals.

Function application distributes *w.r.t.* **conditional**

$$f(x, \left\{ \begin{array}{rcl} p & \Rightarrow & q \\ \neg p & \Rightarrow & r \end{array} \right.) \quad = \quad \left\{ \begin{array}{rcl} p & \Rightarrow & f(x,q) \\ \neg p & \Rightarrow & f(x,r) \end{array} \right. \tag{304}$$

Other rules concerning conditionals

$$q \quad = \quad \left\{ \begin{array}{rcl} p & \Rightarrow & q \\ \neg p & \Rightarrow & q \end{array} \right. \tag{305}$$

$$\left\{ \begin{array}{rcl} p & \Rightarrow & q \\ r & \Rightarrow & q \\ t & \Rightarrow & s \end{array} \right. \quad = \quad \left\{ \begin{array}{rcl} (p \vee r) & \Rightarrow & q \\ t & \Rightarrow & s \end{array} \right. \tag{306}$$

$$\left\{ \begin{array}{rcl} (\left\{ \begin{array}{rcl} p & \Rightarrow & q \\ \neg(p) & \Rightarrow & r \end{array} \right.) & \Rightarrow & s \\ \neg((\left\{ \begin{array}{rcl} p & \Rightarrow & q \\ \neg(p) & \Rightarrow & r \end{array} \right.)) & \Rightarrow & t \end{array} \right. \quad = \quad \left\{ \begin{array}{rcl} p & \Rightarrow & \left\{ \begin{array}{rcl} q & \Rightarrow & s \\ \neg(q) & \Rightarrow & t \end{array} \right. \\ \neg(p) & \Rightarrow & \left\{ \begin{array}{rcl} r & \Rightarrow & s \\ \neg(r) & \Rightarrow & t \end{array} \right. \end{array} \right. \tag{307}$$

$$\left\{ \begin{array}{rcl} p & \Rightarrow & \left\{ \begin{array}{rcl} q & \Rightarrow & a \\ \neg(q) & \Rightarrow & b \end{array} \right. \\ \neg(p) & \Rightarrow & x \end{array} \right. \quad = \quad \left\{ \begin{array}{rcl} (p \wedge q) & \Rightarrow & a \\ (p \wedge \neg q) & \Rightarrow & b \\ \neg p & \Rightarrow & x \end{array} \right. \tag{308}$$

Lazy disjunction and conjunction

$$\begin{array}{rcl} p \vee q & \leq & \left\{ \begin{array}{rcl} p & \Rightarrow & TRUE \\ \neg p & \Rightarrow & q \end{array} \right. \\ p \wedge q & \leq & \left\{ \begin{array}{rcl} p & \Rightarrow & q \\ \neg p & \Rightarrow & FALSE \end{array} \right. \end{array} \tag{309}$$

where $f \leq g$ means that g is more defined that f and so g can always replace f in any expression ϕ where f occurs:

$$\phi(f(x)) \leq \phi(g(x))$$

'Fold/Unfold' rules

Let f be an arbitrary function, for instance,

$$f(x) \stackrel{\text{def}}{=} \left\{ \begin{array}{rcl} p(x) & \Rightarrow & g(x) \\ \neg(p(x)) & \Rightarrow & h(f(e(x))) \end{array} \right.$$

(to be continued in Remark 44.)

REMARK 44: **A quick reference guide to program transformation (conclusion)**

Expression $f(a)$ means the application of f to a particular and valid argument a. Wherever $f(a)$ is a subexpression of a larger expression

$$\phi(f(a))$$

the "**unfold** f"-transformation means to replace f's invocation by f's body (suitably instantiated), that is, $\phi(f(a))$ re-writes to

$$\phi(\left\{\begin{array}{lcl} p(a) & \Rightarrow & g(a) \\ \neg(p(a)) & \Rightarrow & h(f(e(a))) \end{array}\right.)$$

which will lead to

$$\left\{\begin{array}{lcl} p(a) & \Rightarrow & \phi(g(a)) \\ \neg(p(a)) & \Rightarrow & \phi(h(f(e(a)))) \end{array}\right. \tag{310}$$

by application of law (304).
Unfolding thus means to read

$$\phi(f(x)) = \phi(\left\{\begin{array}{lcl} p(x) & \Rightarrow & g(x) \\ \neg(p(x)) & \Rightarrow & h(f(e(x))) \end{array}\right.)$$

from left to right. The opposite transformation, which from right to left makes it possible to re-write (310) into $\phi(f(a))$, is called the "**fold** f"-transformation of an expression. So the unfold-transformation expands an expression where a function occurs while the opposite fold-transformation contracts such an expression by replacing the instantiation of the body of a function by the relevant invocation.

Exercise 26 Recall the factorial function,

$$y! = \left\{\begin{array}{lcl} y = 0 & \Rightarrow & 1 \\ \neg(y = 0) & \Rightarrow & y \times (y - 1)! \end{array}\right.$$

Frame it into the linear monadic schema (286) and calculate its iterative, `while`-loop version.
□

Pragmatics of SETS: Impact on Relational Database Design

- The SETS calculus arose from [Oli87].

  It has been regularly taught in Minho's computer science degrees since 1989, with an increasing degree of formalization.

- Rodrigues [Rod93] work on rephrasing relational database theory in the SETS context has been determinant in the way **database design** is nowadays lectured to Minho's undergraduate students who follow the formal methods course — normalized data models are obtained from formal specifications by SETS calculation rather than using the standard technology.

- Final year projects usually involve the full calculation of an SQL information server, realized on top of MINISQL or ORACLE.

- By far, the most comprehensive account of the calculus is an as yet unpublished textbook, currently in its third Portuguese version [Oli96]. It should become available in English next year.

- Industrial application — many students have reported on the successful application of the calculus at industrial software design level.

  They use the ER-diagram SETS semantics described in [OC93] to "export" their results to the traditional media so that their colleagues in the design team can understand what is happening to the project.

- A list of industrial projects which have resorted to program calculation in SETS at some stage is being compiled at the moment (it will soon become available at the SETS's URL).

- SETS was used in the design of an information system stepwise temporalization strategy for a Portuguese company [Gro96], and in the calculation of the final implementation of the SOUR (EU 379) system.

Current Research on SETS

A list of topics which are under current investigation in the SETS' development group follows — see SETS's home page

`http://www.di.uminho.pt/~jno/html/setshp.html`

for knowing the people, further documentation and on-line information:

- Moving from static data to processes — development of a SETS-based Process Refinement calculus (on-going Ph.D. thesis). This involves:
  - changing the underlying category;
  - developing a functorial approach to process algebras — we have already some results concerning LTSs ("labelled transition systems").
- Integration of SETS with other refinement calculi (on-going Ph.D. thesis).
- Development of a genetic-algorithm-geared SETS animator (recently submitted M.Sc. thesis).
- Development of a SETS-based approach to OLAP technology (on-going M.Sc. project).

Work scheduled for future research:

- Integration of SETS in RAISE.
- Integration of SETS with the Oxford Refinement Calculus (based on comparative work).
- Development of a set of tools supporting SETS-based reverse-engineering — a consortium is being set up in this area for the preparation of a R&D proposal (everybody is welcome).
- Eventually, we want to be able to move into distributed and real-time system software calculation.

Final Remark

Wanting a good book on category theory? My vote goes for [McL85].

References

[ABNO97] J.J. Almeida, L.S. Barbosa, F.L. Neves, and J.N. Oliveira. Camila: Formal software engineering supported by functional programming. 1997. Submitted to DPLE'97: First International Conference on Declarative Programming Languages in Education, Southampton, Sep. 3-5 1997.

[AKM81] M. A. Arbib, A. Kfoury, and R. N. Mool. *A Basis for Theoretical Computer Science*. Texts and Monographs in Computer Science. Springer-Verlag, 1981. D. Gries (Ed.).

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–639, August 1978.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.

[Dar82] J. Darlington. Program transformation. In *Funct. Prog. and Its Applications: An Advanced Course*. Cambridge Univ. Press, 1982.

[Dar84] J. Darlington. *The Design of Efficient Data Representations*, chapter 7, pages 139–156. Macmillan, London, 1984.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Fie80] E. Fielding. The specification of abstract mappings and their implementation as B^+-trees. Technical Report PRG-18, Oxford University, September 1980.

[Gof84] J. Le Goff. *Calendário*, volume I, chapter 8, pages 260–292. I.N.-C.M., 1984. EINAUDI Encyclopedia (Portuguese translation).

[Gro96] INESC 2361 Group. Temporalização da NBDC. Technical report, © SONAE Company, Maia, May 1996. Consultancy report (in Portuguese).

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Software Engineering Series. Pitman, 1978. E. Horowitz (Ed.).

[Jon80] C. B. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, 1980. C. A. R. Hoare.

[Jon86] C. B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986. C. A. R. Hoare.

[Jou92] I. S. Jourdan. Reificação de tipos abstractos de dados: Uma abordagem matemática. Master's thesis, University of Coimbra, 1992. (In Portuguese).

[Knu74] D. Knuth. Structured programming with goto statements. *ACM Comp. Surveys*, 6(4), Dec. 1974.

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

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, New-York, 1974.

[McL85] C. McLarty. *Elementary Categories, Elementary Toposes*. Oxford Logic Guides n.º 21. Calendron Press, Oxford, 1^{st} edition, 1985.

[MG90] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[Mor90] C. Morgan. *Programming from Specification*. Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.

[OC93] J. N. Oliveira and A. M. Cruz. Formal calculi applied to software component knowledge elicitation. Technical Report C19-WP2D, DI/INESC, December 1993. IMI Contract *C.1.9. Sviluppo di Metodologie, Sistemi e Servizi Innovativi in Rete*.

[Oli87] J. N. Oliveira. Refinamento transformacional de especificações (terminais). In *Actas das XII "Jornadas Luso-Espanholas de Matemática"*, volume II, pages 412–417, Braga, Portugal, 4-8 Maio 1987.

[Oli90] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, April 1990.

[Oli92] J. N. Oliveira. Software reification using the SETS calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).

[Oli94] J. N. Oliveira. Hash tables — a case study in $\trianglelefteq$-calculation. Technical Report DI/INESC 94-12-1, DI/INESC, U. Minho, December 1994.

[Oli96] J. N. Oliveira. *Métodos Formais de Programação*. Univ. of Minho, 3^{rd} edition, 1996. Textbook in preparation. Postscript of current version (427pages in Portuguese) available from `http://www.di.uminho.pt/~jno/ps/mfp.ps.tar.gz`.

[Rod93] C. J. Rodrigues. Sobre o desenvolvimento formal de bases de dados. Master's thesis, University of Minho, 1993. (In Portuguese).

[Spi89] J. M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989. C. A. R. Hoare.

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

[Wan79] M. Wand. Final algebraic semantics and data type extensions. *JCSS*, 19:27–44, 1979.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[Wol88] M. Wolczko. Semantics of object-oriented languages. Technical Report UMCS-88-6-1, U. Manchester, 1988.

[Wol90] M. Wolczko. Garbage collection. In *Case Studies in Systematic Software Development*, pages 211–233. Prentice-Hall International, 1990. Shaw R.C. and Jones C.B. (Ed.).

[Woo92] J. C. P. Woodcock. Implementing promoted operations in Z. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 367–378. Springer-Verlag, 8–10 January 1992. (Invited paper).