A Data Structuring Calculus
and
Its Application to Program Development

Universidad de San Luis — Program Calculation Course
27 April–8 May 1998
San Luis
Argentina

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

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

Abstract

- This course is concerned with **program design** by formal **calculation**. A particular calculus (SETS) will be taught which enables the derivation of implementations (programs) from *model-oriented* formal specifications.
- The proposed calculation style is based on **set-theory** and a modest use of **category theory**. At its core, the calculus embodies a collection of explicit design rules which enable transformational data structure **refinement** (reification).
- The calculus blends nicely with other program calculi such as the **Bird-Moor**, the **Fold/Unfold** and **Morgan**'s calculi.

  One of its main features is that *abstraction invariants* "à la Morgan" can be synthesized by simple calculations.
- The calculus is also useful in assessing, comparing, re-using or classifying specifications.
- The main results of the course are concerned with a **functorial** approach to datatype **specification** and **reification**, particularly *wrt.* the systematic implementation of **recursive** data domains on non-recursive (*e.g.* database) run-time environments.
- Examples will be given in the area of **information system** calculation.

  The calculus' potential to replace "old" technologies such as *e.g.* E-R diagram-based **database design** will be described.

Introduction

Software Design — “Riddle Guessing” or Problem Solving?

Faced with a “folk riddle” such as

> *A football team scores 16 points after losing no match at all and drawing twice as many matches as the ones it won. In which week of the championship are we?*

this “72 year old” uneducated man can but guess an answer by trial-and-error,

won	$drawn$	$week$	$score$	$ok?$
1	2	3	4	no
2	4	6	8	no
3	6	9	12	no
4	8	12	16	$yes!$

while this “12 year old” maths student provides an answer,

$$week = 12$$

after solving the following mathematical model of the riddle,

$$\left\{ \begin{array}{rcl} won + drawn & = & week \\ drawn + 2 \times won & = & 16 \\ drawn & = & 2 \times won \end{array} \right.$$

for unknown $week$.

Will the average “36 year old” software designer address the following “riddle”,

> *A bill is a statement of charges for services rendered. A company needs a hierarchical bill manager associating not only charges to service items but also able to unfold such services into sub-bills and so on. Can this hierarchical bill organizer be provided in* MICROSOFT ACCESS*?*

by building the following mathematical model of it,

$$\left\{ \begin{array}{rcl} bill & = & service \rightharpoonup info \\ info & = & charge + subbill \\ subbill & = & bill \\ charge & = & I\!N \end{array} \right.$$

and solving it for variable $bill$?

Will minimal solution

$$bill \;=\; ((\#bill \times service) \rightharpoonup charge) \times ((\#bill \times service) \rightharpoonup \#bill) \times 2^{\#bill}$$

that is, 3 flat tables

$$\begin{array}{|c|c||c|} \hline \#bill & service & charge \\ \hline \cdots & \cdots & \cdots \end{array} + \begin{array}{|c|c||c|} \hline \#bill & service & \#bill \\ \hline \cdots & \cdots & \cdots \end{array} + \begin{array}{|c|} \hline \#bill \\ \hline \cdots \end{array}$$

be reached, subject to a (non-trivial) datatype invariant?

No! the average programmer will probably behave like the 72 year-old person above, in:

- guessing an 'ad hoc' data model (*e.g.* ignoring the data type invariant) and
- debugging his/her code in case something goes wrong (*e.g.* the $bill$ browser doesn't terminate for some of its operations).

Comment

In many teaching contexts software design teaching ignores the **general problem solving** strategy which one is taught at school:

- **understand** your problem
- build a **mathematical** model of it
- **reason** in such a model
- **improve**/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 linear equations, at **secondary** school, then to
- differential/integral equations, in standard calculus, and to
- systems of (recursive) equations on domain spaces in **software calculi**.

Can this be accepted in an University setting?

No, by any means!

Why Formal Methods?

- **Software design** remains among the very few areas in which many professionals develop their activity with little (if any) **scientific** support.
- Designing software with no **mathematical** support cannot be accepted but as an insult to current **engineering standards**.
- Posterity will regard currently popular technologies as typical of a kind of “dark age”, or “medieval” stage of the software industry.

Why Formal Calculi?

- Programming notation can be expressed by *“**formulæ** and **equations** (...) which share the **elegance** of those which underlie physics and chemistry or any other branch of basic science”* (C.Hoare).

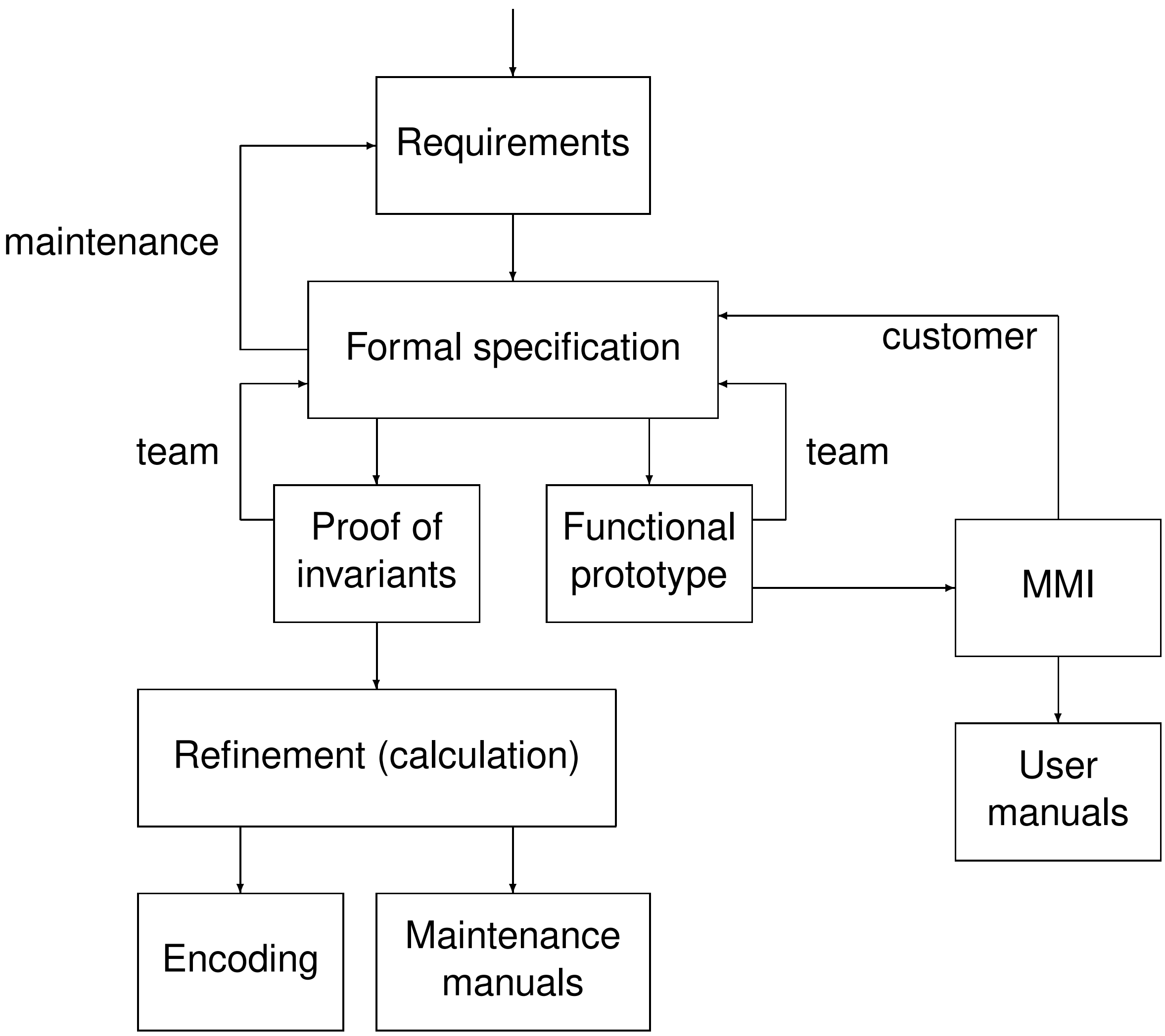
- Mathematical **program calculation** is possible and (we claim) can be made **practical**.
- Eventually, program design will be by-and-large **automated**.

Back in History ...

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." (He who knows by Algebra, he has a scientific mind.) *Pedro Nunes (1502-1578) in libro de algebra, 1567, fol 270v*.

A Formal Software Development Life-cycle



Comments:

- The specification language taught in this course can be “animated” in the CAMILA Toolset [ABNO97b, ABNO97c], a **rapid prototyping** environment aiming at the promotion of *Formal Methods* in industry and academia.
- **Incremental** data refinement by calculation is supported by the prototype / implementation hybridization primitives are available in the Toolset [ABNO97a].
- The CAMILA/SETS approach to formal software development 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`
- CAMILA Home Page:

  `http://camila.di.uminho.pt`

Program Calculation: An Overview

Context

The context of program calculation as understood in this course is **model**-oriented (or **constructive**) specification such as in the VDM [Jon80, Jon86, FL97] and Z [Spi89] traditions.

About the Past ...

$$\text{Validation} \begin{cases} \text{Debug} \\ \text{Verification} \begin{cases} \text{Static (`a posteriori')} \\ \text{Constructive (\textit{invent \& verify})} \end{cases} \end{cases}$$

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 inability 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.

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

- Reification Calculi
  - Static Semantics
    - Algorithmic Refinement
      - ⋮
      - Fold/Unfold [Dar84]
      - Morgan [Mor90]
      - Bird-Moor [BdM97]
      - ⋮
    - Data Refinement
      - ⋮
      - SETS
      - ⋮
  - Dynamic Semantics
    - ⋮
    - Action refinement [AH88]
    - CCS [Mil89]
    - ⋮

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:

$$\begin{array}{cc} \text{What} & \text{(specification)} \\ \uparrow & \\ \boxed{\text{Why}} & \textit{(justification)} \\ \uparrow & \\ \text{How} & \text{(implementation)} \end{array}$$

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 which are already available in textbooks such as [Mor90] and [BdM97].

Reify = make real; materialize some intended thing.

Starting Point

Standard data refinement assumes (*cf.* [Mor90]):

- 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* $a \in A$

Thus we have

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

where

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

Functional Data Refinement in this Course

Instead of simply choosing AF and CI ...

$\boxed{\Rightarrow \ \ldots \text{ we want to synthesize them by calculation.}}$

What is a Model?

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 VDM-SL [FL97] fragment for a very simple bank accounting system,

$$
\begin{array}{l}
AMS = \textsf{map } AccId \textsf{ to } Status \\
\textsf{inv } ams == \textsf{forall } n \textsf{ in set } dom\ ams\ \&\ ams(n).H \neq \emptyset; \\
\\
Status :: \quad H : \textsf{set of } AccHolder \\
\qquad\qquad\quad A : Amount; \\
Amount = nat;
\end{array}
\tag{1}
$$

where the AMS data-type invariant enforces that every account has, at least, one account-holder.

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

$$
\begin{array}{ll}
AMS1 :: \quad HT : \textsf{set of } Row1 & \textit{/*table of account-holders */} \\
\qquad\qquad\ AT : \textsf{set of } Row2 & \textit{/*table of amounts */} \\
\textsf{inv mk\_}AMS1(ht, at) == get_1(at) = get_1(ht) \wedge depKA(at); & \\
& \\
Row1 :: K : AccId \ \ H : AccHolder; & \\
Row2 :: K : AccId \ \ A : Amount &
\end{array}
$$

where, in the $AMS1$ data-type invariant,

$$
\begin{array}{l}
get_1 : \textsf{set of } (A \ \ B) \longrightarrow \textsf{set of } A \\
get_1(\rho) == \{a \mid \ a : A\ \&\ \textsf{exists } b : B\ \&\ (a, b) \in \rho\}
\end{array}
$$

is a generic "get first column" relational-operator, and predicate

$$
\begin{array}{l}
depKA : \textsf{set of } Row2 \longrightarrow Bool \\
depKA(\rho) == \ \textsf{forall } r, s \textsf{ in set } \rho\ \&\ r.K = s.K \Rightarrow r.A = s.A
\end{array}
$$

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?

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

$$\Sigma = \begin{cases} \text{data sorts} \\ \text{operations} \end{cases}$$

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

$$\mathsf{A} : \Sigma \longrightarrow \mathbf{Set}$$

- another Σ-*model*

$$\mathsf{B} : \Sigma \longrightarrow \mathbf{Set}$$

  refines A iff one can find an epimorphism

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

  (h is a collection of abstraction functions.)

What is $\mathbf{Set}$?

Recall ...

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

$\vdots$

- 1970s ...

$$\begin{cases} \text{D. Scott} \\ \text{C. Strachey} \\ \text{D. Lehmann} \\ \text{B. Smyth} \\ \text{G. Plotkin} \end{cases} \quad \begin{array}{l} \text{J. Adámek} \\ \text{M. Arbib} \\ \text{E. Manes} \\ \vdots \end{array}$$

For our purposes:

- $\mathbf{Set}$ denotes the category of countable sets (details to follow).
- SETS stands for "**Set**-based specification" and calculus.

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 $\Sigma : \Omega \longrightarrow (S^\star \times S)$ of a functionality to each function symbol; as usual, we will write $\sigma : s_1 \ldots s_n \longrightarrow s$ or $s_1 \ldots s_n \xrightarrow{\sigma} s$ as shorthands of $\Sigma\,\sigma = ([s_1, \ldots, s_n], s)$. Let $\mathbf{Set}$ denote the class of all countable (finite or denumerable [a]) sets operated by set-theoretical functions. Let these be denoted by $f : X \longrightarrow Y$ or $X \xrightarrow{f} Y$, where X and Y are sets.

A Σ-*algebra* A is a semantic assignment described by a *functor*

$$\mathsf{A} : \Sigma \longrightarrow \mathbf{Set}$$

that is, $\mathsf{A} = (\mathsf{A}_\Omega, \mathsf{A}_S)$ where A_S maps sorts to corresponding carrier-sets, A_Ω maps operator-symbols to set-theoretical functions, and

$$\mathsf{A}_\Omega\,\sigma : \ \mathsf{A}_S\,s_1 \times \ldots \times \mathsf{A}_S\,s_n \longrightarrow \mathsf{A}_S\,s \tag{2}$$

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

$$\mathsf{A}\,\sigma : \ \mathsf{A}\,s_1 \times \ldots \times \mathsf{A}\,s_n \longrightarrow \mathsf{A}\,s$$

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:

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

where

$$C\,s \stackrel{\text{def}}{=} \{\sigma \in \Omega \mid \ \Sigma\,\sigma = ([\,], s)\}$$

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 A, for a particular signature Σ describing its syntactical structure.

Given a specification model $\mathsf{A} : \Sigma \longrightarrow \mathbf{Set}$, a *reified model* (or refinement) of A is any other model $\mathsf{B} : \Sigma \longrightarrow \mathbf{Set}$ such that there is an epimorphism (*abstraction map*) from B to A. In general, B is more *redundant* a model than A, in the sense that a given abstract value $x \in \mathsf{A}\,s$, for some sort $s \in S$, may be represented (reified, refined) in B by more than one value in $\mathsf{B}\,s$. 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 $h : \mathsf{B} \longrightarrow \mathsf{A}$ [b]. The algorithmic structure of A is preserved by B in the sense that every epimorphism h is a homomorphism, that is,

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

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

---

[a] Following the terminology of [AKM81], by a *denumerable* set A we mean a set whose cardinal number equals $\aleph_0$, the cardinal of $I\!N$.

[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].

The $\mathbf{Set}$-based Notation

$\textsc{Sets}$ should not be regarded as a competitor to specification languages as standard as $\textsc{Vdm-Sl}$ (vulg. $\textsc{Vdm}$) 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 $\mathbf{Set}$ Cartesian closed category:

$$\mathbf{Set}\ \text{Notation} \begin{cases} \text{Objects — \textit{cf.} sets, data structures} \\ \text{Morphisms — \textit{cf.} functions, transactions} \end{cases}$$

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.

  Functional application is denoted by juxtaposition, *e.g.* $f\,a$, or $f(a)$ in case a is a structured expression.
- To every set A we associate the identity function id_A which is such that $id_A\,a = a$, for every $a \in A$ (id will be used wherever A is context-implicit).
- Function composition $f \circ g : C \to B$ is such that

  $$(f \circ g)\,c \;=\; f(g\,c) \tag{4}$$

  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.

$\mathbf{Set}$ Objects — The Primitive Constructs

The following table will help in introducing $\mathbf{Set}$'s primitive constructs to "program hackers":

SETS	PASCAL	C/C++	**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,

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

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

  - it has exponentials (A^B):

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

- the empty set ($\emptyset$) is the category's initial object 0, that is, $0 \xrightarrow{?_A} A$ exists and is unique;
- every singleton set

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

  is "the" terminal object 1, that is, $A \xrightarrow{!_A} 1$ exists and is unique;
- $\mathbf{Set}$ also has finite co-products ($A + B$):

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

(See details in remarks 5 to 8.)

$\mathbf{Set}$ Objects — The Derived Constructs

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

REMARK 5: **Category-theory rudiments (1)**

Most $\mathbf{Set}$ constructs which are familiar to the naïve set theorist are in fact instances of wider categorial notions which will be skimmed through in this and other remarks in the sequel.
Note that this provides but a short collection of category theoretical notions. For further information consult *e.g.* [McL95], among many other textbooks avaliable from the literature.

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

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

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

- *for every three $\mathbf{C}$-objects a, b, c, if* $a \xrightarrow{f} b$ *and* $b \xrightarrow{g} c$ *are morphisms, then the* composition morphism $g \circ f \in \mathbf{C}(a, c)$ *exists;*
- *morphism-composition is associative, ie.* $h \circ (g \circ f) = (h \circ g) \circ f$ *;*
- *for every $\mathbf{C}$-object a, there is an* identity morphism $a \xrightarrow{id_a} a$ *which, for every two $\mathbf{C}$-morphisms* $a \xrightarrow{f} b$ *and* $c \xrightarrow{g} a$ *, is such that*

$$f \circ id_a = f \quad \textit{and} \quad id_a \circ g = g \tag{5}$$

□

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

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

$$\exists\, b \xrightarrow{f^{-1}} a\ .(f \circ f^{-1} = id_b \wedge f^{-1} \circ f = id_a)$$

*(*i.e. *f is* invertible *or "iso"), 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$(\mathbf{C})$.* □

In $\mathbf{Set}$, every *invertible* morphism is a bijection between two isomorphical sets.

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

- *for every $\mathbf{C}$-object a,*

$$\mathsf{F}\, id_a \quad = \quad id_{\mathsf{F}\, a} \tag{6}$$

- *everywhere $g \circ h$ is defined,*

$$\mathsf{F}(g \circ h) \quad = \quad (\mathsf{F}\, g) \circ (\mathsf{F}\, h) \tag{7}$$

□

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

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

$\mathbf{Set}$-endofunctors $\mathsf{F} : \mathbf{Set} \longrightarrow \mathbf{Set}$ play a prominent rôle in (recursive) program specification and calculation. The following concepts of an algebra (dually: coalgebra) of an endofunctor will prove very relevant in the sequel.

Definition 5 (Algebras and Coalgebras of a Functor). *Let* F *be an endofunctor in a given category. Any object* a *together with and arrow from* $\mathsf{F}\,a$ *to* a *is called an* F*-algebra:*

$$(a,\ a \xleftarrow{\ \alpha\ } \mathsf{F}\,a\ )$$

Object a *is said to be the* carrier *of algebra* (a, α).
Any object b *together with and arrow from* b *to* $\mathsf{F}\,b$ *is called an* F*-coalgebra:*

$$(b,\ b \xrightarrow{\ \beta\ } \mathsf{F}\,b\ )$$

Object b *is said to be the* carrier *of coalgebra* (b, β). $\Box$

Definition 6 (F-homomorphism). *Let* F *be an endofunctor in a given category. An* F*-homomorphism from* F*-algebra* (a, α) *to* F*-algebra* (b, β) *is a morphism* $a \xrightarrow{\ f\ } b$ *such that diagram*

$$\begin{tikzcd} a \arrow[d, "f"'] & \mathsf{F}\,a \arrow[l, "\alpha"'] \arrow[d, "\mathsf{F}\,f"] \\ b & \mathsf{F}\,b \arrow[l, "\beta"] \end{tikzcd}$$

commutes. $\Box$

Definition 7 (Initial objects). *A* $\mathbf{C}$*-object* i *is said to be* initial *in* $\mathbf{C}$ *iff, for every other* $\mathbf{C}$*-object* b*, there exists exactly one morphism* $i \xrightarrow{\ ?_b\ } b$ *in* $\mathbf{C}(i, b)$. $\Box$

Theorem 1 (Initial objects are isomorphical). *The unique morphism in* $\mathbf{C}(i, b)$ *of Definition 7 is an isomorphism,* i.e. *every two initial objects are* isomorphical. 0 *is the usual denotation of "the" (up to isomorphism) initial object.* $\Box$

In $\mathbf{Set}$, the empty set $\emptyset$ is the category's *initial object*. In fact, $\mathbf{Set}(\emptyset, A)$ contains only the empty function

$$\bot : \emptyset \longrightarrow A$$

i.e. a function whose domain is $\emptyset$.
The concept of a *terminal object* which follows is dual *wrt.* the preceding one.

Definition 8 (Terminal objects). *A* $\mathbf{C}$*-object* t *is said to be* terminal *in* $\mathbf{C}$ *iff, for every other* $\mathbf{C}$*-object* b*, there exists exactly one morphism* $b \xrightarrow{\ !_b\ } t$ *in* $\mathbf{C}(b, t)$. $\Box$

Theorem 2 (Terminal objects are isomorphical). *The unique morphism in* $\mathbf{C}(b, t)$ *of Definition 8 is an isomorphism, ie. every two terminal objects are* isomorphical. 1 *is the usual denotation of "the" terminal object (up to isomorphism).* $\Box$

In $\mathbf{Set}$, any singleton set $\{x\}$ is a terminal object. In fact, $\mathbf{Set}(A, \{x\})$ contains only the *constant* function $\underline{x} : A \longrightarrow \{x\}$ defined by $\underline{x}\ a = x$, for all $a \in A$.

Definition 9 (Initial F-algebras and F-catamorphisms). *An* F*-algebra* (i, ι) *is said to be* initial *if there is a unique* F*-homomorphism from it to any other* F*-algebra* (a, α)*, which is called the* F*-catamorphism of* (a, α) *and denoted by* $(\![\alpha]\!)_{\mathsf{F}}$. *It follows that* ι *is an isomorphism.*

REMARK 7: **Category-theory rudiments (3)**

Definition 10 (Monics). *A* $\mathbf{C}$*-morphism* $a \xrightarrow{f} b$ *is* monic *if, for every pair of* $\mathbf{C}$*-morphisms* $c \xrightarrow{h,k} a$ *,*

$$\begin{tikzcd} c \arrow[r, shift left, "h"] \arrow[r, shift right, "k"'] & a \arrow[r, "f"] & b \end{tikzcd}$$

if $f \circ h = f \circ k$ *then* $h = k$ *(*f *is "cancellable on the left").*

Definition 11 (Epics). *A* $\mathbf{C}$*-morphism* $a \xleftarrow{f} b$ *is* epic *if, for every pair of* $\mathbf{C}$*-morphisms* $c \xleftarrow{h,k} a$ *,*

$$\begin{tikzcd} c & a \arrow[l, shift right, "k"'] \arrow[l, shift left, "h"] & b \arrow[l, "f"] \end{tikzcd}$$

if $h \circ f = k \circ f$ *then* $h = k$ *(*f *is "cancellable on the right").*

Every iso is monic and epic. Epics in $\mathbf{Set}$ correspond to surjective functions and monics to injective functions. In $\mathbf{Set}$ every monic-epic is iso, but this is not true in general [McL95].

Definition 12 (Products). *Given two* $\mathbf{C}$*-objects* a, b*, the* product $a \times b$ *of* $\{a, b\}$ *is pair*

$$(a \times b, \{\ a \times b \xrightarrow{\pi_1} a\ ,\ a \times b \xrightarrow{\pi_2} b\ \})$$

where $a \times b$ *is a* $\mathbf{C}$*-object such that, for any other* $\mathbf{C}$*-object* c *and morphisms* $c \xrightarrow{f_1} a$ *,* $c \xrightarrow{f_2} b$ *, there is exactly one* $c \xrightarrow{\langle f_1, f_2 \rangle} a \times b$ *such that:*

$$\pi_1 \circ \langle f_1, f_2 \rangle = f_1 \quad \textit{and} \quad \pi_2 \circ \langle f_1, f_2 \rangle = f_2$$

Morphisms π_1 *and* π_2 *are termed* projection *morphisms. Products can be generalized to* finite products $a_1 \times a_2 \times \ldots \times a_n$ *of finite families of* $\mathbf{C}$*-objects* $\{a_i\}_{i=1,n}$ *involving projections* $\{\pi_i\}_{i=1,n}$ *such that*

$$\pi_i \circ \langle f_1, \ldots, f_n \rangle = f_i$$

for the unique $c \xrightarrow{\langle f_1, \ldots, f_n \rangle} a_1 \times a_2 \times \ldots \times a_n$ [a]. □

In $\mathbf{Set}$, the product of two sets A and B is the set $A \times B \stackrel{\rm def}{=} \{(a, b) \mid\ a \in A \wedge b \in B\}$, *i.e.* Cartesian product, and

$$\pi_1(a, b) = a \quad \text{and} \quad \pi_2(a, b) = b$$

The following notion of a *coproduct* is dual *wrt.* a product.

Definition 13 (Coproducts). *Given two* $\mathbf{C}$*-objects* a, b*, the* coproduct $a + b$ *of* $\{a, b\}$ *is pair*

$$(a + b, \{\ a \xrightarrow{i_1} a + b\ ,\ b \xrightarrow{i_2} a + b\ \})$$

where $a + b$ *is a* $\mathbf{C}$*-object such that, for any other* $\mathbf{C}$*-object* c *and morphisms* $a \xrightarrow{f_1} c$ *,* $b \xrightarrow{f_2} c$ *, there is exactly one* $a + b \xrightarrow{[\,f_1, f_2\,]} c$ *such that:*

$$[\,f_1, f_2\,] \circ i_1 = f_1 \quad \textit{and} \quad [\,f_1, f_2\,] \circ i_2 = f_2$$

Morphisms i_1 *and* i_2 *are termed* injection *morphisms. Coproducts can be generalized to* finite coproducts $a_1 + a_2 + \cdots + a_n$ *of finite families of* $\mathbf{C}$*-objects* $\{a_j\}_{j=1,n}$ *involving injections* $\{i_j\}_{j=1,n}$ *such that*

$$[\,f_1, \ldots, f_n\,] \circ i_j = f_j$$

for the unique $a_1 + a_2 + \ldots + a_n \xrightarrow{[\,f_1, \ldots, f_n\,]} c$ [b]. □

[a] The finite product of the empty family of objects corresponds to the notion of a *terminal object*, cf. Definition 8.
[b] The finite coproduct of the empty family of objects corresponds to the notion of an *initial object*, *cf.* Definition 7.

REMARK 8: **Category-theory rudiments (4)**

In $\mathbf{Set}$, the coproduct of two sets A and B is the set

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

i.e. the *disjoint union* of A and B.

Definition 14 (Natural transformations). *Let* $\mathsf{F}, \mathsf{G} : \mathbf{C} \longrightarrow \mathbf{D}$ *be two functors between categories* $\mathbf{C}$ *and* $\mathbf{D}$*. A* natural transformation ν *from* F*-objects into* G*-objects is an* $ob(\mathbf{C})$*-indexed collection of arrows* $\mathsf{F}\,a \xrightarrow{\nu_a} \mathsf{G}\,a$ *such that diagram*

$$\begin{tikzcd} a \arrow[d, "f"'] & \mathsf{F}\,a \arrow[r, "\nu_a"] \arrow[d, "\mathsf{F}\,f"'] & \mathsf{G}\,a \arrow[d, "\mathsf{G}\,f"] \\ b & \mathsf{F}\,b \arrow[r, "\nu_b"'] & \mathsf{G}\,b \end{tikzcd} \tag{8}$$

commutes for all arrows $f : a \longrightarrow b$ *in* $\mathbf{C}$*, that is* [a]*,*

$$(\mathsf{G}\,f) \circ \nu = \nu \circ (\mathsf{F}\,f) \tag{9}$$

holds. □

Natural transformations between $\mathbf{Set}$-endofunctors will prove very useful in the calculus described later in this course.

Definition 15 (Bifunctor). *A bifunctor is a binary functor* F *from two given categories* $\mathbf{C}_1$ *and* $\mathbf{C}_2$ *to another category* $\mathbf{D}$*. So we have,*

- *for every* $\mathbf{C}_1$*-object* a *and* $\mathbf{C}_2$*-object* b*,*

$$\mathsf{F}(id_a, id_b) \quad = \quad id_{\mathsf{F}(a,b)} \tag{10}$$

- *everywhere* $g \circ h$ *is defined in* $\mathbf{C}_1$ *and* $i \circ j$ *is defined in* $\mathbf{C}_2$*,*

$$\mathsf{F}(g \circ h, i \circ j) \quad = \quad \mathsf{F}(g, i) \circ \mathsf{F}(h, j) \tag{11}$$

□

Wherever $\mathbf{C}_1$ and $\mathbf{C}_2$ are the same category, to a given bifunctor F is associated the corresponding *diagonal functor* defined by $a \mapsto \mathsf{F}(a, a)$ for every object a of the source category and $f \mapsto \mathsf{F}(f, f)$ for every morphism f in the same category.

Definition 16 (Binatural transformation). *This extends the concept of a natural transformation to bifunctors: the diagram which follows*

$$\begin{tikzcd} A \arrow[d, "f"'] & C \arrow[d, "g"'] & \mathsf{F}(A, C) \arrow[r, "\nu_{A,C}"] \arrow[d, "\mathsf{F}(f,g)"'] & \mathsf{G}(A, C) \arrow[d, "\mathsf{G}(f,g)"] \\ B & D & \mathsf{F}(B, D) \arrow[r, "\nu_{B,D}"'] & \mathsf{G}(B, D) \end{tikzcd}$$

commutes for every f *and* g*:*

$$\mathsf{G}(f, g) \circ \nu = \nu \circ (\mathsf{F}(f, g)) \tag{12}$$

□

[a] We will fell free to drop subscripts from ν_a or ν_b (consequently overloading ν) for improved readability.

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

$$2^A$$

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

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

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

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

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

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

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

- union types (*cf.* $[A]$ in VDM-SL and "pointers" in the analogy in the table of p.15):

$$A + 1$$

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

Introducing Recursive Objects

Recall good old "linked-lists" in PASCAL:

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

The table of p.15 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":

$$X = 1 + A \times X \tag{13}$$

Many other well-known data structures are specified by $\mathbf{Set}$-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 $\mathbf{Set}$-equations of the form

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

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

(Recursive) Datatypes are Functors

Read "F is a $\mathbf{Set}$-expression" above as "F is a $\mathbf{Set}$ endofunctor"

$$\mathsf{F} \;:\; \mathbf{Set} \longrightarrow \mathbf{Set}$$

(see Definitions 3 and 4).

Basically, a functor extends a **data-structuring** construct to a **polymorphic** programming construct, a kind of standard "browser" over such a data-structure.

Example

Functor $\mathsf{F}\,X = X^\star$ which generates, for $A \xrightarrow{f} B$, the polymorphic construct $A^\star \xrightarrow{f^\star} B^\star$:

$$
\begin{array}{rcl}
f^\star & : & A^\star \longrightarrow B^\star \\
f^\star\, x & \stackrel{\rm def}{=} & [f\ a \mid a \leftarrow x]
\end{array}
\tag{14}
$$

— recall $mapcar$ from LISP:

$$
mapcar(f, x) = \left\{ \begin{array}{rcl} x = [\,] & \Rightarrow & [\,] \\ \neg(x = [\,]) & \Rightarrow & [f\ (head\ x)] +\!\!+\ mapcar(f, tail\ x) \end{array} \right.
$$

In a diagram:

$$
\begin{tikzcd}
A \arrow[r, dotted, no head] \arrow[d, "f"'] & A^\star \arrow[d, "f^\star"] \\
B \arrow[r, dotted, no head] & B^\star
\end{tikzcd}
$$

So we have, *e.g.* for $A = B = I\!N$ and $f\ x = x^2 + 1$,

$$
\begin{array}{rcl}
f^\star\,[2, 5, 3] & = & [2^2 + 1, 5^2 + 1, 3^2 + 1] \\
& = & [5, 26, 10]
\end{array}
$$

- Datatype functoriality is a very important device in $\mathbf{Set}$-based specification and calculation.
- Although **not** every F can participate in the definition of a recursive data domain (we will be back to this topic), $\mathbf{Set}$ has solutions to most common domain equations of the form $X \cong \mathsf{F}\,X$ for endofunctors F involving the above constructs.

Running Examples

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

- *Decision trees:* (*cf.* also dialogue controllers, *etc.*)

$$DecTree \cong What \times (Answer \rightharpoonup DecTree) \tag{15}$$

cf. the following recursive VDM-SL abstract syntax:

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

- *Genealogical diagrams*:

$$GenDia \cong Ind \times (GenDia + 1) \times (GenDia + 1) \tag{16}$$

cf. the following recursive VDM-SL abstract syntax fragment:

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

("pedigree view" of family relationship).

REMARK 9: **More about conventional data structuring and Set**

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:

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

that is, $DecTree \cong \mathsf{F}\, DecTree$ for

$$\mathsf{F}\, X \cong What \times (Answer \rightharpoonup X)$$

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.15 — 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 calculation 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'.*

REMARK 10: **On the adopted pointwise and pointfree notations**

Recall the equation which defines function composition (4):

$$(f \circ g)\, c \quad = \quad f(g\, c)$$

Expression $f(g\,c)$ is said to be written in *pointwise* notation because one is specifying the behaviour of a function at **point** (argument) level. Expression $f \circ g$ is said to be written in *pointfree* notation — argument variables are omitted. Although pointfree notation may be regarded a bit criptic, it provides for substantial economy in writing mathematics. Moreover, it corresponds to the morphism-level notation of category theory.
For instance, our point-level notation for specifying (possibly recursive) functions is that of conventional mathematics, *e.g.*

$$x! = \left\{ \begin{array}{rcl} x = 0 & \Rightarrow & 1 \\ \neg(x = 0) & \Rightarrow & x \times (x-1)! \end{array} \right.$$

instead of the more programming oriented *if-then-else* notation:

$$\begin{array}{rl} x! = & if\; x = 0 \\ & then\; 1 \\ & else\; x \times (x-1)! \end{array}$$

In general, pointwise conditional expressions such as *if (p x) then (g x) else (h x)* are written

$$\left\{ \begin{array}{rcl} p\,x & \Rightarrow & g\,x \\ \neg(p\,x) & \Rightarrow & h\,x \end{array} \right.$$

At pointfree-level, such expressions will be written in the so-called "McCarthy conditional" functional form [McC63],

$$p \rightarrow g, h$$

the meaning of which is

$$p \rightarrow g, h \quad = \quad [\, g, h\,] \circ p? \tag{17}$$

where the $p?$ construct associated to every predicate $A \xrightarrow{p} 2$ is given by [a]

$$\begin{tikzcd} A \arrow[r, "p"] \arrow[d, "p?"'] & 2 \\ A + A \arrow[r, "!_A + !_A"'] & 1 + 1 \arrow[u, "{[\, \underline{\mathrm{TRUE}}, \underline{\mathrm{FALSE}}\,]}"'] \end{tikzcd} \qquad \text{that is} \qquad [\, \underline{\mathrm{TRUE}}, \underline{\mathrm{FALSE}}\,] \circ p? = p \tag{18}$$

A standard predicate is the parameterized equality test $A \xrightarrow{=_x} 2$, for a given $x \in A$, which is such that

$$=_x\; a \quad \stackrel{\mathrm{def}}{=} \quad a = x \tag{19}$$

In this course we resort to pointfree notation wherever convenient. Pointwise functional notation includes all set-theoretical operators typical of constructive specification languages such as *e.g.* VDM-SL or Z dealing with lists (A^*), sets (2^A), mappings ($A \rightharpoonup B$) and so on. Finite mappings are enumerated by writing *e.g.*

$$\left(\begin{array}{ccc} 1 & 2 & 3 \\ a & b & c \end{array} \right)$$

instead of $\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$ (*cf.* VDM-SL). This notation extends to mapping-comprehension, *e.g.*:

$$\left(\begin{array}{c} i \\ i+1 \end{array} \right)_{i \in 3}$$

etc. etc.

[a] $\underline{\mathrm{TRUE}}$ denotes $\lambda.\mathrm{TRUE}$ and $\underline{\mathrm{FALSE}}$ denotes $\lambda.\mathrm{FALSE}$, *cf.* (70) later on. For a more extensive treatment of this categorial approach to conditional expressions read [BdM97].

Repertoire of $\mathbf{Set}$ Constructs and Operators

The $\mathbf{Set}$ operator repertoire is very rich and well structured:

- $\mathbf{Set}$ Operator
  - Set-theoretic — *e.g.* $I\!N \xrightarrow{\lambda n.n^2} I\!N$
  - Category-theoretic
    - Implicit — *e.g.* $A \times B \xrightarrow{\pi_2} B$
    - Universal — *e.g.* $A \xrightarrow{\langle f,g \rangle} B \times C$
    - Functorial — *e.g.* $X^\star \xrightarrow{f^\star} X^\star$
    - Natural — *e.g.* $X^\star \xrightarrow{elems} 2^X$

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,

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B \tag{20}$$

so that

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

The universal one is the *construction* [Bac78],

$$\begin{array}{rcl} \langle f, h \rangle & : & A \longrightarrow C \times D \\ \langle f, h \rangle\, a & \stackrel{\mathrm{def}}{=} & (f\,a, h\,a) \end{array} \tag{21}$$

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

The functorial construct associated to products — the product $f \times g$ of two functions $f : A \longrightarrow C$ and $g : B \longrightarrow D$ — means their "parallel application":

$$
\begin{array}{rcl}
f \times g & : & A \times B \longrightarrow C \times D \\
f \times g\,(a, b) & \stackrel{\mathrm{def}}{=} & (f\,a, g\,b)
\end{array}
\qquad (22)
$$

Using a diagram:

$$
\begin{tikzcd}
B & B \times E \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & E \\
C \arrow[u, "j"] & C \times D \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[u, "j \times i"] & D \arrow[u, "i"'] \\
 & A \arrow[ul, "f"] \arrow[u, "\langle f,h \rangle"] \arrow[ur, "h"'] &
\end{tikzcd}
$$

DISJOINT UNION $(A + B)$

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

$$
A \xrightarrow{i_1} A + B \xleftarrow{i_2} B \qquad (23)
$$

which behave as follows:

$$
i_1\,a = (1, a) \quad , \quad i_2\,b = (2, b) \qquad (24)
$$

The universal one is *alternation*:

$$
\begin{array}{rcl}
[\,f, k\,] & : & A + B \longrightarrow C \\
[\,f, k\,]\,x & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{lcl} x = i_1\,a & \Rightarrow & f\,a \\ x = i_2\,b & \Rightarrow & k\,b \end{array} \right.
\end{array}
\qquad (25)
$$

for $k : B \longrightarrow C$.

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":

$$
\begin{array}{rcl}
f + g & : & A + B \longrightarrow C + D \\
(f + g)\,x & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{lcl} x = i_1\,a & \Rightarrow & i_1(f\,a) \\ x = i_2\,b & \Rightarrow & i_2(g\,b) \end{array} \right.
\end{array}
\qquad (26)
$$

Using a diagram:

$$
\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[d, "f"'] & A + B \arrow[d, "f+g"] & B \arrow[l, "i_2"'] \arrow[d, "g"] \\
C \arrow[r, "i_1"] \arrow[dr, "m"'] & C + D \arrow[d, "{[\,m,n\,]}"] & D \arrow[l, "i_2"'] \arrow[dl, "n"] \\
 & E &
\end{tikzcd}
$$

Constants and Identities

For a given object C it makes sense to build the so-called *constant datatype* C, which is functorially expressed by the following diagram:

$$\begin{tikzcd} A \arrow[r, dotted, dash] \arrow[d, "f"'] & C \arrow[d, "id_C"] \\ B \arrow[r, dotted, dash] & C \end{tikzcd}$$

So, for $\mathsf{F}\,X = C$, we have $\mathsf{F}\,A = C$ (on objects) but $\mathsf{F}\,f = id_C$ (on morphisms).

The identity construct $\mathsf{F}\,X = X$ is functorially obvious:

$$\begin{tikzcd} A \arrow[r, dotted, dash] \arrow[d, "f"'] & A \arrow[d, "f"] \\ B \arrow[r, dotted, dash] & B \end{tikzcd}$$

Exponentials, Powersets and Relations

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

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

(the "array look-up" operation).

The universal one is called *transpose* (or *currying*): given $C \times A \xrightarrow{f} B$, its transpose $C \xrightarrow{\overline{f}} B^A$ is such that

$$ap(\overline{f}\, c, a) \;=\; f(c, a) \tag{27}$$

In a diagram:

$$\begin{tikzcd} B^A & B^A \times A \arrow[r, "ap"] & B \\ C \arrow[u, "\overline{f}"] & C \times A \arrow[u, "\overline{f} \times A"] \arrow[ur, "f"'] & \end{tikzcd}$$

For $B = 2$, $\sigma \in B^A$ 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 denotation of the *powerset of* A data domain, and so $2^{A \times B}$ as denotation of 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:

$$\begin{array}{rcl} 2^f & : & 2^A \longrightarrow 2^B \\ 2^f\, x & \stackrel{\mathrm{def}}{=} & \{ f\, a \mid \ a \in x \} \end{array} \tag{28}$$

In a diagram:

$$\begin{tikzcd} A \arrow[r, dash, dotted] \arrow[d, "f"'] & 2^A \arrow[d, "2^f"] \\ B \arrow[r, dash, dotted] & 2^B \end{tikzcd}$$

Exponentials raise a very useful notion:

POLYNOMIAL STRUCTURES

Polynomial structures are represented by $\mathbf{Set}$-expressions of the form

$$\mathsf{F}\, X \;=\; \sum_{i=0}^{n} C_i \times X^i$$

that is, by polynomial functors (Definition 17).

For instance, the right hand-side of (13) is polynomial of the 1^{st} degree ($n = 1, C_0 = 1, C_1 = A$).

As we shall see, polynomial expressions are very useful in specifying **recursive** data structures.

Polynomiality expresses itself on morphisms in a structured way. For instance, for $\mathsf{F}\,X$ the binary search tree functor $1 + A \times X \times X$, one has

$$
\begin{aligned}
\mathsf{F}\, f &= 1 + A \times f \times f && (29)\\
&= \lambda x.(1 + A \times f \times f)x\\
&= \lambda x. \left\{ \begin{array}{lcl} x = i_1\, a & \Rightarrow & i_1(id_1\, a)\\ x = i_2\, b & \Rightarrow & i_2((A \times f \times f)b) \end{array} \right.\\
&= \lambda x. \left\{ \begin{array}{lcl} x = i_1\, a & \Rightarrow & x\\ x = i_2\, b & \Rightarrow & let \quad b = (a, x_1, x_2)\\ & & in \quad i_2(a, f\, x_1, f\, x_2) \end{array} \right. && (30)
\end{aligned}
$$

FINITE MAPPINGS

Functorial construct: In the same way we derive $A \rightharpoonup C$ from two given sets A and C, given $A \xrightarrow{f} B$ and $C \xrightarrow{g} D$ we may define $f \rightharpoonup g$ as follows:

$$
\begin{aligned}
f \rightharpoonup g \quad &: \quad (A \rightharpoonup C) \longrightarrow (B \rightharpoonup D)\\
(f \rightharpoonup g)\sigma \quad &\stackrel{\mathrm{def}}{=} \quad \left(\begin{array}{c} f\, a\\ g(\sigma\, a) \end{array} \right)_{a \in dom\, \sigma}
\end{aligned}
$$

provided that f be an injective function (monic).

$A \rightharpoonup g$ and $f \rightharpoonup C$ will be accepted as shorthands for, respectively, $id_A \rightharpoonup g$ and $f \rightharpoonup id_C$.

In a diagram:

$$
\begin{tikzcd}
A \arrow[d, "f"'] & C \arrow[d, "g"'] \arrow[r, dotted, no head] & A \rightharpoonup C \arrow[d, "f \rightharpoonup g"] \\
B & D \arrow[r, dotted, no head] & B \rightharpoonup D
\end{tikzcd}
$$

FINITE SEQUENCES

Functorial sequential construct: in the same way we derive $A^\star$ from a given set A, from $A \xrightarrow{f} B$ we define $A^\star \xrightarrow{f^\star} B^\star$ — recall (14) above.

Question:

What have we gained so far from the categorial background?

- An integrated notation in which a **data construct** (data structure, *e.g.* $A \times B$) is equipped with its **standard operators**, the implicit ones (*e.g.*

REMARK 11: **Some notes about $\mathbf{Set}$ 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 (recall Definition 12).

- In the tradition of VDM-like specification languages, we tolerate 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 (recall Definition 13).

- $\mathbb{N}$, the set of natural numbers, is a native object of $\mathbf{Set}$. $\mathbb{N}$'s initial segments are defined by primitive recursion:
$$\overline{n} = \begin{cases} n = 0 & \Rightarrow \quad \emptyset \\ \neg(n = 0) & \Rightarrow \quad \{n\} \cup \overline{n-1} \end{cases} \tag{31}$$
For simplicity, we will write n instead of $\overline{n}$, and therefore 0 instead of the empty set $\emptyset$.

  In $\mathbf{Set}$, 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 "cnnonical" standards among each cardinality class, *e.g.* $2 = \{\text{TRUE}, \text{FALSE}\}$ 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 $\mathbf{Set}$-operators [a]: every time two paths in a valid diagram share the same source and target, by equating the corresponding morphism compositions one records a property of the notation. For instance, from the Cartesian product diagram above of p.27, the following facts can be derived:
$$\begin{aligned} \pi_1 \circ \langle g, h \rangle &= g \\ \pi_2 \circ \langle g, h \rangle &= h \\ \pi_1 \circ (j \times i) &= j \circ \pi_1 \quad (32) \\ \pi_2 \circ (j \times i) &= i \circ \pi_2 \quad (33) \end{aligned}$$
— so
$$\langle j \circ \pi_1, i \circ \pi_2 \rangle = j \times i \tag{34}$$
holds, *etc.*

  See remarks 13 and 14 for more about "extracting" properties from diagrams.

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

π_1, π_2), the universal ones (*e.g.* $\langle f, g\rangle$) and the functorial ones (*e.g.* $f \times g$).

- A very compact pointfree notation which allows for much textual and reasoning economy — check pointfree (*i.e.* variable-free) expression (29) aginst pointwise (*i.e.* variable-level) expression (30).
- Such integration is not arbitrary but is rather rooted on the very general categorial notion of a (co)limit.

  In particular, very interesting properties arise about such constructs, namely the **reflexion**, **fusion** and **absorption** ones — see remarks 13 and 14.

The section which follows provides us with a further device in this direction.

Last But Not least: the Natural Operators

Implicit, universal and functorial "arrows" do not cover all possible functions one wishes to specify, *e.g.* function

$$
\begin{array}{rcl}
elems & : & I\!N^\star \longrightarrow 2^{I\!N} \\
elems\, l & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{rcl} l = [\,] & \Rightarrow & \emptyset \\ \neg(l = [\,]) & \Rightarrow & head\, l \cup elems\,(tail\, l) \end{array} \right.
\end{array}
$$

computing the set of natural numbers which occur in some position of a given sequence of natural numbers.

What is novel about $elems$? It involves instances of two different endofunctors, namely $2^{I\!N}$ (instance of $\lambda X.2^X$) and $I\!N^\star$ (instance of $\lambda X.X^\star$). Many operators typical of VDM-SL or Z notations fall into this class of operators.

Clearly, $elems$ can be made more generic since its definition does not rely on $I\!N$ arithmetics:

$$
\begin{array}{rcl}
elems & : & X^\star \longrightarrow 2^{X} \\
elems\, l & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{rcl} l = [\,] & \Rightarrow & \emptyset \\ \neg(l = [\,]) & \Rightarrow & head\, l \cup elems\,(tail\, l) \end{array} \right.
\end{array}
$$

Comment:

This kind of intuition, typical of "polymorphic" programmers, has been for

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

The functorial behaviour of exponentials 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

$$\begin{tikzcd} A \arrow[r, dotted, no head] \arrow[d, "f"'] & A^C \arrow[d, "f^C"] \\ B \arrow[r, dotted, no head] & B^C \end{tikzcd}$$

This underpins the definition of a polynomial functor:

Definition 17 (Polynomial (Endo)functor). *A polynomial (endo) functor is a functor* $\mathsf{F} : \mathbf{Set} \longrightarrow \mathbf{Set}$ *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 $\mathsf{F} : \mathbf{Set} \longrightarrow \mathbf{Set}$ may be put into the canonical form,

$$\begin{array}{rcl} \mathsf{F}\,X & \cong & C_0 + (C_1 \times X) + (C_2 \times X^2) + \cdots + (C_n \times X^n) \\ & = & \sum_{i=0}^{n} C_i \times X^i \end{array} \quad (35)$$

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

$$(A + B)^n \cong \sum_{p=0}^{n} {}^nC_p \times A^{n-p} \times B^p \quad (36)$$

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

$$\begin{tikzcd} A \arrow[r, dotted, no head] \arrow[d, "f"'] & C^A \arrow[d, "C^f"] \\ B \arrow[r, dotted, no head] & C^B \end{tikzcd}$$

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

REMARK 13: **Reflexion, fusion, absorption and cancellation properties (1)**

This and the following remarks present a collection of basic laws which are of capital relevance to program calculation, as will be explained in the sequel. The names of this properties mirror a standardization which is achieved by the highly organized body of mathematics which category theory is.
For simplicity, the "raison d'être" of these (universal) properties is omitted. For the purposes of this course it suffices to check their validity in the $\mathbf{Set}$ category.

Initial object reflexion :

$$\begin{tikzcd} 0 \arrow[loop above, "?_0 = id_0"] \end{tikzcd}$$

$$?_0 = id_0 \tag{37}$$

Initial object fusion :

$$\begin{tikzcd} 0 \arrow[d, "?_A"'] \arrow[dr, "?_B"] & \\ A \arrow[r, "f"'] & B \end{tikzcd}$$

$$f \circ ?_A = ?_B \tag{38}$$

Terminal object reflexion :

$$\begin{tikzcd} 1 \arrow[loop above, "!_1 = id_1"] \end{tikzcd}$$

$$!_1 = id_1 \tag{39}$$

Terminal object fusion :

$$\begin{tikzcd} 1 & \\ A \arrow[u, "!_A"] & B \arrow[l, "f"] \arrow[ul, "!_B"'] \end{tikzcd}$$

$$!_A \circ f = !_B \tag{40}$$

Product reflexion :

$$\begin{tikzcd} A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\ & A \times B \arrow[ul, "\pi_1"] \arrow[u, "id_{A\times B}"] \arrow[ur, "\pi_2"'] & \end{tikzcd}$$

$$\langle \pi_1, \pi_2 \rangle = id_{A \times B} \tag{41}$$

Product fusion :

$$\begin{tikzcd} A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\ & C \arrow[ul, "g"] \arrow[u, "\langle g,h \rangle"] \arrow[ur, "h"'] & \\ & D \arrow[uul, "g \circ f"] \arrow[u, "f"] \arrow[uur, "h \circ f"'] & \end{tikzcd}$$

$$\langle g, h \rangle \circ f = \langle g \circ f, h \circ f \rangle \tag{42}$$

Product absorption :

$$\begin{tikzcd} A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\ D \arrow[u, "i"] & D \times E \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[u, "i \times j"] & E \arrow[u, "j"'] \\ & C \arrow[ul, "g"] \arrow[u, "\langle g,h \rangle"] \arrow[ur, "h"'] & \end{tikzcd}$$

$$(i \times j) \circ \langle g, h \rangle = \langle i \circ g, j \circ h \rangle \tag{43}$$

Product cancellation :

$$\pi_1 \circ \langle g, h \rangle = g \ , \ \pi_2 \circ \langle g, h \rangle = h \tag{44}$$

REMARK 14: **Reflexion, fusion, absorption and cancellation properties (2)**

Coproduct reflexion :

$$\begin{tikzcd} A \arrow[r, "i_1"] \arrow[dr, "i_1"'] & A+B \arrow[d, "id_{A+B}"] & B \arrow[l, "i_2"'] \arrow[dl, "i_2"] \\ & A+B & \end{tikzcd}$$

$$[\, i_1, i_2 \,] = id_{A+B} \tag{45}$$

Coproduct fusion :

$$\begin{tikzcd} A \arrow[r, "i_1"] \arrow[dr, "g"] \arrow[ddr, "f \circ g"'] & A+B \arrow[d, "{[\, g,h \,]}"] & B \arrow[l, "i_2"'] \arrow[dl, "h"'] \arrow[ddl, "f \circ h"] \\ & C \arrow[d, "f"] & \\ & D & \end{tikzcd}$$

$$f \circ [\, g, h \,] = [\, f \circ g, f \circ h \,] \tag{46}$$

Coproduct absorption :

$$\begin{tikzcd} A \arrow[r, "i_1"] \arrow[d, "i"'] & A+B \arrow[d, "i+j"'] & B \arrow[l, "i_2"'] \arrow[d, "j"] \\ D \arrow[r, "i_1"] \arrow[dr, "g"'] & D+E \arrow[d, "{[\, g,h \,]}"] & E \arrow[l, "i_2"] \arrow[dl, "h"] \\ & C & \end{tikzcd}$$

$$[\, g, h \,] \circ (i + j) = [\, g \circ i, h \circ j \,] \tag{47}$$

Coproduct cancellation :

$$[\, g, h \,] \circ i_1 = g \;,\; [\, g, h \,] \circ i_2 = h \tag{48}$$

Exponentials reflexion :

$$\begin{tikzcd} B^A \\ B^A \arrow[u, "id_{B^A}"] \end{tikzcd} \qquad \begin{tikzcd} B^A \times A \arrow[r, "ap"] & B \\ B^A \times A \arrow[u, "id_{B^A} \times A"] \arrow[ur, "ap"'] & \end{tikzcd}$$

$$\overline{ap} = id_{B^A} \tag{49}$$

Exponentials fusion :

$$\begin{tikzcd} B^A \\ C \arrow[u, "\overline{g}"] \\ D \arrow[u, "f"] \end{tikzcd} \qquad \begin{tikzcd} B^A \times A \arrow[r, "ap"] & B \\ C \times A \arrow[u, "\overline{g} \times A"] \arrow[ur, "g"] & \\ D \times A \arrow[u, "f \times A"] \arrow[uur, "g \circ (f \times A)"'] & \end{tikzcd}$$

$$\overline{g \circ (f \times A)} = \overline{g} \circ f \tag{50}$$

Exponentials absorption :

$$\begin{tikzcd} D^A \\ B^A \arrow[u, "f^A"] \\ C \arrow[u, "\overline{g}"] \end{tikzcd} \qquad \begin{tikzcd} D^A \times A \arrow[r, "ap"] & D \\ B^A \times A \arrow[u, "f^A \times A"] \arrow[r, "ap"] & B \arrow[u, "f"'] \\ C \times A \arrow[u, "\overline{g} \times A"] \arrow[ur, "g"] & \end{tikzcd}$$

$$\overline{f \circ g} = f^A \circ \overline{g} \tag{51}$$

Exponentials cancellation :

$$ap \circ (\overline{g} \times A) = g \tag{52}$$

REMARK 15: **More about finite sets, sequences and relations**

- Occasionally, we may write $f[x]$ instead of $2^f\,x$, recall (28). This construct 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

(53)

is specified by mathematical object

$$r \;=\; \{(a,b,c),(a,d,e),(f,d,g)\}$$

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

$$\begin{array}{rcl} proj & : & \overline{n} \times 2^{A_1\times\ldots\times A_n} \longrightarrow \bigcup_{i=1}^{n} 2^{A_i} \\ proj(i,t) & \stackrel{\rm def}{=} & \{\pi_i\,r \mid\ r \in t\} \\ & = & 2^{\pi_i}\,t \end{array} \tag{54}$$

and

$$\begin{array}{rcl} sel & : & (\sum_{i=1}^{n} 2^{A_i}) \times 2^{A_1\times\ldots\times A_n} \longrightarrow 2^{A_1\times\ldots\times A_n} \\ sel((i,s),\tau) & \stackrel{\rm def}{=} & \{t \in \tau \mid\ (\pi_i\,t) \in s\} \end{array} \tag{55}$$

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

$$r^{-1} = \{(b,a) \mid\ (a,b) \in r\} \tag{56}$$

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

(57)

$$2^f(\emptyset) = \emptyset \tag{58}$$
$$2^f(x \cup y) = 2^f\,x \cup 2^f\,y \tag{59}$$
$$2^f(x \cap y) = 2^f\,x \cap 2^f\,y \tag{60}$$
$$(r \circ s)^{-1} = s^{-1} \circ r^{-1} \tag{61}$$
$$r \circ (s \cup t) = (r \circ s) \cup (r \circ t) \tag{62}$$
$$r \circ \emptyset = \emptyset \tag{63}$$

□

REMARK 16: **On the usefulness of a functorial approach to data structuring (1)**

Following [MA86], it has become standard in the program calculation community to regard a "data sort" as an (endo)functor in some category [Oli92, BdM97] — $\mathbf{Set}$, in our case. For instance, constant functors such as

$$\mathsf{F}\,X \quad \cong \quad 2$$

provide for primitive data sorts, while other functors such as

$$\mathsf{F}\,X \quad \cong \quad X^*$$

(*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,

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

where

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

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 $\lambda X.A \rightharpoonup X$ (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

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

where

$$2^f\, x \quad \stackrel{\rm def}{=} \quad \{f\,a \mid\ a \in x\}$$

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

$$\begin{tikzcd}
A \arrow[r, dotted, no head] \arrow[d, "f"'] & \mathsf{F}\,A \arrow[d, "\mathsf{F}\,f"] \\
B \arrow[r, dotted, no head] & \mathsf{F}\,B
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
A \arrow[d, "f"'] & C \arrow[r, dotted, no head] \arrow[d, "g"'] & \mathsf{G}(A,C) \arrow[d, "\mathsf{G}(f,g)"] \\
B & D \arrow[r, dotted, no head] & \mathsf{G}(B,D)
\end{tikzcd}$$

(the obvious extension of G to n-ary parametric data sorts is of little practical interest.)

REMARK 17: **On the usefulness of a functorial approach to data structuring (2)**

$\mathbf{Set}$-functorial approach to data structuring extends naturally to the specification of recursive data sorts [a]. These are described by fixpoint endofunctors, for instance

$$\mathsf{F}\,X \;\cong\; \mu Y.(1 + X \times Y^2)$$

— *cf.* equation

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

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

$$\mathsf{F}\,f \;\stackrel{\mathrm{def}}{=}\; \mu y.(1 + f \times y^2)$$

that is, function [b]

$$(\mathsf{F}\,f)y \;\stackrel{\mathrm{def}}{=}\; \left\{ \begin{array}{rcl} y = NIL & \Rightarrow & y \\ y = (x, y_1, y_2) & \Rightarrow & (f\,x, (\mathsf{F}\,f)y_1, (\mathsf{F}\,f)y_2) \end{array} \right.$$

and we have the following property,

$$\mathsf{F}\,(f \circ g) \;=\; (\mathsf{F}\,f) \circ (\mathsf{F}\,g)$$

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 classical 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) F, there is an immediate extension of such a specification which includes the F-extension of all morphisms in the original collection. Moreover, every such an extension is packaged with the following generic (*i.e.* functorial) properties:

$$\begin{array}{rcl} \mathsf{F}\,(f \circ g) & = & (\mathsf{F}\,f) \circ (\mathsf{F}\,g) \\ \mathsf{F}\,id_A & = & id_{\mathsf{F}\,A} \end{array}$$

(as we know, id_A denotes the identity function on data sort A).

Exercise 2 Check each of the following equalities:

$$\begin{array}{rcll} (f \times g) \circ \langle \pi_2, \pi_1 \rangle & = & \langle \pi_2, \pi_1 \rangle \circ (g \times f) & (64) \\ (f \circ h) \times (g \circ i) & = & (f \times g) \circ (h \times i) & (65) \\ id_{A \times B} & = & id_A \times id_B & (66) \\ \pi_1 \circ \langle f, g \rangle & = & f & \\ \pi_2 \circ \langle f, g \rangle & = & g & \\ (f \circ h) + (g \circ i) & = & (f + g) \circ (h + i) & (67) \\ id_{A+B} & = & id_A + id_B & (68) \\ [\,f, g\,] \circ i_1 & = & f & \\ [\,f, g\,] \circ i_2 & = & g & \end{array}$$

□

[a] We will provide details about this fact later on, in Remark 21.

[b] The injection functions strictly required by alternation (26) have been dropped from the body of this function, for improved readability.

many years the very basis of "à la $\mathrm{V}\scriptstyle{\mathrm{DM}}$" specification techniques, grounded on naïve set-theory.

In our calculus, this is just another beneficial consequence of the functorial approach to data modelling — polymorphic operators are but "functor transformers", that is, *natural transformations* (Definition 14) in category-theoretic terms:

- **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.

For instance, the *theorem for free* of the $elems$ function (p.32) is

$$2^f \circ elems \;=\; elems \circ f^\star \tag{69}$$

which is captured by the following diagram

$$\begin{array}{ccc} A & & A^\star \xrightarrow{elems_A} 2^A \\ {\scriptstyle f}\downarrow & & {\scriptstyle f^\star}\downarrow \qquad\quad \downarrow{\scriptstyle 2^f} \\ B & & B^\star \xrightarrow[elems_B]{} 2^B \end{array}$$

(for simplicity, we drop subscripts in natural transformation equations).

Even constructs as simple as the constant function

$$\begin{array}{rcl} \underline{c} & : & A \longrightarrow C \\ \underline{c}\, a & \stackrel{\mathrm{def}}{=} & c \end{array} \tag{70}$$

(where $C \neq 0$ and $c \in C$) have a *theorem for free*:

$$\begin{array}{ccc} A & & A \xrightarrow{\underline{c}_A} C \\ {\scriptstyle f}\downarrow & & {\scriptstyle f}\downarrow \qquad \downarrow{\scriptstyle 1_C} \\ B & & B \xrightarrow[\underline{c}_B]{} C \end{array} \qquad (1_C) \circ \underline{c} = \underline{c} \circ (f) \tag{71}$$

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

- Head and tail of a sequence:

$$\begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad \begin{tikzcd} A^\star \arrow[r, "head_A"] \arrow[d, "f^\star"'] & A \arrow[d, "f"] \\ B^\star \arrow[r, "head_B"'] & B \end{tikzcd} \quad (f) \circ head = head \circ (f^\star) \tag{72}$$

$$\begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad \begin{tikzcd} A^\star \arrow[r, "tail_A"] \arrow[d, "f^\star"'] & A^\star \arrow[d, "f^\star"] \\ B^\star \arrow[r, "tail_B"'] & B^\star \end{tikzcd} \quad (f^\star) \circ tail = tail \circ (f^\star) \tag{73}$$

- Sequence concatenation:

$$\begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad \begin{tikzcd} A^\star \times A^\star \arrow[r, "+\!\!+_A"] \arrow[d, "f^\star \times f^\star"'] & A^\star \arrow[d, "f^\star"] \\ B^\star \times B^\star \arrow[r, "+\!\!+_B"'] & B^\star \end{tikzcd} \quad (f^\star) \circ +\!\!+ = +\!\!+ \circ (f^\star \times f^\star) \tag{74}$$

- Domain of a finite mapping (monic f):

$$\begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad \begin{tikzcd} A \rightharpoonup C \arrow[r, "dom_A"] \arrow[d, "f \rightharpoonup C"'] & 2^A \arrow[d, "2^f"] \\ B \rightharpoonup C \arrow[r, "dom_B"'] & 2^B \end{tikzcd} \quad (2^f) \circ dom = dom \circ (f \rightharpoonup C) \tag{75}$$

- Range of a finite mapping:

$$\begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad \begin{tikzcd} C \rightharpoonup A \arrow[r, "rng_A"] \arrow[d, "C \rightharpoonup f"'] & 2^A \arrow[d, "2^f"] \\ C \rightharpoonup B \arrow[r, "rng_B"'] & 2^B \end{tikzcd} \quad (2^f) \circ rng = rng \circ (C \rightharpoonup f) \tag{76}$$

- *etc. etc.*

See Remark 25 for further *theorems for free* which prove especially useful in program calculations.

REMARK 18: **On the algebras of finite mappings and finite sequences**

The following properties hold for arbitrary finite mappings f, g e h involving standard operators $\cup$ (coherent map union), $\dagger$ (map overwrite), dom (map domain), rng (map range), $|$ (map domain restriction) and $\setminus$ (map domain subtraction). S and R denote finite sets.

$$
\begin{array}{rcll}
f \cup g & = & g \cup f & (77)\\
f \cup f & = & f & (78)\\
f \dagger f & = & f & (79)\\
f \cup (g \cup h) & = & (f \cup g) \cup h & (80)\\
f \dagger (g \dagger h) & = & (f \dagger g) \dagger h & (81)\\
f & = & f \cup (\quad) & (82)\\
f & = & (\quad) \cup f & (83)\\
f & = & f \dagger (\quad) & (84)\\
f & = & (\quad) \dagger f & (85)\\
f[dom\, f] & = & rng\, f & (86)\\
(f \setminus S) \setminus R & = & f \setminus (S \cup R) & (87)\\
f \circ (g \setminus S) & = & (f \circ g) \setminus S & (88)\\
f \mid (S \cup R) & = & (f \mid S) \cup (f \mid R) & (89)\\
(f \mid R) \mid S & = & (f \mid S) \mid R & (90)
\end{array}
\qquad
\begin{array}{rcll}
(f \mid R) \setminus R & = & f \mid \emptyset & (91)\\
f \setminus S \cup f \mid S & = & f & (92)\\
(f \setminus R) \mid S & = & f \mid (S - R) & (93)\\
S \cap dom\, f = \emptyset & \Rightarrow & f \setminus S = f & (94)\\
dom\, f \subseteq S & \Rightarrow & f \mid S = f & (95)\\
dom\,(f \mid S) & = & (dom\, f \cap S) & (96)\\
f \dagger g & = & f \setminus dom\, g \cup g & (97)\\
f \mid dom\, f & = & f & (98)\\
f & = & f \setminus S \cup f \mid S & (99)\\
f \setminus dom\, f & = & (\quad) & (100)\\
f \setminus (S \cap R) & = & f \setminus S \cup f \setminus R & (101)\\
(\quad) \setminus S & = & (\quad) & (102)\\
(\quad) \mid S & = & (\quad) & (103)
\end{array}
$$

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

$$
\begin{array}{rcll}
head(cons(a,s)) & = & a & (104)\\
tail(cons(a,s)) & = & s & (105)\\
s +\!\!+ [\,] & = & s & (106)\\
[\,] +\!\!+ s & = & s & (107)\\
s +\!\!+ (r +\!\!+ t) & = & (s +\!\!+ r) +\!\!+ t & (108)\\
g^* \circ f^* & = & (g \circ f)^* & (109)\\
length\,[\,] & = & 0 & (110)\\
f^*(s +\!\!+ r) & = & f^*\, s +\!\!+ f^*\, r & (111)
\end{array}
\qquad
\begin{array}{rcll}
length(cons(a,s)) & = & 1 + length\, s & (112)\\
length \circ f^* & = & length & (113)\\
elems\,[\,] & = & \emptyset & (114)\\
elems(cons(a,s)) & = & \{a\} \cup elems\, s & (115)\\
elems \circ f^* & = & 2^f \circ elems & (116)\\
elems\,[f\,a \mid a \leftarrow s \wedge p\,a] & \subseteq & 2^f(elems\, s) & (117)
\end{array}
$$

For $s \neq [\,]$ and $1 \leq i \leq length\, s$, one has:

$$
\begin{array}{rcll}
(tail\; s)i & = & s(i+1) & (118)\\
tail(s +\!\!+ r) & = & (tail\; s) +\!\!+ r & (119)\\
head(s +\!\!+ r) & = & head\, s & (120)\\
elems(tail\, s) & \subseteq & elems\, s & (121)\\
length(tail\, s) & = & length\, s - 1 & (122)\\
cons \circ \langle head, tail \rangle & = & id & (123)
\end{array}
$$

Exercise 3 Show that:

$$
\begin{array}{rcll}
\setminus \circ \langle id, dom \rangle & = & id & (124)\\
dom \circ (A \rightharpoonup f) & = & dom & (125)
\end{array}
$$

□

Introduction to $\mathbf{Set}$-based Calculation

Questions

- What does it mean to say that two $\mathbf{Set}$-objects 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.*

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

equivalent to

```
record
    F: record
           F: A;
           S: B
       end;
    S: C;
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 "$\cong$" ($\mathbf{Set}$-isomorphism):

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

- $\mathbf{Set}$ 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.15 is a suggestive exercise.

The $\cong$-Subcalculus

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

$$\begin{align*}
A \times B &\cong B \times A \tag{126}\\
A \times (B \times C) &\cong (A \times B) \times C \tag{127}\\
A \times 1 &\cong A \tag{128}\\
A + B &\cong B + A \tag{129}\\
A + (B + C) &\cong (A + B) + C \tag{130}\\
A + 0 &\cong A \tag{131}\\
A \times 0 &\cong 0 \tag{132}\\
A \times (B + C) &\cong (A \times B) + (A \times C) \tag{133}
\end{align*}$$

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

$$A_1 \times \ldots \times A_n \ \text{ that is } \ \Pi_{i=1}^{n} A_i$$

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

and we have

$$\begin{align*}
\underbrace{A \times \ldots \times A}_{n} &\cong A^n \tag{134}\\
\underbrace{A + \ldots + A}_{n} &\cong n \times A \tag{135}
\end{align*}$$

Concerning exponentials we have:

$$\begin{align*}
A^0 &\cong 1 \tag{136}\\
A^1 &\cong A \tag{137}\\
(B \times C)^A &\cong B^A \times C^A \tag{138}\\
C^{A \times B} &\cong \left(C^B\right)^A \tag{139}\\
A^{B+C} &\cong A^B \times A^C \tag{140}\\
1^A &\cong 1 \tag{141}
\end{align*}$$

Relating exponentials with finite mappings we have:

$$\begin{aligned} A \rightharpoonup B &\cong (B+1)^A && (142) \\ (A \rightharpoonup B)^C &\cong (C \times A) \rightharpoonup B && (143) \end{aligned}$$

Also note that

$$A^\star \cong \sum_{n \geq 0} A^n \qquad (144)$$

$$A \rightharpoonup B \cong \sum_{K \subseteq A} B^K \qquad (145)$$

since

$$\begin{aligned} A \neq B &\Rightarrow X^A \cap X^B = \emptyset \\ A \cap B = \emptyset &\Rightarrow A \cup B \cong A + B \end{aligned}$$

Example of $\cong$-Reasoning

Re-write (15) abstractly as follows,

$$T \cong I \times (A \rightharpoonup T) \qquad (146)$$

and instantiate $A = 2$. Then

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

that is, (146) becomes

$$T \cong I \times (T+1) \times (T+1) \qquad (147)$$

Finally, compare (147) with (16).

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.).

Recursive Specification Revisited

Consider school mathematics equation

$$x = 3 + \frac{x}{2} \tag{148}$$

in $\mathbb{R}$. What does it mean?

- A standard answer: if solved for x, it will exhibit solution $x = 6$.
- A less standard one: it is a recursive definition of number $6 \ldots$

Now, what do **we** exactly **mean** by a recursive data type specification $X = \mathsf{F}\,X$?

- We want to regard $X = \mathsf{F}\,X$ as an equation in $\mathbf{Set}$ and solve it for X.
- However, "=" is not very useful in $\mathbf{Set}$ — we should write

  $$X \;\cong\; \mathsf{F}\,X \tag{149}$$

  instead.
- Back to (148):
  its solution is not 6 alone, but rather the pair $(6, 6 = 3 + \frac{6}{2})$ providing evidence that 6 turns the equation into a tautology.

  'Mutatis mutandis' (149):
  a solution to this equation will be a pair $(X_0,\ X_0 \xleftarrow{\iota} \mathsf{F}\,X_0\ )$, where ι is an isomorphism providing evidence of fact $X_0 \cong \mathsf{F}\,X_0$.
- In summary: a solution to (149) will be an F-algebra in $\mathbf{Set}$ — recall Definition 5.

  Obviously: the F-coalgebra $(X_0,\ X_0 \xrightarrow{\iota^{-1}} \mathsf{F}\,X_0\ )$ could also be selected as "witness" of the same isomorphism.

Example

Equation: (13)

Endofunctor: $\mathsf{F}\,X = 1 + A \times X$

Solution: $(A^\star,\ A^\star \xleftarrow{\iota} \mathsf{F}\, A^\star\,)$ such that

$$\frac{\begin{array}{l} A^\star \xleftarrow{\quad nil \quad} 1 \\ A^\star \xleftarrow{\quad cons \quad} A \times A^\star \end{array}}{A^\star \xleftarrow[\iota = [\, nil, cons\,]]{} 1 + A \times A^\star} \qquad \text{where } nil \stackrel{\mathrm{def}}{=} \underline{[\,]}$$

Inverse solution: $(A^\star, \iota^{-1})$ where

$$\iota^{-1} \;=\; (!_{A^\star} + \langle head, tail\rangle) \circ =_{[\,]}?$$

that is

$$\begin{array}{rcl} \iota^{-1}\, l & = & (!_{A^\star} + \langle head, tail\rangle) \left\{ \begin{array}{rcl} l = [\,] & \Rightarrow & i_1\, l \\ \neg(l = [\,]) & \Rightarrow & i_2\, l \end{array} \right. \\ & = & \left\{ \begin{array}{rcl} l = [\,] & \Rightarrow & i_1\, NIL \\ \neg(l = [\,]) & \Rightarrow & i_2(head\, l, tail\, l) \end{array} \right. \end{array}$$

Question:

What is the relation between $(A^\star, \iota)$ and any other F-algebra, *e.g.* $(2^A, \alpha)$ such that

$$\frac{\begin{array}{l} 2^A \xleftarrow{\quad \underline{\emptyset} \quad} 1 \\ 2^A \xleftarrow{\quad puts \quad} A \times 2^A \end{array}}{2^A \xleftarrow[\alpha = [\, \underline{\emptyset}, puts\,]]{} 1 + A \times 2^A} \qquad \text{where } puts(a, s) = \{a\} \cup s \text{ ?}$$

Because F is a functor, we can draw the diagram which follows

$$\begin{array}{ccc} A^\star & \underset{\iota}{\overset{\iota^{-1}}{\rightleftarrows}} & 1 + A \times A^\star \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle 1 + A \times f} \\ 2^A & \xleftarrow[\alpha]{} & 1 + A \times 2^A \end{array}$$

that is, write equation

$$f \;=\; \alpha \circ (1 + A \times f) \circ \iota^{-1}$$

and "solve" it for f:

$$
\begin{array}{rcl}
f & = & \alpha \circ (1 + A \times f) \circ \iota^{-1} \\
& = & \quad \{ \text{ definitions of } \alpha \text{ and } \iota^{-1} \} \\
& & [\, \underline{\emptyset}, puts \,] \circ (1 + A \times f) \circ (!_{A^\star} + \langle head, tail \rangle) \circ =_{[\,]}? \\
& = & \quad \{ \; + \text{ is bifunctor (11) } \} \\
& & [\, \underline{\emptyset}, puts \,] \circ (id_1 \circ !_{A^\star} + (A \times f) \circ \langle head, tail \rangle) \circ =_{[\,]}? \\
& = & \quad \{ \text{ identity (5) and } \times\text{-absorption (43)} \} \\
& & [\, \underline{\emptyset}, puts \,] \circ (!_{A^\star} + \langle id_A \circ head, f \circ tail \rangle) \circ =_{[\,]}? \\
& = & \quad \{ \; +\text{-absorption (47) } \} \\
& & [\, \underline{\emptyset} \circ !_{A^\star}, puts \circ \langle id_A \circ head, f \circ tail \rangle \,] \circ =_{[\,]}? \\
& = & \quad \{ \text{ constant function } \underline{\emptyset} \text{ (70) and identity (5) } \} \\
& & [\, \underline{\emptyset}, puts \circ \langle head, f \circ tail \rangle \,] \circ =_{[\,]}? \\
& = & \quad \{ \text{ introducing McCarthy's conditional (17) } \} \\
& & =_{[\,]} \to \underline{\emptyset}, puts \circ \langle head, f \circ tail \rangle \\
& = & \quad \{ \text{ going pointwise and definition of } =_{[\,]} \text{ (19) } \} \\
& & \lambda l. \left\{ \begin{array}{rcl} l = [\,] & \Rightarrow & \emptyset \\ \neg(l = [\,]) & \Rightarrow & puts(head\, l, f(tail\, l)) \end{array} \right. \\
& = & \quad \{ \text{ definition of } puts \} \\
& & \lambda l. \left\{ \begin{array}{rcl} l = [\,] & \Rightarrow & \emptyset \\ \neg(l = [\,]) & \Rightarrow & \{head\, l\} \cup f(tail\, l) \end{array} \right.
\end{array}
$$

Conclusions:

- f — which is but the $elems$ function (p.32) — is an F-homomorphism (Definition 6).
- In fact, it is the *unique* F-homomorphism from $(A^\star, \iota)$ to $(2^A, \alpha)$, because ι is an isomorphism — $(A^\star, \iota)$ is the **initial** F-algebra.
- So, α fully determines f: one says that f is the F-*catamorphism* determined by α and write $(\![\alpha]\!)_{\mathsf{F}}$ instead of f — see Definition 9 and Remark 19.

REMARK 19: **Introducing catamorphisms**

The definition of a function

$$f : B \longrightarrow A$$

can be regarded as a kind of "contract": the function is committed to produce an A-value provided it is supplied with a B-value.
In many situations we know that B happens to be the least fixpoint $\mu\mathsf{F}$ of a given equation $X \cong \mathsf{F}\,X$. Intuitively, f is expected to be recursive.
Wherever it exists [a], the least fixpoint $\mu\mathsf{F}$ is the carrier of the initial F-algebra (say ι_F). So it is natural to express f as an F-*catamorphism* (Definition 9) of some F-algebra α on target type A:

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "f=(\!|\alpha|\!)_\mathsf{F}"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota_\mathsf{F}"'] \arrow[d, "\mathsf{F}(\!|\alpha|\!)_\mathsf{F}"] \\ A & \mathsf{F}\,A \arrow[l, "\alpha"] \end{tikzcd}$$

— such is the case of $elems$ (p.32), as checked in the main text. Initiality provides catamorphisms with the expected (universal) properties:

Catamorphism fusion :

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "(\!|\alpha|\!)_\mathsf{F}"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota_\mathsf{F}"'] \arrow[d, "\mathsf{F}(\!|\alpha|\!)_\mathsf{F}"] \\ A \arrow[d, "f"'] & \mathsf{F}\,A \arrow[l, "\alpha"] \arrow[d, "\mathsf{F}\,f"] \\ B & \mathsf{F}\,B \arrow[l, "\beta"] \end{tikzcd}$$

$$f \circ (\!|\alpha|\!)_\mathsf{F} = (\!|\beta|\!)_\mathsf{F} \quad \text{if} \quad f \circ \alpha = \beta \circ \mathsf{F}\,f \tag{150}$$

Catamorphism reflexion :

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "(\!|\iota_\mathsf{F}|\!)_\mathsf{F}"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota_\mathsf{F}"'] \arrow[d, "\mathsf{F}(\!|\iota_\mathsf{F}|\!)_\mathsf{F}"] \\ \mu\mathsf{F} & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota_\mathsf{F}"] \end{tikzcd}$$

$$(\!|\iota_\mathsf{F}|\!)_\mathsf{F} = id_{\mu\mathsf{F}} \tag{151}$$

In summary, $(\!|\alpha|\!)_\mathsf{F}$ captures the abstract notion of a $\mu\mathsf{F}$ "browser", "parser", or better: of the abstract F-induced recursion schema.
Note that laws such as (150) and (151) express themselves independently of F. For instance, the natural number *initial segment* operator (31) arises as catamorphism $(\!|[\,\underline{\emptyset}, \lambda m.\{m\}\,]|\!)_{\lambda X.1+X}$ because $\mathbb{N}_0$ is the least fixpoint of the "natural number" equation $X \cong 1 + X$ [b]. So we are talking about *higher-order* polymorphism, — that is, about *polytypism* [JJ96].
Moreover, in the same way we identified the source type B above with $\mu\mathsf{F}$, for some F, we may happen to identify the target type A with $\mu\mathsf{G}$, for some G:

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "f"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota_\mathsf{F}"'] \arrow[d, "\mathsf{F}\,f"] \\ \mu\mathsf{G} & \mathsf{F}\,\mu\mathsf{G} \arrow[l, "\alpha"] \end{tikzcd} \tag{152}$$

Now, it would be "unnatural" to ignore the initial G-algebra ι_G, which is the standard constructor of values of type $\mu\mathsf{G}$.
Clearly, f analyses input data according to the F recursive pattern and synthesizes output data according to the G recursive pattern — it behaves like a "protocol" between such patterns of recursion.
How can such a protocol be captured in this setting? An F to G natural transformation (and its *theorem for free*) seems the most "natural" device for this purpose. This is illustrated and explained in remarks 20 and 21, bringing about the broader concept of a *hylomorphism*.

[a] This happens at least for polynomial F (Def. 17) or, more generally, for *regular* F [Par98].

[b] The initial algebra is $(\mathbb{N}_0, [\,\underline{0}, suc\,])$, for $suc\,n = n + 1$. Note that this catamorphism may be regarded as the counterpart of $elems$ for the natural number equation functor.

REMARK 20: **Introducing hylomorphisms**

The issue raised by diagram (152) can be approached by somehow "pasting" ι_{G} into the diagram,

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[d, "f"'] & & \mathsf{F}\,\mu\mathsf{F} \arrow[ll, "\iota_{\mathsf{F}}"'] \arrow[d, "\mathsf{F}\,f"] \\
\mu\mathsf{G} & \mathsf{G}\,\mu\mathsf{G} \arrow[l, "\iota_{\mathsf{G}}"] & \mathsf{F}\,\mu\mathsf{G} \arrow[l, "\cdots"]
\end{tikzcd}$$

leaving it open how to fill the " ... " arrow. First, let us illustrate this diagram with a typical "protocol" function, that which should compute the length of a finite sequence:

$$\begin{tikzcd}
C^\star \arrow[d, "length"'] & & 1 + C \times C^\star \arrow[ll, "{[\,nil,cons\,]}"'] \arrow[d, "1+C\times length"] \\
\mathbb{N}_0 & 1 + \mathbb{N}_0 \arrow[l, "{[\,\underline{0},suc\,]}"] & 1 + C \times \mathbb{N}_0 \arrow[l, "1+\pi_2"]
\end{tikzcd}$$

In this example, the " ... " arrow was filled with natural transformation $1 + \pi_2$,

$$\begin{tikzcd}
A \arrow[d, "f"'] & 1 + C \times A \arrow[r, "(1+\pi_2)_A"] \arrow[d, "1+C\times f"'] & 1 + A \arrow[d, "1+f"] \\
B & 1 + C \times B \arrow[r, "(1+\pi_2)_B"'] & 1 + B
\end{tikzcd}$$

which provides for the required recursion pattern "protocol". So, the diagram can be enriched thus,

$$\begin{tikzcd}
C^\star \arrow[d, "length"'] & 1 + C^\star \arrow[d, "1+length"'] & 1 + C \times C^\star \arrow[l, "1+\pi_2"] \arrow[ll, bend right, "{[\,nil,cons\,]}"'] \arrow[d, "1+C\times length"] \\
\mathbb{N}_0 & 1 + \mathbb{N}_0 \arrow[l, "{[\,\underline{0},suc\,]}"] & 1 + C \times \mathbb{N}_0 \arrow[l, "1+\pi_2"]
\end{tikzcd}$$

immediately delivering properties

$$length \circ nil = \underline{0} \tag{153}$$
$$length \circ cons = succ \circ length \circ \pi_2 \tag{154}$$

whatever path is followed up in the right-hand square of the diagram.
A constructive definition of $length$ will pop out by closing the left-hand side square of the diagram with coalgebra $C^\star \xrightarrow{\gamma} 1 + C^\star$, which is fully determined by

$$\begin{aligned}
\gamma &= (1 + \pi_2) \circ [\,nil, cons\,]^{-1} \\
&= (! + tail) \circ =_{[\,]}?
\end{aligned}$$

Therefore,

$$length \stackrel{\text{def}}{=} [\,\underline{0}, suc \circ length \circ tail\,] \circ =_{[\,]}? \tag{155}$$

What have been the basic building blocks of this specification? Input F-coalgebra γ + output G-algebra $[\,\underline{0}, suc\,]$ + natural transformation $1 + \pi_2 : \mathsf{F} \longrightarrow \mathsf{G}$. These are precisely the components of a so-called *hylomorphism*, as is presented in [TM95] and defined by Definition 18 of the remark which follows.

REMARK 21: **Hylomorphisms and type functors**

Definition 18 (Hylomorphism). *Let* F, G *be two* $\mathbf{Set}$ *endofunctors. An* F*-coalgebra* $(B,\ B \xrightarrow{\beta} \mathsf{F}\,B\ )$*, together with a* G*-algebra* $(A,\ A \xleftarrow{\alpha} \mathsf{G}\,A\ )$ *and a natural transformation* $\nu : \mathsf{F} \longrightarrow \mathsf{G}$ *define the hylomorphism* $[\![\beta, \nu, \alpha]\!]_{\mathsf{F},\mathsf{G}}$*, which is the morphism from* B *to* A *defined by the least fixpoint of equation*

$$f = \alpha \circ \nu_A \circ (\mathsf{F}\,f) \circ \beta \qquad \text{cf.} \qquad \begin{tikzcd} B \arrow[d, "f"'] \arrow[rr, bend left, "\beta"] & \mathsf{G}\,B \arrow[d, "\mathsf{G}\,f"] & \mathsf{F}\,B \arrow[l, "\nu_B"'] \arrow[d, "\mathsf{F}\,f"] \\ A & \mathsf{G}\,A \arrow[l, "\alpha"] & \mathsf{F}\,A \arrow[l, "\nu_A"] \end{tikzcd}$$

We write short-hand $[\![\beta, \alpha]\!]_{\mathsf{F}}$ *wherever* $\mathsf{F} = \mathsf{G}$ *and* $\nu = id$*.* $\Box$

Note that a catamorphism is a special case of a hylomorphism, for $\nu = id$ ($\mathsf{F} = \mathsf{G}$) and B the least fixpoint of $X \cong \mathsf{F}\,X$. Many useful programming schemata arise from the hylomorphism construct even where no initial algebras are involved. For instance, take $\beta = (! + gets) \circ =_{\emptyset}?$, “the” inverse [a] of the $[\ \underline{\emptyset}, puts\ ]$ algebra (p.46), on the same carrier 2^X, where

$$gets\,s \stackrel{\rm def}{=} \begin{array}{ll} let & x \in s \\ in & (x, s - \{x\}) \end{array} \tag{156}$$

Then $[\![\beta, \alpha]\!]_{\mathsf{F}}$ will be the standard “finite set parser” (for $\alpha = [\ \alpha_1, \alpha_2\ ]$):

$$f\,s \stackrel{\rm def}{=} \left\{ \begin{array}{rcl} s = \emptyset & \Rightarrow & \alpha_1 \\ \neg(s = \emptyset) & \Rightarrow & \begin{array}{ll} let & (x, r) = gets\,s \\ in & \alpha_2(x, f\,r) \end{array} \end{array} \right. \tag{157}$$

A similar “parser” will arise for browsing $X \rightharpoonup Y$ (of a different functor, though — see Remark 26 in the sequel) *etc. etc.*
Functorial morphisms arise when both A and B in Definition 18 are least fixpoints of endofunctors which share the same base structure, *cf.* for instance the following hylomorphical interpretation of $g^\star$:

$$\begin{tikzcd} C \arrow[d, "g"'] & C^\star \arrow[d, "g^\star"'] \arrow[rr, bend left, "(!+\langle head, tail\rangle)\circ =_{[\,]}?"] & 1 + D \times C^\star \arrow[d, "1+D\times g^\star"'] & 1 + C \times C^\star \arrow[l, "1+g\times id_{C^\star}"] \arrow[d, "1+C\times g^\star"] \\ D & D^\star & 1 + D \times D^\star \arrow[l, "{[\ nil, cons\ ]}"] & 1 + C \times D^\star \arrow[l, "1+g\times id_{D^\star}"] \end{tikzcd}$$

for natural transformation

$$\begin{tikzcd} A \arrow[d, "f"'] & 1 + C \times A \arrow[r, "1+g\times id_A"] \arrow[d, "1+C\times f"'] & 1 + D \times A \arrow[d, "1+D\times f"] \\ B & 1 + C \times B \arrow[r, "1+g\times id_B"'] & 1 + D \times B \end{tikzcd}$$

In the literature, such generic functorial morphisms are called *type functors* [BdM97]:

Definition 19 (Type functor). *Let* $\mathsf{B}(X, Y)$ *be a (regular* [b]*) bifunctor. Define* $\mathsf{F}\,Y \stackrel{\rm def}{=} \mathsf{B}(A, Y)$ *and* $\mathsf{G}\,Y \stackrel{\rm def}{=} \mathsf{B}(B, Y)$ *so that, for a given* $A \xrightarrow{f} B$ *,* $\mathsf{B}(f, id)$ *is an* F *to* G *natural transformation. A type functor* T *of base* B *is a functor which maps every object* A *into* $\mathsf{T}\,A = \mu Y.\mathsf{B}(A, Y)$ *and which maps every morphism* $A \xrightarrow{f} B$ *into hylomorphism* $\mathsf{T}\,f = [\![\iota_{\mathsf{F}}^{-1}, \mathsf{B}(f, id), \iota_{\mathsf{G}}]\!]_{\mathsf{F},\mathsf{G}}$*, the* ι *algebras being the corresponding initial algebras.* $\Box$

For an example, recall the type functor of base $1 + X \times Y^2$ discussed in Remark 17.

[a] Strictly speaking, this inverse is a relation and one should switch to the broader category $\mathbf{Rel}$ of [BdM97].
[b] See [Par98] for the definition of this class of functors. For our purposes here it suffices to know that polynomial functors are regular.

A Redundancy Ordering in $\mathbf{Set}$

Question

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

Starting Point

$\mathbf{Set}$-*theoretical* cardinality:

$$A \preceq B \stackrel{\text{def}}{=} \exists\, B \xrightarrow{f} A \,:\, f \text{ is surjective (=epic).}$$

Informal meanings of $A \preceq B$:

- A is "smaller" than B
- B 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 "witness" of 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,

$$A \preceq_{id_A} A$$

- $\preceq$ 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:

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

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

In fact:

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

for

$$fdp \stackrel{\rm def}{=} \lambda\rho.\forall (a,b),(a',b') \in \rho : (a = a' \Rightarrow b = b')$$

and

$$mkf \stackrel{\rm def}{=} \pi_1 \rightharpoonup \pi_2 \qquad (158)$$

Therefore:

$$A \rightharpoonup B \trianglelefteq^{fdp}_{mkf} 2^{A \times B} \qquad (159)$$

i π_1 is monic in $\{\rho \in 2^{A \times B} \mid fdp\,\rho\}$.

An Abstraction Invariant Ordering in Set

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

$$A \trianglelefteq B \stackrel{\rm def}{=} \exists S \subseteq B : A \preceq_f S. \qquad (160)$$

i.e.

$$A \trianglelefteq_f^{\phi} B \tag{161}$$

where ϕ denotes the characteristic function of S in B.

Generally speaking:

$$A \trianglelefteq_f^{\phi} B \stackrel{\text{def}}{=} \exists\, B \xrightarrow{f} A \;:\; f \textit{ is partial and surjective } \wedge \\ \phi\, b \Leftrightarrow b \in dom\, f$$

is the same as stating the following **abstraction invariant**:

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

Properties of $\trianglelefteq$:

- Reflexivity:

  $$A \trianglelefteq_{id_A}^{\underline{\mathsf{TRUE}}} A$$

  (NB: we will omit concrete invariant $\underline{\mathsf{TRUE}}$ 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

  $$\rho = \wedge \circ \langle \gamma, \phi \circ g \rangle \tag{162}$$

  that is, $\rho\, c = \gamma\, c \wedge \phi(g\, c)$.

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

  $$\trianglelefteq_i \begin{cases} f_i \\ \phi_i \end{cases}$$

  we have

  $$\trianglelefteq \begin{cases} f \stackrel{\text{def}}{=} \bigcirc_{i=1}^{n} f_i \\ \phi \stackrel{\text{def}}{=} \lambda x.\ \bigwedge_{i=n}^{1} \phi_i((\bigcirc_{j=i+1}^{n} f_j)x) \end{cases} \tag{163}$$

Example ($n = 3$):

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

The $\trianglelefteq$-Subcalculus

Most $\trianglelefteq$-results concern the reification of finite **partial** maps. Do laws (136) to (141) concerning exponentials (**total** maps) hold for partial maps?

- (136) holds:

$$
\begin{aligned}
0 \rightharpoonup A &\cong (A+1)^0 \\
&\cong 1 \qquad (164)
\end{aligned}
$$

- (137) becomes:

$$
\begin{aligned}
1 \rightharpoonup A &\cong (A+1)^1 \\
&\cong A+1 \qquad (165)
\end{aligned}
$$

- (140) holds:

$$
\begin{aligned}
(B+C) \rightharpoonup A &\cong (A+1)^{B+C} \\
&\cong (A+1)^B \times (A+1)^C \\
&\cong (B \rightharpoonup A) \times (C \rightharpoonup A) \qquad (166)
\end{aligned}
$$

- (141) becomes:

$$
\begin{aligned}
A \rightharpoonup 1 &\cong (1+1)^A \\
&\cong 2^A \qquad (167)
\end{aligned}
$$

still in the $\cong$-subcalculus.

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

- (138) "holds" at $\rightharpoonup$-level provided that $\trianglelefteq$ replaces $\cong$:

$$A \rightharpoonup B \times C \trianglelefteq (A \rightharpoonup B) \times (A \rightharpoonup C) \qquad (168)$$

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

$$(A \times B) \rightharpoonup C \trianglelefteq A \rightharpoonup (B \rightharpoonup C) \qquad (169)$$

Instead of the uncurrying law "symmetrical" *wrt.* (169) we have

$$A \rightharpoonup (B \rightharpoonup C) \trianglelefteq 2^A \times ((A \times B) \rightharpoonup C) \tag{170}$$

which is a particular case of

$$A \rightharpoonup D \times (B \rightharpoonup C) \trianglelefteq (A \rightharpoonup D) \times ((A \times B) \rightharpoonup C) \tag{171}$$

— let $D = 1$ and apply basic properties of $\times$ and (167).

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

$$(B + C)^A \not\cong B^A \times B^C$$

and yet we have the following map-decomposition law:

$$A \rightharpoonup (B + C) \trianglelefteq (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{172}$$

Another law of this kind is

$$A \rightharpoonup B \times C \quad \trianglelefteq \quad (A \times B) \rightharpoonup C \tag{173}$$

which induces concrete invariant $fdp \circ dom$ and generalizes (159).

Finally, recall law

$$2^A \quad \trianglelefteq_{elems} \quad A^\star \tag{174}$$

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

$$2^A \quad \trianglelefteq_{ran} \quad I\!N \rightharpoonup A \tag{175}$$

(refinement consists of indexing data)

$$A^\star \trianglelefteq I\!N \rightharpoonup A \tag{176}$$

(refinement consists of rendering sequence order explicit), and

$$A \rightharpoonup B \quad \trianglelefteq_{finv} \quad B \rightharpoonup 2^A_+ \tag{177}$$

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

REMARK 22: **A few side notes and some exercises**

Notes:

- In order to save SETS-expressions from too many parentheses, the following operator priorities are assumed: $^\star \;>\; \_^{\_} \;>\; \times \;>\; \rightharpoonup \;>\; +$. For instance, expression $X \rightharpoonup A^2 \times B^\star + C$ will abbreviate $(X \rightharpoonup ((A^2) \times (B^\star))) + C$.
- Note that mkf (158) is partial and that its domain of definition is captured by fdp. It can be defined alternatively by

  $$mkf \stackrel{\rm def}{=} \lambda\rho.\left(\begin{array}{c} a \\ the\{b \in B \mid\; a\,\rho\,b\} \end{array}\right)_{a \in 2^{\pi_1}\rho}$$

  where the is the operator

  $$the : 2^A \longrightarrow A \tag{178}$$

  which yields the unique element of a singleton set — $the\{a\} = a$ — and is otherwise undefined.
- The $\wedge$-connective in (162) should be regarded as non-strict on its second argument (*e.g.* FALSE $\wedge \perp =$ FALSE) due to the partiality of g.
- 2^A_+ in (177) means the non-empty elements of the powerset of A. Similarly, $(B \rightharpoonup C)_+$ will mean $B \rightharpoonup C$ minus the empty mapping.

Exercise 4 Show that, for $B \ncong 0$ and $C \ncong 0$, one has:

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

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

Exercise 5 Consider the following $\mathbf{Set}$-facts:

$$A^{B \cup X} \cong A^B \times A^X \;\Leftarrow\; B \cap X = \emptyset \tag{179}$$
$$A^n \cong A \times A^{n-1} \tag{180}$$
$$n \neq m \;\Rightarrow\; X^n \cap X^m = \emptyset \tag{181}$$

1. Which of these facts conveys our intuition that *no two sequences of different length can be the same sequence*?
2. Show that (179) can be drawn from previous $\mathbf{Set}$-facts.
3. $cons$ is a function which comes to mind in order to inject the right-hand side of (180) into its left-hand side. Which function would you suggest in the opposite direction? Justify.

□

Exercise 6 Wishing to prove the following fact, in $\mathbf{Set}$,

$$A^\star \times I\!\!N \cong A^\star \times A^\star$$

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

$$pack(l, l') \stackrel{\rm def}{=} (l +\!\!+ l', (length\, l) + 1)$$

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

Exercise 7 Show that (173) generalizes (159). **Hint**: let $C = 1$.
□

REMARK 23: **A little C-programming "trick" based on elementary $\unlhd$-results**

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

$$A \unlhd_{\lambda(i,a).a}^{\lambda x.x=i_1\ a} A + B \tag{182}$$
$$B \unlhd_{\lambda(i,b).b}^{\lambda x.x=i_2\ b} A + B \tag{183}$$

(injecting A into a variant record structure), and

$$A \unlhd_{\pi_1} A \times B \tag{184}$$
$$B \unlhd_{\pi_2} A \times B \tag{185}$$

(coupling A with another data structure B). Now, let us ask the following question: 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:

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

Therefore, not only

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

but also

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

So, data structure

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

is very convenient: it is a C-representation of both a product and the dual coproduct in Set.

Exercise 8 Select from above the fact which best supports the following triviality:

Every piece of data of type B can be represented by a "pointer to B" piece of data.

□

Exercise 9 Justify the calculations above and, in each case, record the corresponding abstraction invariant.

□

REMARK 24: **Details about laws (167), (168), (169) and (171)**

Isomorphism (167) is established by

$$A \rightharpoonup 1 \underset{set2fm}{\overset{dom}{\rightleftarrows}} 2^A \qquad \textit{cf. } (232): \qquad dom \circ set2fm = id \tag{186}$$

Concerning law (168), we have

$$A \rightharpoonup (B \times C) \quad \trianglelefteq_{\bowtie} \quad (A \rightharpoonup B) \times (A \rightharpoonup C)$$

where the finite mapping "join" operator

$$\sigma \bowtie \tau \quad \stackrel{\mathrm{def}}{=} \quad \begin{pmatrix} a \\ (\sigma\, a, \tau\, a) \end{pmatrix}_{a \in dom\, \sigma \cap dom\, \tau} \tag{187}$$

establishes how $\rightharpoonup$ "distributes" over product $\times$. The corresponding *theorem for free*

$$E \rightharpoonup (f \times g) \circ \bowtie = \bowtie \circ ((E \rightharpoonup f) \times (E \rightharpoonup g)) \tag{188}$$

— *cf.*

$$\begin{array}{ccc} A & & C \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ B & & D \end{array} \qquad \begin{tikzcd} (E \rightharpoonup A) \times (E \rightharpoonup C) \arrow[r, "\bowtie_{A,C}"] \arrow[d, "(E \rightharpoonup f) \times (E \rightharpoonup g)"'] & E \rightharpoonup (A \times C) \arrow[d, "E \rightharpoonup (f \times g)"] \\ (E \rightharpoonup B) \times (E \rightharpoonup D) \arrow[r, "\bowtie_{B,D}"'] & E \rightharpoonup (B \times D) \end{tikzcd}$$

— and facts

$$\begin{aligned} (A \rightharpoonup \pi_1) \circ \bowtie \quad &= \quad | \circ (id \times dom) && (189) \\ \dagger \circ (\bowtie \times \bowtie) \quad &= \quad \bowtie \circ \langle \dagger \circ (\pi_1 \times \pi_1), \dagger \circ (\pi_2 \times \pi_2) \rangle && (190) \end{aligned}$$

will prove especially useful in calculations involving $\bowtie$.
Concerning law (169), the expected currying map, from left to right,

$$\begin{aligned} pcurry \quad &: \quad (A \times B) \rightharpoonup C \longrightarrow A \rightharpoonup (B \rightharpoonup C) \\ pcurry\ f \quad &\stackrel{\mathrm{def}}{=} \quad \left(\begin{array}{c} a \\ \begin{pmatrix} \pi_2\, p \\ f\, p \end{pmatrix}_{p \in dom\, f \wedge (\pi_1\, p) = a} \end{array} \right)_{a \in 2^{\pi_1}(dom\, f)} \end{aligned} \tag{191}$$

is not surjective — think of the mappings in $A \rightharpoonup (B \rightharpoonup C)$ containing $(\quad)$ in their range. These will be lost by the opposite uncurrying map,

$$puncurry \quad \stackrel{\mathrm{def}}{=} \quad \lambda\sigma. \begin{pmatrix} (a, b) \\ (\sigma\, a) b \end{pmatrix}_{a \in dom\, \sigma \wedge b \in dom\, (\sigma\, a)} \tag{192}$$

which therefore is non-injective.
Note the following fact about $pcurry$:

$$(A \rightharpoonup dom) \circ pcurry \quad = \quad collect \circ dom \tag{193}$$

where $collect : 2^{A \times B} \longrightarrow A \rightharpoonup 2^B$ is defined by (216).
Law (171) is established by an elaboration of $\bowtie$:

$$\bowtie\!\!\!\!\!n \quad \stackrel{\mathrm{def}}{=} \quad \bowtie \circ \langle \pi_1, \dagger \circ ((A \rightharpoonup \underline{(\quad)}) \times pcurry) \rangle \tag{194}$$

The behaviour of abstraction $\sigma \bowtie\!\!\!\!\!n\ \sigma'$ is clearly non-injective: all $a \in A$ found in $\sigma' \in A \times B \rightharpoonup C$ which cannot be found in $\sigma \in A \rightharpoonup D$ will be lost in the abstraction process.

REMARK 25: **Some permutative laws and natural transformations**

A permutative law is one which can be regarded as stating a homomorphism (Definition 6) between two F-algebras of the identity functor (unary case) or of the product diagonal functor $\mathsf{F}\,X = X \times X$ (binary case).
Using diagrams, one will have

Unary

$$\begin{tikzcd} A \arrow[d, "f"] & A \arrow[l, "\alpha"'] \arrow[d, "f"] \\ B & B \arrow[l, "\beta"] \end{tikzcd}$$

$$f \circ \alpha \;=\; \beta \circ f$$

Binary

$$\begin{tikzcd} A \arrow[d, "f"] & A \times A \arrow[l, "\alpha"'] \arrow[d, "f\times f"] \\ B & B \times B \arrow[l, "\beta"] \end{tikzcd}$$

$$f \circ \alpha \;=\; \beta \circ (f \times f)$$

In pointwise notation, *e.g.* for the binary case, one has:

$$f(\alpha(x, y)) \;=\; \beta(f\,x, f\,y)$$

In the following collection of permutative laws, $|_S$ abbreviates curried map restriction (that is, $|_S = | \circ \langle id, \underline{S}\rangle$) and relational operators $update : 2^{A\times B} \times 2^{A\times B} \longrightarrow 2^{A\times B}$ and $sel : (\sum_{i\in\overline{n}} 2^{A_i}) \times 2^{A_1\times\ldots\times A_n} \longrightarrow 2^{A_1\times\ldots\times A_n}$ are defined resp. by (217) and (55):

$$\begin{aligned}
elems \circ +\!\!+ &= \cup \circ (elems \times elems) && (195)\\
length \circ +\!\!+ &= + \circ (length \times length) && (196)\\
dom \circ \cup &= \cup \circ (dom \times dom) && (197)\\
dom \circ \dagger &= \cup \circ (dom \times dom) && (198)\\
rng \circ \cup &= \cup \circ (rng \times rng) && (199)\\
\backslash_S \circ \cup &= \cup \circ (\backslash_S \times \backslash_S) && (200)\\
|_S \circ \dagger &= \dagger \circ (|_S \times |_S) && (201)
\end{aligned}$$

$$\begin{aligned}
\backslash_S \circ \dagger &= \dagger \circ (\backslash_S \times \backslash_S) && (202)\\
|_S \circ \bowtie &= \bowtie \circ (|_S \times |_S) && (203)\\
mkf \circ update &= \dagger \circ (mkf \times mkf) && (204)\\
dom \circ pcurry &= 2^{\pi_1} \circ dom && (205)\\
mkf \circ sel_{(i_1\,S)} &= |_S \circ mkf && (206)\\
mkf \circ 2^{A\times f} &= (A \rightharpoonup f) \circ mkf && (207)
\end{aligned}$$

Next, we record some natural transformations involving some useful operators:

$$\begin{tikzcd} A \arrow[d, "f"'] & (C \rightharpoonup A) \times (C \rightharpoonup A) \arrow[r, "\dagger_A"] \arrow[d, "(C\rightharpoonup f)\times(C\rightharpoonup f)"'] & (C \rightharpoonup A) \arrow[d, "(C\rightharpoonup f)"] \\ B & (C \rightharpoonup B) \times (C \rightharpoonup B) \arrow[r, "\dagger_B"'] & (C \rightharpoonup B) \end{tikzcd} \quad ((C \rightharpoonup f)) \circ \dagger = \dagger \circ ((C \rightharpoonup f) \times (C \rightharpoonup f)) \quad (208)$$

$$\begin{tikzcd} A \arrow[d, "f"'] & C \times D \rightharpoonup A \arrow[r, "pcurry_A"] \arrow[d, "C\times D\rightharpoonup f"'] & C \rightharpoonup (D \rightharpoonup A) \arrow[d, "C\rightharpoonup(D\rightharpoonup f)"] \\ B & C \times D \rightharpoonup B \arrow[r, "pcurry_B"'] & C \rightharpoonup (D \rightharpoonup B) \end{tikzcd} \quad (C \rightharpoonup (D \rightharpoonup f)) \circ pcurry = pcurry \circ (C \times D \rightharpoonup f) \quad (209)$$

$$\begin{tikzcd} A \arrow[d, "f"'] & C \rightharpoonup A \arrow[r, "|_{S\,A}"] \arrow[d, "C\rightharpoonup f"'] & C \rightharpoonup A \arrow[d, "C\rightharpoonup f"] \\ B & C \rightharpoonup B \arrow[r, "|_{S\,B}"'] & C \rightharpoonup B \end{tikzcd} \quad (C \rightharpoonup f) \circ |_S = |_S \circ (C \rightharpoonup f) \quad (210)$$

REMARK 26: **Laws (172) and (176) and some exercises**

The abstraction invariant implicit in law (172) consists of retrieve function

$$\overset{+}{\bowtie} \;\stackrel{\mathrm{def}}{=}\; \cup \circ ((A \rightharpoonup i_1) \times (A \rightharpoonup i_2))$$

coupled with concrete invariant

$$djd \;\stackrel{\mathrm{def}}{=}\; =_{\emptyset} \circ \cap \circ (dom \times dom)$$

$I\!N \rightharpoonup A$ in (176) is an adequate representation for $A^\star$ provided some total ordering criterion is assumed in $I\!N$, *e.g.* the $a < b$ ordering. The abstraction map can be expressed by hylomorphism

$$\begin{tikzcd} I\!N \rightharpoonup A \arrow[d, "{[\![\beta,\nu,[\,nil,cons\,]]\!]_{\mathsf{F},\mathsf{G}}}"'] \\ A^\star \end{tikzcd} \tag{211}$$

where $\mathsf{F}\,X = 1 + I\!N \times A \times X$, $\mathsf{G}\,X = 1 + A \times X$, $\nu = 1 + \langle \pi_2, \pi_3 \rangle$ and $\beta = (! + getMin) \circ =_{(\ )}?$, for

$$getMin\,\sigma \stackrel{\mathrm{def}}{=} \begin{array}{ll} let & m = Min(dom\,\sigma) \\ in & (m, \sigma\,m, \sigma \setminus \{m\}) \end{array}$$

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

$$Min\,S \stackrel{\mathrm{def}}{=} \left\{ \begin{array}{lcl} card\,S = 1 & \Rightarrow & the\,S \\ \neg(card\,S = 1) & \Rightarrow & \begin{array}{ll} let & x \in dom\,\sigma \\ & m = Min(S - \{x\}) \\ in & \left\{ \begin{array}{lcl} x < m & \Rightarrow & x \\ \neg(x < m) & \Rightarrow & m \end{array} \right. \end{array} \end{array} \right. \tag{212}$$

Exercise 10 Show that abstraction function (211) is equivalent to

$$f\,\sigma \;\stackrel{\mathrm{def}}{=}\; \left\{ \begin{array}{lcl} \sigma = (\ ) & \Rightarrow & [\,] \\ \neg(\sigma = (\ )) & \Rightarrow & \begin{array}{ll} let & m = Min(dom\,\sigma) \\ in & [\sigma\,m] +\!\!+ f(\sigma \setminus \{m\}) \end{array} \end{array} \right. \tag{213}$$

□

Exercise 11 Express Min (212) as a hylomorphism of functor $\mathsf{F}\,X = I\!N + I\!N \times X$.

□

Exercise 12 Complete the following hylomorphism description,

$$\begin{tikzcd} A \rightharpoonup B \arrow[r] \arrow[d, "{[\![\beta,\nu,[\,\alpha_1,\alpha_2\,]]\!]_{\mathsf{F},\mathsf{G}}}"'] & \cdots \\ C & \cdots \arrow[l] \end{tikzcd} \tag{214}$$

so that it is able to specify the standard finite mapping abstract "parser":

$$f\,\sigma \;\stackrel{\mathrm{def}}{=}\; \left\{ \begin{array}{lcl} \sigma = (\ ) & \Rightarrow & \alpha_1 \\ \neg(\sigma = (\ )) & \Rightarrow & \begin{array}{ll} let & a \in dom\,\sigma \\ in & \alpha_2(a, \sigma\,a, f(\sigma \setminus \{a\})) \end{array} \end{array} \right. \tag{215}$$

□

REMARK 27: **Abstraction invariant cookbook**

Abstraction invariant components f and ϕ in (161), 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 28 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

$$
\begin{array}{rcl}
f_{(126)} & = & \langle \pi_2, \pi_1 \rangle \\
f_{(127)} & = & \langle \pi_1 \circ \pi_1, \langle \pi_2 \circ \pi_1, \pi_2 \rangle\rangle \\
f_{(128)} & = & \pi_1 \\
f_{(129)} & = & [\, i_2, i_1 \,] \\
f_{(130)} & = & [\, [\, i_1, i_2 \circ i_1 \,], i_2 \circ i_2 \,] \\
f_{(131)} & = & i_1 \\
f_{(132)} & = & id_0 \\
f_{(133)} & = & [\, A \times i_1, A \times i_2 \,] \\
f_{(134)} & = & \lambda \sigma.(\sigma\, 1, \ldots, \sigma\, n) \\
f_{(135)} & = & \lambda x.x \\
f_{(136)} & = & \lambda x.\left(\quad\right) \\
f_{(137)} & = & \lambda a.\begin{pmatrix} NIL \\ a \end{pmatrix} \\
f_{(138)} & = & \lambda (x,y).\begin{pmatrix} a \\ (x\, a, y\, a) \end{pmatrix}_{a \in A} \\
f_{(139)} & = & uncurry \\
f_{(140)} & = & \cup \circ (A^{i_1} \times A^{i_2}) \\
f_{(141)} & = & \lambda x.\begin{pmatrix} a \\ NIL \end{pmatrix}_{a \in A} \\
f_{(142)} & = & \lambda \sigma.\begin{pmatrix} a \\ b \end{pmatrix}_{a \in A \wedge \sigma\, a = i_1\, b} \\
f_{(143)} & = & \lambda \sigma.\begin{pmatrix} c \\ \begin{pmatrix} \pi_1\, p \\ \sigma\, p \end{pmatrix}_{p \in dom\, \sigma \wedge \pi_2\, p = c} \end{pmatrix}_{c \in C} \\
f_{(159)} & = & mkf \\
f_{(164)} & = & \lambda x.\left(\quad\right) \\
f_{(165)} & = & \lambda x.\left\{ \begin{array}{rcl} x = i_1\, a & \Rightarrow & \begin{pmatrix} NIL \\ a \end{pmatrix} \\ \neg(x = i_1\, a) & \Rightarrow & \left(\quad\right) \end{array} \right. \\
f_{(166)} & = & \cup \circ (i_1 \rightharpoonup A) \times (i_2 \rightharpoonup A) \\
f_{(167)} & = & dom \\
f_{(168)} & = & \bowtie \\
f_{(169)} & = & puncurry \\
f_{(171)} & = & \bowtie_n \\
f_{(172)} & = & \overset{+}{\bowtie}
\end{array}
$$

REMARK 28: **Abstraction invariant cookbook (continuation)**

$$
\begin{array}{rcl}
f_{(173)} & = & \lambda\sigma.\left(\begin{array}{c} a \\ (b,\sigma((a,b))) \end{array}\right)_{(a,b)\in dom\,\sigma} \\
f_{(174)} & = & elems \\
f_{(177)} & = & finv \\
f_{(184)} & = & \pi_1 \\
f_{(185)} & = & \pi_2
\end{array}
$$

Concrete Invariants

$$
\begin{array}{rcl}
\phi_{(159)} & = & fdp \\
\phi_{(168)} & = & eqd \\
\phi_{(172)} & = & djd \\
\phi_{(173)} & = & fdp \circ dom \\
\phi_{(177)} & = & mdjfam
\end{array}
$$

Auxiliary Functions and Predicates

$$collect\,\rho \stackrel{\rm def}{=} \left(\begin{array}{c} a \\ \{x \in B \mid \ a\rho x\} \end{array}\right)_{a\in(2^{\pi_1})\rho} \tag{216}$$

$$update(x,y) \stackrel{\rm def}{=} \{(a,b) \mid \ (a,b) \in x \wedge a \not\in 2^{\pi_1}\,y\} \cup y \tag{217}$$

$$djd \stackrel{\rm def}{=} =_{\emptyset} \circ \cap \circ (dom \times dom) \tag{218}$$

$$dpi \stackrel{\rm def}{=} \supseteq \circ (dom \times 2^{\pi_1} \circ dom) \tag{219}$$

$$elems\,b \stackrel{\rm def}{=} \{b\,i \mid \ i \in length\,b\} \tag{220}$$

$$finv(\sigma) \stackrel{\rm def}{=} \bigcup_{b\in dom\,\sigma} \left(\begin{array}{c} a \\ b \end{array}\right)_{a\in\sigma\,b} \tag{221}$$

$$fdp\,r \stackrel{\rm def}{=} \forall t,t' \in r : \pi_1\,t = \pi_1\,t' \Rightarrow \pi_2\,t = \pi_2\,t' \tag{222}$$

$$mdjfam(\sigma) \stackrel{\rm def}{=} \forall b,b' \in dom\,\sigma : b \neq b' \Rightarrow (\sigma\,b) \cap \sigma\,b' = \emptyset \tag{223}$$

$$i_1\,a \stackrel{\rm def}{=} (1,a) \tag{224}$$

$$i_2\,a \stackrel{\rm def}{=} (2,a) \tag{225}$$

$$\sigma \bowtie \tau \stackrel{\rm def}{=} \left(\begin{array}{c} a \\ (\sigma\,a,\tau\,a) \end{array}\right)_{a\in dom\,\sigma\cap dom\,\tau} \tag{226}$$

$$\stackrel{+}{\bowtie} \stackrel{\rm def}{=} \cup \circ ((A \rightharpoonup i_1) \times (A \rightharpoonup i_2)) \tag{227}$$

$$\bowtie_n \stackrel{\rm def}{=} \bowtie \circ\langle\pi_1, \dagger \circ ((A \rightharpoonup \underline{(\quad)}) \times pcurry)\rangle \tag{228}$$

$$mkf\,\rho \stackrel{\rm def}{=} \left(\begin{array}{c} a \\ the(\{b \in B \mid \ a\,\rho\,b\}) \end{array}\right)_{a\in 2^{\pi_1}\ \rho} \tag{229}$$

$$puncurry \stackrel{\rm def}{=} \lambda\sigma.\left(\begin{array}{c} (a,b) \\ (\sigma\,a)b \end{array}\right)_{a\in dom\,\sigma\wedge b\in dom\,(\sigma\,a)} \tag{230}$$

$$rem((i,s),\tau) \stackrel{\rm def}{=} \{t \in \tau \mid \ \pi_i\,t \not\in s\} \tag{231}$$

$$set2fm\,r \stackrel{\rm def}{=} \left(\begin{array}{c} p \\ NIL \end{array}\right)_{p\in r} \tag{232}$$

$$uncurry\,t \stackrel{\rm def}{=} \left(\begin{array}{c} (b,c) \\ (t\,b)c \end{array}\right)_{b\in B\wedge c\in C} \tag{233}$$

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

Then:

- Laws such as (168,170,171,172) and (176) 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,
$$A \rightharpoonup B \trianglelefteq 2^{A \times B}$$
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 $A \rightharpoonup 2^B$ (A-indexed families of B-subsets) and we know that $2^B \trianglelefteq B^\star$ holds. Can we infer that

$$A \rightharpoonup 2^B \quad \trianglelefteq \quad A \rightharpoonup B^\star \tag{234}$$

holds? Such an inference will be allowed provided functor $\mathsf{F}\, X = A \rightharpoonup X$ is $\trianglelefteq$-monotone.

And, in fact — see theorem 3 and corollary 1 — we may state that, for every functor F we have introduced so far, if

$$A \trianglelefteq_f^\phi B$$

holds, then

$$\mathsf{F}\,A \trianglelefteq^{\mathsf{F}\,\phi}_{\mathsf{F}\,f} \mathsf{F}\,B$$

will hold.

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

$\mathsf{F}\,X$	$\mathsf{F}\,f$	$\mathsf{F}\,\phi$
X	f	ϕ
C	1_C	$\lambda c.\mathsf{TRUE}$
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	$\begin{array}{rcl} f^C & = & \lambda\sigma.f \circ \sigma \\ & = & \lambda\sigma.\left(\begin{array}{c} c \\ f(\sigma\,c) \end{array}\right)_{c \in C} \end{array}$	$\phi^C = \lambda\sigma.\forall c \in C : \phi(\sigma\,c)$
$C \rightharpoonup X$	$\begin{array}{rcl} C \rightharpoonup f & = & \lambda\sigma.f \circ \sigma \\ & = & \lambda\sigma.\left(\begin{array}{c} c \\ f(\sigma\,c) \end{array}\right)_{c \in dom\,\sigma} \end{array}$	$C \rightharpoonup \phi = \lambda\sigma.\forall x \in rng(\sigma) : \phi\,x$
$\mathsf{G}\,X \times \mathsf{H}\,X$	$\mathsf{G}\,f \times \mathsf{H}\,f$	$\lambda(x,y).(\mathsf{G}\,\phi)x \wedge (\mathsf{H}\,\phi)y$
$\mathsf{G}\,X + \mathsf{H}\,X$	$\mathsf{G}\,f + \mathsf{H}\,f$	$\lambda x.\left\{\begin{array}{lcl} x = i_1\,a & \Rightarrow & (\mathsf{G}\,\phi)a \\ x = i_2\,b & \Rightarrow & (\mathsf{H}\,\phi)b \end{array}\right.$
$\mathsf{G}\,X + 1$	$\lambda x.\left\{\begin{array}{lcl} x = i_2\,NIL & \Rightarrow & x \\ x = i_1\,a & \Rightarrow & i_1((\mathsf{G}\,f)a) \end{array}\right.$	$\lambda x.\left\{\begin{array}{lcl} x = i_2\,NIL & \Rightarrow & \mathsf{TRUE} \\ x = i_1\,a & \Rightarrow & (\mathsf{G}\,\phi)a \end{array}\right.$

Thus,

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

Examples of Structural Calculation

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

REMARK 29: **Structural abstraction invariant calculation and $\trianglelefteq$-monotonicity (1)**

This remarks contains the basic theory which underpins structural abstraction invariant calculation. First of all, we need to be more categorial about what we mean by (161), that is, by $A \trianglelefteq^{\phi}_{f} B$: we mean the existence of an epic $A \xleftarrow{f} B_\phi$ where, in set-theoretical terms, B_ϕ denotes set $\{b \in B \mid \phi\, b\}$ but, more generally, it is the "greatest" object which (up to isomorphism) makes the following diagram to commute, where i is monic [a]:

$$\begin{tikzcd} B_\phi \arrow[r, "!_{B_\phi}"] \arrow[d, "i"'] & 1 \arrow[d, "\underline{\text{TRUE}}"] \\ B \arrow[r, "\phi"'] & 2 \end{tikzcd} \qquad \phi(i\, b) = \text{TRUE} \tag{235}$$

In category theory one will say that B_ϕ is the *pullback* of the "corner" of arrows made by ϕ and $\underline{\text{TRUE}}$. We may express also by a pullback the effect of forcing a map $B \xleftarrow{g} C$ to produce its output "inside" B_ϕ, that is, of satisfying invariant ϕ on its output:

$$\begin{tikzcd} C_{\phi \circ g} \arrow[r, "g'"] \arrow[d, "i'"'] & B_\phi \arrow[r, "!_{B_\phi}"] \arrow[d, "i"] & 1 \arrow[d, "\underline{\text{TRUE}}"] \\ C \arrow[r, "g"'] & B \arrow[r, "\phi"'] & 2 \end{tikzcd}$$

Then $\trianglelefteq$-transitivity will be expressed by

$$\begin{tikzcd} C_{\wedge \circ \langle \gamma, \phi \circ f \rangle} \arrow[r, "g'"] \arrow[d, "i'"'] & B_\phi \arrow[r, "f"] \arrow[d, "i"] & A \arrow[r, "!_A"] & 1 \arrow[d, "\underline{\text{TRUE}}"] \\ C_\gamma \arrow[r, "g"'] & B \arrow[rr, "\phi"'] & & 2 \end{tikzcd}$$

for g and f both epic.
The following theorem is central to abstraction invariant structural calculation [b].

Theorem 3 (Functorial $\trianglelefteq$-monotonicity in $\mathbf{Set}$). *Let* F *be a* $\mathbf{Set}$ *endofunctor. If*

$$A \trianglelefteq^{\phi}_{f} B \tag{236}$$

holds, then

$$\mathsf{F}\,A \trianglelefteq^{\mathsf{F}\,\phi}_{\mathsf{F}\,f} \mathsf{F}\,B \tag{237}$$

Proof: *(236) means the existence of epic* $A \xleftarrow{f} B_\phi$ *. In* $\mathbf{Set}$*, all functors preserve epics* [c]*, that is, epic* $\mathsf{F}A \xleftarrow{\mathsf{F}f} \mathsf{F}(B_\phi)$ *exists. Finally, the fact that* $\mathsf{F}(B_\phi) \cong B_{(\mathsf{F}\,\phi)}$ *holds is shown in Theorem 4.* $\square$

The scope of application of this result is wide and extends to bifunctors, *cf. e.g.* Corollary 1. The table of page p.64 provides an account of abstraction invariant synthesis for the most common functors. For instance, back to (234), it is now straightforward to calculate, from $2^B \trianglelefteq_{elems} B^\star$ the implied abstraction function [d]:

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

[a] i performs the "inclusion" of B_ϕ into B.
[b] This result is due to [Bar97, Rod97]. A restricted version of it was presented by [Oli92].
[c] This is not true in every category, see *e.g.* [McL95].
[d] We are ignoring concrete invariants here because they are trivially the everywhere TRUE predicate.

REMARK 30: **Structural abstraction invariant calculation and $\trianglelefteq$-monotonicity (2)**

Theorem 4 (Functorial subobjects). *Let* F *be a* $\mathbf{Set}$ *endofunctor and consider* B_ϕ *from diagram (235). Then* $\mathsf{F}(B_\phi) \cong B_{(\mathsf{F}\,\phi)}$ *holds.*

Proof [a]*: Apply* F *to diagram (235) and introduce* $(2,\ 2 \xleftarrow{\ \square_{\mathsf{F}}\ } \mathsf{F}\,2\ )$*, the* F*-algebra which reduces an* F*-structure of Booleans to a single Boolean via the* $\wedge$ *logical connective:*

$$\begin{tikzcd}
\mathsf{F}(B_\phi) \arrow[r, "\mathsf{F}(!_{B_\phi})"] \arrow[d, "\mathsf{F}\,i"'] & \mathsf{F}\,1 \arrow[d, "\mathsf{F}\,\underline{\mathrm{TRUE}}"] & \\
\mathsf{F}\,B \arrow[r, "\mathsf{F}\,\phi"'] & \mathsf{F}\,2 \arrow[r, "\square_{\mathsf{F}}"'] & 2
\end{tikzcd}$$

This raises pullback square

$$\begin{tikzcd}
\mathsf{F}(B_\phi) \cong B_{\square_{\mathsf{F}} \circ (\mathsf{F}\,\phi)} \arrow[rr, "!_{\mathsf{F}(B_\phi)}"] \arrow[d, "\mathsf{F}\,i"'] & & 1 \arrow[d, "\underline{\mathrm{TRUE}}"] \\
\mathsf{F}\,B \arrow[r, "\mathsf{F}\,\phi"'] & \mathsf{F}\,2 \arrow[r, "\square_{\mathsf{F}}"'] & 2
\end{tikzcd}$$

So, in fact, $\mathsf{F}(B_\phi) \cong B_{\square_{\mathsf{F}} \circ (\mathsf{F}\,\phi)}$*. To avoid notation cluttering we abbreviate* $B_{\square_{\mathsf{F}} \circ (\mathsf{F}\,\phi)}$ *to* $B_{\mathsf{F}\,\phi}$*.* $\square$

See instances of $\mathsf{F}\,\phi$ — that is: $\square_{\mathsf{F}} \circ (\mathsf{F}\,\phi)$ — in corollaries 1 and 2.

Corollary 1 ($\trianglelefteq$-Monotonicity of $\mathbf{Set}$-constructs). *The* $\mathbf{Set}$*-constructs* $\times$ *and* $+$ *are monotone* wrt. *the* $\trianglelefteq$*-ordering (160); exponentials require isomorphical exponents. That is, given* $\mathbf{Set}$*-objects* A*,* B*,* X*,* Y*,* M*,* N *such that*

$$A \trianglelefteq_f^\phi X,\ B \trianglelefteq_g^\varphi Y,\ M \cong_h N \tag{238}$$

hold, then facts

$$A \times B \quad \trianglelefteq_{f \times g}^{\phi \times \varphi} \quad X \times Y \tag{239}$$
$$A + B \quad \trianglelefteq_{f + g}^{\phi + \varphi} \quad X + Y \tag{240}$$
$$A^M \quad \trianglelefteq_{f^h}^{\phi^N} \quad X^N \tag{241}$$

hold.
Proof*:*

1. *Equation (239): Theorem 3 for bifunctor* $\times$*. Recall that the product of two functions* f *and* g *is "parallel application" (22) and the product of two predicates* ϕ *and* φ *is "parallel conjunction"*

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

2. *Equation (240): Theorem 3 for bifunctor* $+$*. Recall that the sum of two functions* f *and* g *is given by (26) and the sum of two predicates* ϕ *and* φ *is*

$$\left\{ \begin{array}{lcl} (\phi + \varphi) i_1\, a & = & \phi\, a \\ (\phi + \varphi) i_2\, b & = & \varphi\, b \end{array} \right.$$

3. *Equation (241): Theorem 3 for exponential bifunctor, restricted to monics on the exponent. Thus* M *and* N *are isomorphic, and, for* $\sigma \in X^N$ *and* h *the isomorphism between* M *and* N *(238):*

$$\begin{array}{lcl} f^h\, \sigma & = & f \circ \sigma \circ h^{-1} \\ \phi^M\, \sigma & = & \forall n \in N : \phi(\sigma\, n) \end{array}$$

$\square$

[a]From [Bar97].

We know that ...

$$
\begin{aligned}
A^\star &= \bigcup_{n=0}^{\infty} A^n \\
&\cong \sum_{n=0}^{\infty} A^n \\
&\cong 1 + A + A^2 + \ldots
\end{aligned}
\tag{242}
$$

cf. (144), (136) and (137), respectively.
Let us introduce two auxiliary variables L and N,

$$
\begin{aligned}
N &= A + A^2 + \ldots \\
L &= 1 + N
\end{aligned}
\tag{243}
$$

so that we have, by substitution,

$$
A^\star \cong L
\tag{244}
$$

and

$$
\begin{aligned}
N &= A + A^2 + \ldots \\
&\cong A \times 1 + A \times A + A \times A^2 + \ldots \\
&\cong A \times (1 + A + A^2 + \ldots) \\
&\cong A \times L
\end{aligned}
\tag{245}
$$

where we resorted to laws (137), (180) e (136), to the finitary version of (133), and where step (245) was obtained by "folding" over step (242). In summary,

$$
\begin{aligned}
A^\star \cong\ & L \\
& \mathit{where} \quad \begin{aligned} L &= 1 + N \\ N &\cong A \times L \end{aligned}
\end{aligned}
\tag{246}
$$

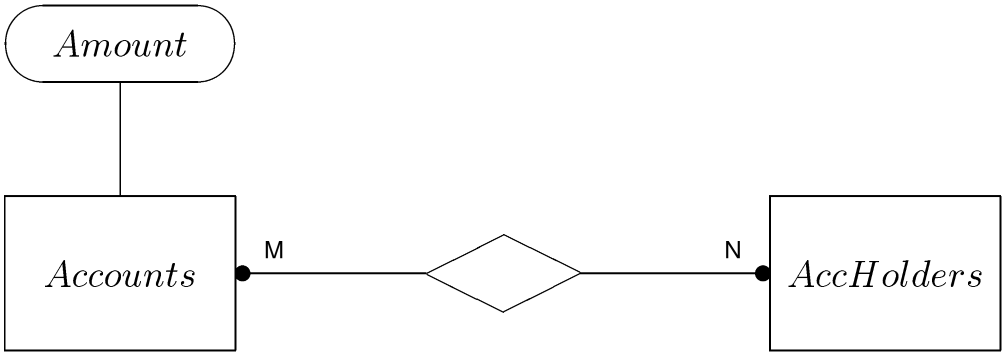
Recalling the table of p.15, L (246) can be encoded thus, in *e.g.* PASCAL:

```
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 a Toy Banking Application

Consider the following E-R diagram expressing some basic information structure for a toy "bank account management system" ($BAMS$):



Such a starting-step can be tolerated by "converting" the diagram into SETS notation,

$$\left\{\begin{array}{rcl} BAMS & \cong & Accounts \times AccHolders \\ Accounts & \cong & AccId \rightharpoonup (2^{AHoId} \times Amount) \\ AccHolders & \cong & AHoId \rightharpoonup AccHolder \end{array}\right.$$

using a few rules omitted here but reported elsewhere [OC93, Rod93].

$Accounts$ can be recognized as VDM-SL fragment AMS in (1).

Since our calculation method is structural,

- while somebody else deals with the $AccHolders$ component,
- we choose to concentrate on $Accounts$,

$$Accounts \;=\; AccId \rightharpoonup (2^{AHoId} \times Amount) \qquad (247)$$

which we want to implement in a relational database.

$$
\begin{array}{rcl}
Accounts & = & AccId \rightharpoonup (2^{AHoId} \times Amount) \\
& \cong_1 & \quad \{ \text{ fact (126)} \} \\
& & AccId \rightharpoonup (Amount \times 2^{AHoId}) \\
& \cong_2 & \quad \{ \text{ fact (167)} \} \\
& & AccId \rightharpoonup (Amount \times (AHoId \rightharpoonup 1)) \\
& \trianglelefteq_3 & \quad \{ \text{ fact (171)} \} \\
& & (AccId \rightharpoonup Amount) \times ((AccId \times AHoId) \rightharpoonup 1) \\
& \cong_4 & \quad \{ \text{ fact (167)} \} \\
& & (AccId \rightharpoonup Amount) \times (2^{AccId \times AHoId}) \\
& \trianglelefteq_5 & \quad \{ \text{ fact (159)} \} \\
& & \underbrace{2^{AccId \times Amount}}_{\substack{\text{amounts} \\ \text{table}}} \times \underbrace{2^{AccId \times AHoId}}_{\substack{\text{account-holders} \\ \text{table}}} \\
& = & Accounts_5
\end{array}
$$

where:

$$
\begin{array}{ll}
\cong_1 & \{ \; f_1 = AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle \\
\cong_2 & \{ \; f_2 = AccId \rightharpoonup (id \times dom) \\
\trianglelefteq_3 & \{ \; f_3 = \bowtie \\
\cong_4 & \{ \; f_4 = id \times set2fm \\
\trianglelefteq_5 & \left\{ \begin{array}{l} f_5 = mkf \times id \\ \phi_5 = fdp \times \underline{\text{TRUE}} \end{array} \right.
\end{array}
$$

Illustration:

$Accounts$ object

AccId	AHoIds	Amount
0636024550431	J N Oliveira + L.S. Barbosa	3000
0636024000999	Neves, Fernando L. + J N Oliveira	9000
0636099999000	J.J.A.	9500
0636055566677		10

represented *e.g.* by:

AccId	Amount
0636024550431	3000
0636024000999	9000
0636099999000	9500
0636055566677	10

\+

AccId	AHoId
0636024550431	J N Oliveira
0636024550431	L.S. Barbosa
0636024000999	Neves, Fernando L.
0636024000999	J N Oliveira
0636099999000	J.J.A.

Overall Abstraction Invariant Synthesis

ABSTRACTION FUNCTION:

$$
\begin{aligned}
f &= f_1 \circ f_2 \circ f_3 \circ f_4 \circ f_5 \\
&= (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle)\circ \\
&\quad (AccId \rightharpoonup (id \times dom))\circ \bowtie_n \circ(id \times set2fm)\circ \\
&\quad (mkf \times id) \\
&= \qquad \{ \text{ See Remark 31} \} \\
&\quad (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle)\circ \\
&\quad \bowtie \circ\langle \pi_1, \dagger \circ ((AccId \rightharpoonup \underline{\emptyset}) \times collect)\rangle\circ \\
&\quad (mkf \times id)
\end{aligned}
\tag{248}
$$

CONCRETE INVARIANT:

$$
\begin{aligned}
\phi(\rho, \gamma) &= \phi_5(\rho, \gamma) \wedge \\
&\quad \phi_4(f_5(\rho, \gamma)) \wedge \\
&\quad \phi_3(f_4(f_5(\rho, \gamma))) \wedge \\
&\quad \phi_2(f_3(f_4(f_5(\rho, \gamma)))) \wedge \\
&\quad \phi_1(f_2(f_3(f_4(f_5(\rho, \gamma))))) \\
&= \phi_5(\rho, \gamma) \\
&= fdp\, \rho \wedge \mathsf{TRUE} \\
&= fdp\, \rho
\end{aligned}
$$

imposing a functional dependence on the “amounts table” ρ (=the balance of evey account is not ambiguous).

REMARK 31: **The rôle of natural transformations in abstraction function synthesis**

Steps $\cong_2$, $\trianglelefteq_3$ and $\cong_4$ in the $Accounts_5$ calculation, captured by generic abstraction function

$$\begin{array}{c} A \rightharpoonup C \times 2^B \\ \uparrow {\scriptstyle (A \rightharpoonup (id \times dom)) \circ \bowtie \circ (id \times set2fm)} \\ (A \rightharpoonup C) \times 2^{A \times B} \end{array} \tag{249}$$

are very common in relational data refinement. The expansion of this abstraction function

$$(A \rightharpoonup (id \times dom)) \circ \underbrace{\bowtie \circ \langle \pi_1, \dagger \circ ((A \rightharpoonup \underline{(\ \ )}) \times pcurry) \rangle}_{\bowtie} \circ (id \times set2fm) \tag{250}$$

is, however, too clumsy an expression to be useful in calculations. In the reasoning which follows we show how calculate a simpler version of it. Fact

$$f \circ \underline{c} \quad = \quad \underline{f\, c} \tag{251}$$

will be taken into account. We start by abbreviating $A \rightharpoonup \underline{(\ \ )}$ to α_1 and $\dagger \circ (\alpha_1 \times pcurry)$ to α:

$$\begin{array}{ll}
& f_{(249)} \\
= & (A \rightharpoonup (id \times dom)) \circ \bowtie \circ \langle \pi_1, \alpha \rangle \circ (id \times set2fm) \\
& \quad \{ \text{ by } \times\text{-fusion (42) } \} \\
= & (A \rightharpoonup (id \times dom)) \circ \bowtie \circ \langle \pi_1 \circ (id \times set2fm), \alpha \circ (id \times set2fm) \rangle \\
& \quad \{ \text{ by (32), identity of composition and expansion of } \alpha \ \} \\
= & (A \rightharpoonup (id \times dom)) \circ \bowtie \circ \langle \pi_1, \dagger \circ (\alpha_1 \times pcurry) \circ (id \times set2fm) \rangle \\
& \quad \{ \text{ bifunctoriality of } \times \text{ (11) and id. of composition } \} \\
= & (A \rightharpoonup (id \times dom)) \circ \bowtie \circ \langle \pi_1, \dagger \circ (\alpha_1 \times pcurry \circ set2fm) \rangle \\
& \quad \{ \bowtie \text{ is binatural transformation (188) } \} \\
= & \bowtie \circ ((A \rightharpoonup id) \times (A \rightharpoonup dom)) \circ \langle \pi_1, \dagger \circ (\alpha_1 \times pcurry \circ set2fm) \rangle \\
& \quad \{ \text{ by } \times\text{-absorption (43) } \} \\
= & \bowtie \circ \langle (A \rightharpoonup id) \circ \pi_1, (A \rightharpoonup dom) \circ \dagger \circ (\alpha_1 \times pcurry \circ set2fm) \rangle \\
& \quad \{ \text{ by (6) for } \mathsf{F}\, X = A \rightharpoonup X \text{ and } \dagger \text{ is natural transformation (208) } \} \\
= & \bowtie \circ \langle \pi_1, \dagger \circ ((A \rightharpoonup dom) \times (A \rightharpoonup dom)) \circ (\alpha_1 \times pcurry \circ set2fm) \rangle \\
& \quad \{ \text{ bifunctoriality of } \times \text{ (11) } \} \\
= & \bowtie \circ \langle \pi_1, \dagger \circ ((A \rightharpoonup dom) \circ \alpha_1) \times (A \rightharpoonup dom) \circ (pcurry \circ set2fm) \rangle \\
& \quad \{ \text{ (7) for } \mathsf{F}\, X = A \rightharpoonup X, \text{ (251), fact } dom\ (\ \ ) = \emptyset \text{ and (193) } \} \\
= & \bowtie \circ \langle \pi_1, \dagger \circ (A \rightharpoonup \underline{\emptyset}) \times (collect \circ dom \circ set2fm) \rangle \\
& \quad \{ \text{ by (186) and id. of composition } \} \\
= & \bowtie \circ \langle \pi_1, \dagger \circ ((A \rightharpoonup \underline{\emptyset}) \times collect) \rangle
\end{array}$$

In summary, we can rely on the following compact version

$$\bowtie \circ \langle \pi_1, \dagger \circ ((A \rightharpoonup \underline{\emptyset}) \times collect) \rangle \tag{252}$$

of abstraction function (249).

The important rôle of functoriality and natural transformation (*"theorems for free"*) in the reasonings we have carried above should have be apparent to the reader.

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

- invent $Accounts_5$, 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

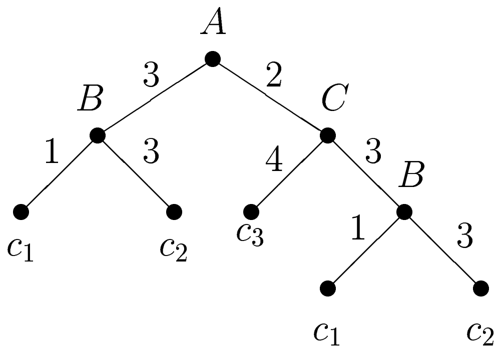
Consider the following data model for a “toy factory” production database:

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

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 32 and 33 for details):

— $Stock$ e $Pricelist$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{Stock \times Pricelist} \\
\cong & (\#Unit \rightharpoonup \mathbb{N}) \times (\#Comp \rightharpoonup \mathbb{N}) \\
\cong & ((\#Comp + \#Equip) \rightharpoonup \mathbb{N}) \times (\#Comp \rightharpoonup \mathbb{N}) \\
\cong & (\#Comp \rightharpoonup \mathbb{N}) \times (\#Equip \rightharpoonup \mathbb{N}) \times (\#Comp \rightharpoonup \mathbb{N}) \\
\cong & (\#Comp \rightharpoonup \mathbb{N})^2 \times (\#Equip \rightharpoonup \mathbb{N}) \\
\unlhd & 2^{2 \times \#Comp \times \mathbb{N}} \times 2^{\#Equip \times \mathbb{N}}
\end{array}
$$

— $Equip$:

$$
\begin{array}{rcl}
Equip & \cong & \#Equip \rightharpoonup (\#Unit \rightharpoonup \mathbb{N})_+ \\
& \cong & (\#Equip \times \#Unit) \rightharpoonup \mathbb{N} \\
& \cong & (\#Equip \times \#Comp + \#Equip \times \#Equip) \rightharpoonup \mathbb{N} \\
& \cong & (\#Equip \times \#Comp \rightharpoonup \mathbb{N}) \times (\#Equip \times \#Equip \rightharpoonup \mathbb{N}) \\
& \unlhd & 2^{\#Equip \times \#Comp \times \mathbb{N}} \times 2^{\#Equip \times \#Equip \times \mathbb{N}}
\end{array}
$$

By compound reasoning we will then have:

$$PPD \unlhd RPPD$$

REMARK 32: **Detailed calculation of $RPPD$ from PPD (1)**

Calculation steps:

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

Step by step abstraction invariants [a]:

$$
\begin{array}{ll}
\cong_1 & \{\; f_1 = f_{(166)} \times id \times pcurry \\
\cong_2 & \{\; f_2 = id \times id \times (f_{(133)} \rightharpoonup I\!N) \\
\cong_3 & \{\; f_3 = id \times id \times f_{(166)} \\
\cong_4 & \{\; f_4 = \lambda((\rho_1, \rho_2), \rho_3, \rho_4, \rho_5) \cdot ((\rho_1, \rho_3), \rho_2, (\rho_4, \rho_5)) \\
\trianglelefteq_5 & \left\{ \begin{array}{l} f_5 = id \times mkf \times mkf \times mkf \\ \phi_5 = \underline{\mathrm{TRUE}} \times fdp \times fdp \times fdp \end{array} \right. \\
\cong_6 & \{\; f_6 = f_{(143)} \times id \times 2^j \times 2^j \\
\trianglelefteq_7 & \left\{ \begin{array}{l} f_7 = (mkf \circ 2^j) \times id \times id \times id \\ \phi_7 = (fdp \circ 2^j) \times \Pi_{i=1}^{3} \underline{\mathrm{TRUE}} \end{array} \right.
\end{array}
$$

[a] Omitting "everywhere TRUE" concrete invariants for the sake of brevity. Variable j abbreviates $\lambda(a, b, c).((a, b), c)$ and $pcurry$ is given by (191).

REMARK 33: **Detailed calculation of $RPPD$ from PPD (2)**

Overall abstraction invariant synthesis:

- Abstraction function:

$$\begin{aligned} f &= f_1 \circ f_2 \circ f_3 \circ f_4 \circ f_5 \circ f_6 \circ f_7 \\ &= (f_{(166)} \times id \times (pcurry \circ (f_{(133)} \rightharpoonup I\!N) \circ f_{(166)})) \circ \\ &\quad f_4 \circ \\ &\quad ((f_{(143)} \circ mkf \circ 2^j) \times id \times mkf \circ 2^j \times mkf \circ 2^j) \end{aligned}$$

- Concrete invariant:

$$\phi = (fdp \circ 2^j) \times fdp \times (fdp \circ 2^j) \times (fdp \circ 2^j) \tag{253}$$

where j abbreviates

$$\langle\langle \pi_1, \pi_2 \rangle, \pi_3 \rangle \tag{254}$$

in the context of natural isomorphism

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

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 (253) into:

$$\begin{aligned} (fdp \circ 2^j)r &= fdp\,\{((\pi_1\, t, \pi_2\, t), \pi_3\, t) \mid t \in r\} \\ &= \forall t, t' \in r : \pi_1\, t = \pi_1\, t' \wedge \pi_2\, t = \pi_2\, t' \Rightarrow \pi_3\, t = \pi_3\, t' \end{aligned}$$

and so on.

Exercise 13 Let A^+ denote set

$$A^+ = \bigcup_{n \geq 1} A^{\overline{n}}$$

A^+ is a fixpoint of functor $\mathsf{F}\,X = A + A \times X$. Characterize the corresponding initial algebra $(A^+,\ A^+ \xleftarrow{\iota} \mathsf{F}\,A^+\ )$.
□

Exercise 14 Consider the following data model for *priority queues*,

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

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,

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

subject to concrete invariant

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

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 $\mathbf{Set}$ operators, recall what we have said about the "category theoretic" operators of $\mathbf{Set}$ and the properties "hidden" in their diagrams.

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

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

and where

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

Recursive Data Model Reification

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

$$X \cong 1 + A \times X \textit{ /* finite lists on } A\textit{, recall (13) */} \tag{255}$$

$$X \cong 1 + A \times X^2 \textit{ /* binary trees on } A \textit{ */} \tag{256}$$

$$X \cong 1 + A \times X^{\star} \textit{ /* generic trees on } A \textit{ */}$$

$$X \cong V + A \times X^{\star} \textit{ /* formal terms on } V \textit{ and } A \textit{ */}$$

etc.

What can we do about refining a recursive data model?

Motivation

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

$$S1 = [Bt1]$$
$$Bt1 :: S1\ Key\ Data\ S1$$

that is, an instance of recursive definition (256) in $\mathbf{Set}$, 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":

$$\begin{array}{l} S2 :: ROOT : [Ptr]\ ARRAY : Ptr \xrightarrow{m} Node2 \\ Node2 :: [Ptr]\ Key\ Data\ [Ptr] \end{array} \tag{257}$$

that is,

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

for an abstract domain $Ptr \cong I\!N$ of "pointers".

Moreover, there is a need for a ...

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

Question

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

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

An elegant way of doing it is to regard F as a polynomial (endo)functor in $\mathbf{Set}$ — recall Remark 17.

A Result for Recursion Removal

Let

$$(X_0,\ X_0 \xleftarrow{\delta} \mathsf{F}X_0\ )$$

be a fixpoint solution of

$$X \cong \mathsf{F}\,X \tag{258}$$

in $\mathbf{Set}$.

Reification step:

$$X_0 \trianglelefteq_f^\phi K \times (K \rightharpoonup \mathsf{F}\, K) \tag{259}$$

for K a domain of “pointers” such that $K \cong I\!N$ (see Theorem 5).

Abstraction Function f

We need to define a surjection

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

that is, to find appropriate f and ϕ.

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

$$\begin{tikzcd} K \arrow[d, "\sigma"] \\ \mathsf{F}\, K \end{tikzcd}$$

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:

$$\begin{tikzcd} X_0 & K \arrow[l, "f_\sigma"'] \arrow[d, "\sigma"] \\ & \mathsf{F}\, K \end{tikzcd}$$

The diagram can immediately completed,

$$\begin{tikzcd} X_0 & K \arrow[l, "f_\sigma"'] \arrow[d, "\sigma"] \\ \mathsf{F}\, X_0 \arrow[u, "\delta"] & \mathsf{F}\, K \arrow[l, "\mathsf{F}\, f_\sigma"] \end{tikzcd}$$

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

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

or

$$\begin{array}{rcl} f & : & K \times (K \rightharpoonup \mathsf{F}\, K) \longrightarrow X_0 \\ f(k, \sigma) & \stackrel{\mathrm{def}}{=} & \delta((\mathsf{F}\, f_\sigma)(\sigma\, k)) \end{array} \tag{260}$$

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

Concrete Invariant ϕ

Face pointer undefinedness in case σ is partial — $\sigma\, k$ in (260) is undefined wherever $k \notin dom\, \sigma$ and other pointers k' in the range of σ, reachable from k, may be in the same situation.

- Pointer reachability — transitive closure of

$$k_1 <_{\mathsf{F}} k \stackrel{\rm def}{=} k \in dom\, \sigma \wedge k_1 \in_{\mathsf{F}} \sigma\, k \tag{261}$$

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

$$\begin{array}{rcl} k \in_C x & \stackrel{\rm def}{=} & \mathsf{FALSE} \\ k \in_{\lambda X.X} x & \stackrel{\rm def}{=} & k = x \\ k \in_{\mathsf{F}\times\mathsf{G}} (x,y) & \stackrel{\rm def}{=} & k \in_{\mathsf{F}} x \vee k \in_{\mathsf{G}} y \\ k \in_{\mathsf{F}+\mathsf{G}} x & \stackrel{\rm def}{=} & \left\{ \begin{array}{lcl} k \in_{\mathsf{F}} y & \Leftarrow & x = i_1\, y \\ k \in_{\mathsf{G}} z & \Leftarrow & x = i_2\, z \end{array} \right. \end{array} \tag{262}$$

  that is

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

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

$$\begin{array}{rcll} \phi(k,\sigma) & \stackrel{\rm def}{=} & let & P = \{k\} \cup \{k' \in K \mid k' <^+_{\mathsf{F}} k\} \\ & & in & P \subseteq dom\, \sigma \wedge \\ & & & \forall \emptyset \subset C \subseteq P : \exists m \in C : \forall k' <_{\mathsf{F}} m : k' \notin C \end{array} \tag{263}$$

The proof of surjectiveness of f (260) for $(X_0,\ \mathsf{F}\, X_0 \xrightarrow{\delta} X_0\,)$ a finite or denumerable fixpoint of F, is a lengthy one! [Jou92]

Trivial Example

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

$$\begin{array}{rcl} f(k,\sigma) & \stackrel{\rm def}{=} & \delta((\mathsf{F}\, f_\sigma)(\sigma\, k)) \\ & = & id_C((C\, f_\sigma)(\sigma\, k)) \\ & = & id_C(id_C(\sigma\, k)) \\ & = & \sigma\, k \end{array}$$

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

$$\phi(k,\sigma) \stackrel{\rm def}{=} k \in dom\,\sigma \tag{264}$$

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.

An Illustration of Calculated Reification

Recall

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

Data-level Reification

First transformation is recursion removal according to (259)

$$DecTree \ \ \trianglelefteq_1 \ \ DecTree_1$$

leading to

$$DecTree_1 \ = \ K \times (K \rightharpoonup What \times (Answer \rightharpoonup K)) \tag{265}$$

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

1. For K a domain of *pointers*,

   $$K \rightharpoonup What \times (Answer \rightharpoonup K) \tag{266}$$

   — that is:

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

   — 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”, (266) models the object database

   $$name \rightarrow object$$

implicit in an object-oriented programming environment, *cf.* the following VDM "sugaring" of (265):

$$\begin{array}{rcrl} DecTree_1 & :: & ObjName: & K \\ & & Archive: & ObjBase \\ ObjBase & = & K \stackrel{m}{\to} Attributes & \\ Attributes & :: & Q: & What \\ & & SubObjs: & Answer \stackrel{m}{\to} K \end{array}$$

Proceeding via law (171):

$$\begin{array}{rcl} DecTree_1 & = & K \times (K \rightharpoonup What \times (Answer \rightharpoonup K)) \\ & \trianglelefteq_2 & K \times ((K \rightharpoonup What) \times ((K \times Answer) \rightharpoonup K)) \\ & = & DecTree_2 \end{array} \tag{267}$$

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

- $(K \times Answer) \rightharpoonup K$ = ***state transition diagram*** of a (deterministic) ***finite state automaton*** ($Answer$ = input stimuli);
- first factor $k \in K$ in (267) = current state of the automaton;
- $K \rightharpoonup What$ = semantic table assigning a meaning to each state.

Carrying on our reasoning,

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

we obtain a final model,

$$DecTree_4 = K \times (2^{K \times What} \times 2^{K \times Answer \times K}) \tag{268}$$

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 37 and 38 for details.)

The synthesis of the overall abstraction invariant is performed by rule (163), *i.e.*,

$$f = \bigcirc_{i=1}^{4} f_i$$

will lead to abstraction function

$$f(k,(t,t')) \stackrel{\mathrm{def}}{=} \begin{array}{ll} let & w = the(\{\pi_2\, r \mid\ r \in t \wedge \pi_1\, r = k\}) \\ in & (w, \left(\begin{array}{c} a \\ f(k',(t,t')) \end{array}\right)_{(k'',a,k') \in t' \wedge k''=k}) \end{array} \tag{269}$$

and the overall concrete invariant,

$$\phi \stackrel{\mathrm{def}}{=} \phi_3 \circ f_4 \wedge \phi_1 \circ \bigcirc_{i=2}^{4} f_i$$

will lead to

$$\phi(k,(t,t')) = \tag{270}$$

$$\begin{array}{l} (\forall (k,w),(k',w') \in t : k = k' \Rightarrow w = w') \wedge \\ (\forall (k_1,a,k_1'),(k_2,b,k_2') \in t' : (k_1 = k_2 \wedge a = b) \Rightarrow k_1' = k_2') \wedge \\ aux(k,t,t',\emptyset) \end{array} \tag{271}$$

where

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

Another Example of Recursion Removal

Recall (16),

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

in which we find the “pedigree functor”:

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

REMARK 34: **On SETS's first recursion removal theorem**

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

Theorem 5 (First Recursion Removal Theorem).

Let F *be a* $\mathbf{Set}$ *polynomial endofunctor and let* $(X_0,\ X_0 \xleftarrow{\delta} \mathsf{F}X_0\ )$ *be a fixpoint of* 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* $K \times (K \rightharpoonup \mathsf{F}\,K) \xrightarrow{\phi} 2$ *be a predicate defined by* [b]

$$\begin{array}{rcl}\phi(k_0,\sigma) \stackrel{\rm def}{=} k_0 \in dom\,\sigma & \wedge & (k' <_{\mathsf{F}} k \Rightarrow k' \in dom\,\sigma) \\ & \wedge & (\not\exists k \in K : k <^{+}_{\mathsf{F}} k) \\ & \wedge & dom\,\sigma \text{ is finite}\end{array}$$

Finally, let $(K \times (K \rightharpoonup \mathsf{F}\,K))_{\phi} \xrightarrow{f} X_0$ *be a function defined by*

$$f(k_0,\sigma) \stackrel{\rm def}{=} f_\sigma\,k_0 \qquad (272)$$

where $dom\,\sigma \xrightarrow{f_\sigma} X_0$ *is an auxiliary recursive function defined as follows:*

$$f_\sigma\,k \stackrel{\rm def}{=} \delta((\mathsf{F}\,f_\sigma)(\sigma\,k)) \qquad (273)$$

Then

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

holds.

Proof. See [Jou92]. □

Let us see an example of abstraction function f (272) synthesis concerning polynomial functor (255). Algebra $(A^\star,\delta)$ where $\delta = [\,\underline{[\,]}, cons\,]$ — recall ι on p.46 — is a well-known fixpoint solution of this functor [MA86]. For $\mathsf{F}\,f_\sigma$ we obtain $(1 + A \times f_\sigma)$ and so

$$\begin{array}{rcl}\delta \circ (1 + A \times f_\sigma) & = & ([\,\underline{[\,]}, cons\,]) \circ (1 + A \times f_\sigma) \\ & = & \{\ +\text{-absorption (47)}\ \} \\ & & [\,\underline{[\,]}, cons \circ (A \times f_\sigma)\,]\end{array}$$

So,

$$\begin{array}{rcl}f(k,\sigma) & \stackrel{\rm def}{=} & (\delta \circ (1 + A \times f_\sigma))(\sigma\,k) \\ & = & [\,\underline{[\,]}, cons \circ (A \times f_\sigma)\,](\sigma\,k) \\ & = & \left\{\begin{array}{rcl}\sigma\,k = i_1\,NIL & \Rightarrow & [\,] \\ \sigma\,k = i_2\,(a,k') & \Rightarrow & cons(a, f_\sigma\,k')\end{array}\right. \\ & = & \left\{\begin{array}{rcl}\sigma\,k = i_1\,NIL & \Rightarrow & [\,] \\ \sigma\,k = i_2\,(a,k') & \Rightarrow & cons(a, f(k',\sigma))\end{array}\right.\end{array}$$

which may be written in a more "VDM-like" notation as follows:

$$\begin{array}{rcl}f(k,\sigma) & \stackrel{\rm def}{=} & if\ \textit{is-NIL}(\sigma\,k) \\ & & then\ [\,] \\ & & else\ \ let\ \ \ \sigma\,k = (a,k') \\ & & \qquad\ \ in\ \ \ \ cons(a, f(k',\sigma))\end{array} \qquad (274)$$

[a] Note that a least fixpoint solution to F (258) always exists wherever 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 (261) and (262) for the definition of $<_{\mathsf{F}}$.

REMARK 35: **More about the recursion-removal concrete invariant**

Equation (263) 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 (263) by introducing an auxiliary function

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

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

$$\begin{array}{ll} reach(k,\sigma) \stackrel{\rm def}{=} & \\ & \{k\} \cup \{k' \in K \mid\ k' <^{+}_{\mathsf{F}} k\} \\ = & \{k\} \cup \left\{ \begin{array}{lll} \emptyset & \Leftarrow & k \not\in dom\,\sigma \\ \bigcup_{k' <_{\mathsf{F}} k} \{k'\} \cup \{k'' \in K \mid\ k'' <^{+}_{\mathsf{F}} k'\} & \Leftarrow & k \in dom\,\sigma \end{array} \right. \\ = & \{k\} \cup \left\{ \begin{array}{lll} \emptyset & \Leftarrow & k \not\in dom\,\sigma \\ \bigcup_{k' <_{\mathsf{F}} k} reach(k',\sigma) & \Leftarrow & k \in dom\,\sigma \end{array} \right. \end{array}$$

The test for definedness may be encapsulated in a predicate

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

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

$$defined(k,\sigma) \stackrel{\rm def}{=} \left\{ \begin{array}{lll} \text{FALSE} & \Leftarrow & k \not\in dom\,\sigma \\ \forall k' <_{\mathsf{F}} k : defined(k',\sigma) & \Leftarrow & k \in dom\,\sigma \end{array} \right.$$

Note that both $reach$ and $defined$ do not compute the intended set-theoretical fixpoint implicit in the transitive closure of $<_{\mathsf{F}}$ wherever this relation is cyclic [a].
A $<_{\mathsf{F}}$-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

$$\phi(k,\sigma) = \phi_{aux}(k,\sigma,\emptyset) \tag{275}$$

where

$$\begin{array}{l} \phi_{aux}(k,\sigma,C) \stackrel{\rm def}{=} \\ \quad \left\{ \begin{array}{lll} \text{FALSE} & \Leftarrow & k \not\in dom\,\sigma \vee k \in C \\ \forall k' <_{\mathsf{F}} k : \phi_{aux}(k',\sigma,C \cup \{k\}) & \Leftarrow & k \in dom\,\sigma \end{array} \right. \end{array} \tag{276}$$

Exercise 15 Show that, for $\mathsf{F}\,X = C \rightharpoonup \mathsf{G}\,X$, $\in_{\mathsf{F}}$ expands to:

$$k \in_{C \rightharpoonup \mathsf{G}} x \stackrel{\rm def}{=} \exists c \in dom\,x : k \in_{\mathsf{G}} x\,c \tag{277}$$

□

[a] Note in passing that, similarly to (162), logical conjunction in (261) should once again regarded as non-strict.

REMARK 36: **Particular cases of recursive data model reification**

Further to the constant functor $\mathsf{F}\,X = C$ which lead us to (264), other particular cases are of interest concerning (259):

– Constant functor for $C = 0$. This case trivializes (258) to the definition $X \cong 0$ of an empty data domain; note that $K \times (K \rightharpoonup \mathsf{F}\,K)$ becomes $K \times (K \rightharpoonup 0)$ which reduces to

$$\begin{array}{rcl} K \times (K \rightharpoonup 0) & \cong & K \times \bigcup_{X \subseteq K} 0^X \\ & \cong & K \times 0^0 \end{array}$$

That is, $\sigma : 0 \to 0$ is bound to be completely undefined, and $dom(\sigma) = \emptyset$; then $k \in dom\,\sigma =$ FALSE and (264) reduces to

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

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

– Identity functor $\mathsf{F}\,X = X$. This case trivializes (258) 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 (263) 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 *wrt.* $k_1 <_{\lambda X.X} k$ (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 (263):

$$\exists m \in P : \sigma\,m \notin P \tag{278}$$

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 (278). Thus the conjunction of definedness and well-foundedness in (264) is impossible, and we obtain

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

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

Exercise 16 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

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

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

REMARK 37: **Calculation details for abstraction function (269)**

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

$$f_4 = id_K \times (id_{(2^{K \times What})} \times 2^{\langle\langle \pi_1, \pi_2 \rangle, \pi_3\rangle})$$

Abstraction map f_3 is based on mkf (158), which abstracts mappings from relations (wherever possible),

$$f_3 \quad = \quad id_K \times (mkf \times mkf)$$

while f_2 simply is

$$f_2 = id_K \times \Join_n$$

recall (194). Finally, the first step is abstracted by recursive function

$$\begin{array}{rcll}
f(k,\sigma) & \stackrel{\mathrm{def}}{=} & (\mathsf{F}\, f_\sigma)(\sigma\, k) & \\
& = & (What \times (Answer \rightharpoonup f_\sigma))(\sigma\, k) & \\
& = & let & \sigma\, k = (w, \sigma') \\
& & in & (What\, w, (Answer \rightharpoonup f_\sigma)\sigma') \\
& = & let & \sigma\, k = (w, \sigma') \\
& & in & (w, \begin{pmatrix} a \\ f_\sigma(\sigma'\, a) \end{pmatrix}_{a \in dom\, \sigma'}) \\
& = & let & \sigma\, k = (w, \sigma') \\
& & in & (w, \begin{pmatrix} a \\ f(\sigma'\, a, \sigma) \end{pmatrix}_{a \in dom\, \sigma'})
\end{array} \tag{279}$$

cf. (260) and (22), and the $C \rightharpoonup f$ entry in the table of page p.64. Then

$$f_2 \circ f_3 \circ f_4 \quad = \quad id_K \times (\Join_n \circ (mkf \times mkf \circ 2^{\langle\langle \pi_1, \pi_2 \rangle, \pi_3\rangle}))$$

which boils down to pointwise level as follows:

$$\lambda(k,(t,t')).(k, \begin{pmatrix} x \\ (w, \begin{pmatrix} a \\ k' \end{pmatrix}_{(k'',a,k') \in t' \wedge k'' = x}) \end{pmatrix}_{(x,w) \in t}) \tag{280}$$

Composing (280) with (279) one obtains, after some simplifications,

$$\begin{array}{rll}
f(k,(t,t')) \stackrel{\mathrm{def}}{=} & let & w = the(\{\pi_2\, r \mid r \in t \wedge \pi_1\, r = k\}) \\
& in & (w, \begin{pmatrix} a \\ f(k',(t,t')) \end{pmatrix}_{(k'',a,k') \in t' \wedge k'' = k})
\end{array}$$

that is, (269).

REMARK 38: **Calculation details for concrete invariant (271)**

Let us now address the calculation of (271), which should stem from

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

where both ϕ_2 and ϕ_4 are $\underline{\text{TRUE}}$, the everywhere TRUE predicate, and ϕ_3 involves fdp (222),

$$\phi_3 = \underline{\text{TRUE}} \times fdp \times fdp$$

Then

$$\begin{array}{rcl} \phi_3(f_4(k,(t,t'))) & = & \text{TRUE} \wedge fdp\, t \wedge fdp(\{((k,a),k') \mid (k,a,k') \in t'\}) \\ & = & fdp\, t \wedge fdp(\{((k,a),k') \mid (k,a,k') \in t'\}) \end{array} \tag{281}$$

By calculating $fdp(\{((k,a),k') \mid (k,a,k') \in t'\})$ further in (281) we will have

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

Finally, ϕ_1 (263) is based upon the ordering

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

cf. (277). Recalling (275) and (276) above,

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

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

$$\begin{array}{l} aux(k,t,t',C) \stackrel{\text{def}}{=} \\ \quad \left\{ \begin{array}{lcl} \text{FALSE} & \Leftarrow & k \not\in 2^{\pi_1}\, t \vee \\ & & k \in C \\ \forall r \in \{\pi_1\, s = k \mid s \in t'\} : aux(\pi_3\, r, t, t', C \cup \{k\}) & \Leftarrow & k \in 2^{\pi_1}\, t \end{array} \right. \end{array} \tag{282}$$

Putting everything together, we finally obtain the following (non-trivial) concrete invariant over (268):

$$\begin{array}{l} \phi(k,(t,t')) = \\ \qquad (\forall (k,w),(k',w') \in t : k = k' \Rightarrow w = w') \wedge \\ \qquad (\forall (k_1,a,k_1'),(k_2,b,k_2') \in t' : (k_1 = k_2 \wedge a = b) \Rightarrow k_1' = k_2') \wedge \\ \qquad aux(k,t,t',\emptyset) \end{array} \tag{283}$$

where aux is the recursive auxiliary predicate given by (282).
The reader who finds (283) a too complex concrete invariant should think of 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.

— see also (147).

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

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

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

Next, Newton's *binomial* formula (36) will be of some help,

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

— a reasoning chain in which law (172) 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 39 for some more results available in SETS for recursion removal.

Hierarchical Data Models as Recursive Functorial Extensions

Every data model specified by some given $\mathbf{Set}$ functorial expression $\mathsf{F}\,A$, for some nonempty A, gives birth to its "canonical" hierarchical extension, which is recursive data model

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

Examples

REMARK 39: **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 (259) 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.15.

Pointers have particularly to do with recursive patterns of the shape $\mathsf{F}\,X = 1 + \mathsf{G}\,X$. Although such patterns fall well into Theorem 5, 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 6 (Second Recursion Removal Theorem).
Let G *be a polynomial functor in* $\mathbf{Set}$ *and let* $(X_0,\ X_0 \xleftarrow{\ \delta\ } \mathsf{F}X_0\ )$ *be a fixpoint solution of equation*

$$X \cong 1 + \mathsf{G}\,X$$

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,*

$$X_0 \trianglelefteq_f^{\phi} ((1+K) \times (K \rightharpoonup \mathsf{G}(1+K)))$$

where f *is given by*

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

and where

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

both partial over concrete invariant

$$\begin{array}{l} \phi((i,x),\sigma) \stackrel{\rm def}{=} \\ \quad ((x \neq NIL \Rightarrow x \in dom\,\sigma) \quad \wedge \quad (k' <_\sigma k \Rightarrow k' \in dom\,\sigma) \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad (\not\exists k \in K : k <_\sigma^+ k) \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad dom\,\sigma \text{ is finite}) \end{array}$$

Proof. See [Jou92]. □

Exercise 17 Recalculate (274) by applying Theorem 6 instead of Theorem 5 to polynomial functor (255). **Hint**: have a look at Remark 46.
□

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

REMARK 40: **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 (256) intuitively implement finite lists (255) — 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 7 ($\trianglelefteq$-Monotonicity of $\mathbf{Set}$-Recursion). *Let* F *and* G *be two* $\mathbf{Set}$*-endofunctors, built by arbitrary composition of the* $\trianglelefteq$*-monotone operators of Corollary 1. If*

$$\mathsf{F}\,X \trianglelefteq \mathsf{G}\,X \tag{284}$$

for any X*, then a solution to domain equation*

$$X \cong \mathsf{G}\,X \tag{285}$$

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

$$X \cong \mathsf{F}\,X \tag{286}$$

Proof. See [Oli90] for a non-constructive proof. The diagram which follows synthesises abstraction function $\mu\mathsf{F} \xleftarrow{f} G_0$ mapping data of the carrier of solution G_0 to (285) to $\mu\mathsf{F}$ as a hylomorphism $[\![\beta,\nu,\iota]\!]_{\mathsf{F},\mathsf{G}}$,

$$\begin{tikzcd}
G_0 \arrow[rr, bend left, "\beta"] \arrow[d, "f"'] & \mathsf{F}\,G_0 \arrow[d, "\mathsf{F}\,f"'] & \mathsf{G}\,G_0 \arrow[l, "\nu_{G_0}"] \arrow[d, "\mathsf{G}\,f"] \\
\mu\mathsf{F} & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota"] & \mathsf{G}\,\mu\mathsf{F} \arrow[l, "\nu_{\mu\mathsf{F}}"]
\end{tikzcd} \qquad f \stackrel{\mathrm{def}}{=} \iota \circ \nu \circ (\mathsf{F}\,f) \circ \beta \tag{287}$$

where ν is the (natural) epic implicit in (284). The fact that f is epic follows (by fixpoint induction) from both δ and ι being iso, ν being epic and $\mathbf{Set}$-functors preserving epics — recall Remark 29. The synthesis of concrete invariant ϕ is similar. □

Let $\mathsf{F}\,X = 1 + A \times X$ and $\mathsf{G}\,X = 1 + A \times X^2$, for instance. Then $\mathsf{F}\,X \trianglelefteq \mathsf{G}\,X$ holds — under abstraction map $1 + A \times \pi_1$, for instance, a functorial consequence of (184) — 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 5 and 6, where constraint "$dom\,\sigma$ is finite" is explicitly quoted in the concrete invariant (note that $K \cong I\!N$). This also happens implicitly in other situations where $I\!N$ is allowed to occur in the exponent of exponentials. In (159), for instance, a non-finite B (allowed in $A \rightharpoonup B$ provided A is finite) jumps to the exponent of $2^{A \times B}$ on the right-hand side. But fdp (222), for a finite A, restores finiteness. In law (175) finiteness is guaranteed by a concrete invariant imposing injectivity (otherwise indexing becomes ambiguous). In law (176) the concrete invariant imposes a finite domain on the right-hand side, otherwise an infinite list is retrieved.

- In general, $X \cong \mathsf{F}\,X$ does not always have solutions in $Sets$ because exponentials are 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 $\mathsf{F}\,X$ 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 18 For A a nonempty set, prove the following trivial refinement law

$$1 \quad \trianglelefteq \quad A \tag{288}$$

and use this result and Theorem 7 to show that $I\!N$, the least solution to $X \cong 1 + X$, is refined by $A^\star$, the least solution to (255). What is the intuition behind this (highly redundant) reification? Given a particular sequence of As, what is the natural number it represents?
□

Take the finite-mapping datatype $\mathsf{F}\,A \stackrel{\rm def}{=} C \rightharpoonup A$, for some C, *e.g.*

$$bill = service \rightharpoonup charge$$

recalling requirement

> *"A bill is a statement of charges for services rendered."*

from p.2 of this course.

Then take its hierarchical extension

$$X \;\cong\; C \rightharpoonup (A + X)$$

and apply it to $bill$.

Congratulations: you have just specified the "hierarchical bill organizer" demanded on p.2:

$$X \;=\; service \rightharpoonup (charge + X)$$

Want to find a solution for it, implementable in *e.g.* MICROSOFT ACCESS?

Just calculate it:

$$\begin{array}{rcl}
\mu\mathsf{F} & \trianglelefteq & K \times (K \rightharpoonup (service \rightharpoonup (charge + K))) \\
 & \cong & K \times (K \rightharpoonup 1 \times (service \rightharpoonup (charge + K))) \\
 & \trianglelefteq & K \times ((K \rightharpoonup 1) \times (K \times service \rightharpoonup (charge + K))) \\
 & \cong & K \times 2^K \times (K \times service \rightharpoonup charge) \times (K \times service \rightharpoonup K)
\end{array}$$

Congratulations once again: you have just calculated the relational data model

$$\begin{array}{|c|c||c|}\hline \#bill & service & charge \\ \hline \cdots & \cdots & \cdots \end{array} + \begin{array}{|c|c||c|}\hline \#bill & service & \#bill \\ \hline \cdots & \cdots & \cdots \end{array} + \begin{array}{|c|}\hline \#bill \\ \hline \cdots \end{array}$$

suggested in the introduction to this course.

By the way: make $C = FileId$ and $A = File$ and check that you have just specified the basic structure of a hierarchical file system

$$X \;\cong\; FileId \rightharpoonup (File + X)$$

Now, let $\mathsf{F}\,A \stackrel{\mathrm{def}}{=} (A \rightharpoonup I\!N)_+$, take its hierarchical extension,

$$X \;\cong\; ((A+X) \rightharpoonup I\!N)_+$$

and calculate its nonrecursive refinement:

$$\mu\mathsf{F} \;\;\trianglelefteq\;\; K \times (K \rightharpoonup (((A+K) \rightharpoonup I\!N)_+))$$

Let $A = \#Comp$ and rename K to $\#Equip$:

$$\mu\mathsf{F} \;\;\trianglelefteq\;\; \#Equip \times \underbrace{(\#Equip \rightharpoonup (((\#Comp + \#Equip) \rightharpoonup I\!N)_+))}_{Equip}$$

Congratulations once again: you have just re-specified the $Equip$ component of the PPD data model of p.72.

Recursion Refining Recursion: Epilogue

We have seen how to refine recursive structures into **non-recursive** ones.

In the opposite direction, refining recursive structures by other (**more**) recursive structures is also of practical interest.

For instance, it is to be expected that binary search-trees (256),

$$X \;\cong\; 1 + A \times X^2$$

be able to implement finite lists (255) ...

$$X \;\cong\; 1 + A \times X$$

As a matter of fact (Theorem 7), wherever

$$\mathsf{F}\,X \trianglelefteq^{\phi}_{\nu} \mathsf{G}\,X$$

holds, then

- any solution $(G_0,\ G_0 \xleftarrow{\delta} \mathsf{G}\,G_0\ )$ to recursive type $X\ \cong\ \mathsf{G}\,X$ is a $\unlhd$-refinement of the least solution $(\mu\mathsf{F},\ \mu\mathsf{F} \xleftarrow{\iota} \mathsf{F}\,\mu\mathsf{F}\ )$ to recursive type $X \cong \mathsf{F}\,X$;
- the abstraction map is hylomorphism $[\![\delta^{-1}, \nu, \iota]\!]_{\mathsf{F},\mathsf{G}}$:

$$\begin{tikzcd}
G_0 \arrow[rr, bend left, "\delta^{-1}"] \arrow[d, "f"'] & \mathsf{F}\,G_0 \arrow[d, "\mathsf{F}\,f"'] & \mathsf{G}\,G_0 \arrow[l, "\nu_{G_0}"] \arrow[d, "\mathsf{G}\,f"] \\
\mu\mathsf{F} & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota"] & \mathsf{G}\,\mu\mathsf{F} \arrow[l, "\nu_{\mu\mathsf{F}}"]
\end{tikzcd}$$

Example

$\mathsf{F}\,X = 1 + A \times X$ (linear lists) — $\mu\mathsf{F} = A^\star$

$\mathsf{G}\,X = 1 + A \times X \times X$ (binary trees) —- we denote by BT_A the selected fixpoint G_0.

Let us calculate how BT_A refines $A^\star$. Everything stems from law (184)

$$A \unlhd_{\pi_1} A \times B$$

In fact,

$$\begin{array}{rcl}
\mathsf{F}\,X & = & 1 + A \times X \\
& \unlhd & \qquad \{ \text{ under abstraction } 1 + \pi_1 \times X \ \} \\
& & 1 + (A \times X) \times X \\
& \cong & \qquad \{ \text{ under abstraction } 1 + \langle\langle \pi_1, \pi_2\rangle, \pi_3\rangle \ \} \\
& & 1 + A \times X \times X \\
& = & \qquad \{ \text{ by definition } \} \\
& & \mathsf{G}\,X
\end{array}$$

The overall abstraction function is, then,

$$
\begin{array}{cl}
\nu = & (1 + \pi_1 \times X) \circ (1 + \langle\langle \pi_1, \pi_2 \rangle, \pi_3\rangle) \\
= & \qquad \{ \text{ bifunctor } + \text{ commutes with composition (11), constant } 1 \ \} \\
& 1 + ((\pi_1 \times X) \circ \langle\langle \pi_1, \pi_2 \rangle, \pi_3\rangle) \\
= & \qquad \{ \ \times\text{-absorption (43) and constant } X \ \} \\
& 1 + \langle \pi_1 \circ \langle \pi_1, \pi_2 \rangle, \pi_3 \rangle \\
= & \qquad \{ \ \times\text{-cancellation (44) } \} \\
& 1 + \langle \pi_1, \pi_3 \rangle
\end{array}
$$

Then we draw the hylomorphism diagram (287):

$$
\begin{tikzcd}
BT_A \arrow[d, "f"'] \arrow[rr, bend left, "\delta^{-1}"] & 1 + A \times BT_A \arrow[d, "1+A\times f"'] & 1 + A \times BT_A \times BT_A \arrow[l, "1+\langle\pi_1,\pi_3\rangle_{BT_A}"] \arrow[d, "1+A\times f\times f"] \\
A^\star & 1 + A \times A^\star \arrow[l, "{[\ nil, cons\ ]}"] & 1 + A \times A^\star \times A^\star \arrow[l, "1+\langle\pi_1,\pi_3\rangle_{A^\star}"]
\end{tikzcd}
$$

Final calculation:

$$
\begin{array}{cl}
f \stackrel{\mathrm{def}}{=} & [\, nil, cons \,] \circ (1 + \langle \pi_1, \pi_3 \rangle) \circ (1 + A \times f) \circ \delta^{-1} \\
= & \qquad \{ \text{ bifunctor } + \text{ commutes with composition (11), constant } 1 \ \} \\
& [\, nil, cons \,] \circ (1 + \langle \pi_1, \pi_3 \rangle \circ (A \times f)) \circ \delta^{-1} \\
= & \qquad \{ \ \times\text{-fusion (42), (32) } \textit{etc.} \} \\
& [\, nil, cons \,] \circ (1 + \langle A \circ \pi_1, f \circ \pi_3 \rangle) \circ \delta^{-1} \\
= & \qquad \{ \ +\text{-absorption (47) and constant } A \ \} \\
& [\, nil, cons \circ \langle \pi_1, f \circ \pi_3 \rangle \,] \circ \delta^{-1} \\
= & \qquad \{ \text{ going pointwise, for the obvious } \delta^{-1} \ \} \\
& \lambda t. \left\{ \begin{array}{rcl} t = i_1\, NIL & \Rightarrow & [\,] \\ t = i_2\,(a, t_1, t_2) & \Rightarrow & cons(a, f\, t_2) \end{array} \right.
\end{array}
$$

MORAL OF THIS CALCULATION:

If we systematically ignore the lefthand side branch of a binary tree, it "degenerates" into (*i.e.* represents) a linear list — as we were expecting!

‘In Loco’ Data-type Invariant Handling

The main purpose of this section is to describe how our method deals with so-called *local* (“in loco”) data type invariants.

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

- Invariant
  - (1) concrete — entailed by data refinement process
  - (2) in loco
    - (2.1) purely semantic
    - (2.2) for algorithmic efficiency

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

$$Date \cong 31 \times 12 \times I\!N$$

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

$$\begin{array}{l}
dateOk : Date \longrightarrow 2 \\
dateOk(d,m,y) \stackrel{\rm def}{=} \\
\quad \left\{ \begin{array}{lcl}
m \in \{1,3,5,7,8,10,12\} & \Rightarrow & d \leq 31 \wedge \\
 & & (\neg(y = 1582 \wedge m = 10) \\
 & & \vee (d < 5) \vee (14 < d)) \\
m \in \{4,6,9,11\} & \Rightarrow & d \leq 30 \\
m = 2 \wedge leapYear\, y & \Rightarrow & d \leq 29 \\
m = 2 \wedge \neg leapYear\, y & \Rightarrow & d \leq 28
\end{array} \right.
\end{array} \tag{289}$$

where

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

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 calculations?

Case (2.1)

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

$$\{a \in A \mid \ \alpha\, a\}$$

REMARK 41: **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. $dateOk$ (289) 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 (174), 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 *wrt.* the abstraction function $elems$ (p.32) 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 19 Concerning the PPD data model specification of p.72, 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 (289) 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.)

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

$$Date_{dateOk}$$

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

Theorem 8. *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:*

$$A_\alpha \trianglelefteq_f^\phi B_{\wedge\circ\langle\phi,\alpha\circ f\rangle} \tag{290}$$

Proof: *Recall Remark 29.*
□

Case (2.2)

Let

$$A \trianglelefteq_f^\phi B \tag{291}$$

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,

$$\begin{array}{rcl} f|\beta & : & B_{\wedge\circ\langle\phi,\beta\rangle} \longrightarrow A \\ b & \leadsto & f\,b \end{array}$$

be surjective, then "stronger" refinement

$$A \trianglelefteq_{f|\beta}^{\wedge\circ\langle\phi,\beta\rangle} B \tag{292}$$

will be valid and can replace (291) (see Theorem 9).

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

LIMIT CASE:

$$A \cong B_{\wedge\circ\langle\phi,\beta\rangle} \tag{293}$$

for instance:

$$2^A \quad \cong_{elems|linear} \quad (A^\star)_{linear} \tag{294}$$

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 **left-hand** side of a $\unlhd$-law (Theorem 8) and another in which it is introduced on the **right-hand** side (Theorem 9).

Finally, the following result (recall Theorem 4) helps in reasoning about subscripted data domains:

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

Example of Local Invariant Impact Calculation

Consider that

- the following requirement on $Accounts$ (p.69) is added to the specification:

  No account's balance can ever be lower than m_0

- this happens after our $Accounts_5$ 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:

$$\alpha \;=\; AccId \rightharpoonup (min \circ \pi_2)$$

for $min\, a \stackrel{\mathrm{def}}{=} a \geq m_0$.

Following (290), the extra work amounts to calculating $\alpha \circ f$, where f is given by (248):

$$\begin{array}{rcl}
\alpha \circ f & = & (AccId \rightharpoonup (min \circ \pi_2)) \circ (248) \\
& = & (AccId \rightharpoonup (min \circ \pi_2)) \circ (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ \\
& & \bowtie \circ \langle \pi_1, \dagger \circ ((AccId \rightharpoonup \underline{\emptyset}) \times collect) \rangle \circ (mkf \times id)
\end{array}$$

REMARK 42: **Main results concerning 'in loco' invariant handling**

Theorem 9. *Let* $A \trianglelefteq^{\phi}_{f} B$ *hold and* $\beta : B \longrightarrow 2$ *be a predicate such that the restriction of* f *by* β*,*

$$f|\phi : B_{\wedge\circ\langle\phi,\beta\rangle} \longrightarrow A$$

is still a surjection. Then

$$A \trianglelefteq^{\wedge\circ\langle\phi,\beta\rangle}_{f|\phi} B$$

holds.

Proof. Obvious, from the definition of the $\trianglelefteq$-order [Oli90]. $\square$

Corollary 2. *For* 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*

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

Proof. Apply Theorem 4. Concerning the constant functor, one has $C(A_\phi) = C$ which is the same as $(C\, A)_{C\,\phi} = C_{\underline{\text{TRUE}}}$.
Concerning products we have:

$$\begin{array}{rcl} A_\phi \times B_\varphi & = & \{(a,b) \mid\ a \in A_\phi \wedge b \in B_\varphi\} \\ & = & \{(a,b) \mid\ a \in A \wedge b \in B \wedge \phi\, a \wedge \varphi\, b\} \\ & = & \{(a,b) \mid\ (a,b) \in A \times B \wedge (\phi \times \varphi)(a,b)\} \\ & = & (A \times B)_{\phi\times\varphi} \end{array}$$

Concerning co-products we have:

$$\begin{array}{rcl} A_\phi + B_\varphi & = & \{i_1\, a \mid\ a \in A \wedge \phi\, a\} \cup \{i_2\, b \mid\ b \in B \wedge \phi\, b\} \\ & = & \{x \mid\ x = i_1\, a \wedge a \in A \wedge \phi\, a \vee x = i_2\, b \wedge b \in B \wedge \phi\, b\} \\ & = & \{x \mid \left\{ \begin{array}{lcl} x = i_1\, a \wedge a \in A & \Rightarrow & \phi\, a \\ x = i_2\, b \wedge b \in B & \Rightarrow & \phi\, b \end{array} \right. \} \\ & = & (A + B)_{\phi+\varphi} \end{array}$$

Concerning finite exponentials 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:

$$\begin{array}{rcl} 2^{A_\phi} & = & \{S \mid\ S \subseteq A_\phi\} \\ & = & \{S \subseteq A \mid\ \forall a \in S : \phi\, a\} \\ & = & (2^A)_{2^\phi} \end{array}$$

$\square$

Note that Theorem 4 extends to recursive functors which have a solution in $\mathbf{Set}$. 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 4, 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:

$$\psi\, l \stackrel{\text{def}}{=} \left\{ \begin{array}{rcl} l = i_1\, NIL & \Rightarrow & \text{TRUE} \\ l = i_2((a, l')) & \Rightarrow & even\, a \wedge \psi\, l' \end{array} \right.$$

We handle the first composition in the first place:

$$
\begin{array}{cl}
 & (AccId \rightharpoonup (min \circ \pi_2)) \circ (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \\
= & \quad \{ \text{ functors commute with composition (7) } \} \\
 & AccId \rightharpoonup (min \circ \pi_2 \circ \langle \pi_2, \pi_1 \rangle) \\
= & \quad \{ \ \times\text{-cancellation (44) } \} \\
 & AccId \rightharpoonup (min \circ \pi_1)
\end{array}
$$

Next we compose this with $\bowtie$:

$$
\begin{array}{cl}
 & (AccId \rightharpoonup (min \circ \pi_1)) \circ \bowtie \\
= & \quad \{ \text{ functors commute with composition (7) } \} \\
 & (AccId \rightharpoonup min) \circ (AccId \rightharpoonup \pi_1) \circ \bowtie \\
= & \quad \{ \text{ by (189) } \} \\
 & (AccId \rightharpoonup min) \circ | \circ (id \times dom)
\end{array}
$$

Finally:

$$
\begin{array}{cl}
 & (AccId \rightharpoonup min) \circ | \circ (id \times dom) \circ \langle \pi_1, \dagger \circ ((AccId \rightharpoonup \underline{\emptyset}) \times collect) \rangle \circ \\
 & (mkf \times id) \\
= & \quad \{ \text{ by } \times\text{-absorption (43) and identity } \} \\
 & (AccId \rightharpoonup min) \circ | \circ \langle \pi_1, dom \circ \dagger \circ ((AccId \rightharpoonup \underline{\emptyset}) \times collect) \rangle \circ \\
 & (mkf \times id) \\
= & \quad \{ \text{ by (198), } \times \text{ is bifunctor and (125) } \} \\
 & (AccId \rightharpoonup min) \circ | \circ \langle \pi_1, \cup \circ (dom \times dom \circ collect) \rangle \circ (mkf \times id) \\
= & \quad \{ \text{ by (95) } \} \\
 & (AccId \rightharpoonup min) \circ \pi_1 \circ (mkf \times id) \\
= & \quad \{ \text{ by (32) } \} \\
 & (AccId \rightharpoonup min) \circ mkf \circ \pi_1 \\
= & \quad \{ \text{ see Exercise 22 } \} \\
 & 2^{min \circ \pi_2} \circ \pi_1 \\
= & \lambda(\rho, \gamma). \forall t \in \rho : \pi_2\, t \geq m_0
\end{array}
$$

So the “least strengthened” concrete invariant over $Accounts_5$ which accommodates such a requirement is (*cf.* p.70):

$$(fdp\, \rho) \wedge (\forall t \in \rho : \pi_2\, t \geq m_0)$$

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

$$Database \;\cong\; 2^A \tag{295}$$

as a convenient (however primitive!) model for the database file itself:

$$\begin{array}{rcl}
\hline
initialDb & : & \longrightarrow Database \\
initialDb & \stackrel{\rm def}{=} & \emptyset \\
\hline
insert & : & A \times Database \longrightarrow Database \\
insert(a,\sigma) & \stackrel{\rm def}{=} & \sigma \cup \{a\} \\
\hline
find & : & A \times Database \longrightarrow 2 \\
find(a,\sigma) & \stackrel{\rm def}{=} & (a \in \sigma) \\
\hline
remove & : & A \times Database \longrightarrow Database \\
remove(a,\sigma) & \stackrel{\rm def}{=} & \sigma - \{a\} \\
\hline
\end{array} \tag{296}$$

We want to calculate the hash-table implementation of (295). We concentrate on $Database$'s reification, the refinement of the its operators (296) being provided later.

Let $n \in I\!N$ be the number of entries of our intended hash table, and

$$H : A \longrightarrow n$$

Exercise 20 Specify the $discollect$ and zip surjective (epic) operators,

$$\begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad \begin{tikzcd} C \rightharpoonup 2^A \arrow[r, "discollect_A"] \arrow[d, "C \rightharpoonup 2^f"'] & 2^{C \times A} \arrow[d, "2^{C \times f}"] \\ C \rightharpoonup 2^B \arrow[r, "discollect_B"'] & 2^{C \times B} \end{tikzcd} \qquad (2^{C \times f}) \circ discollect = discollect \circ (C \rightharpoonup 2^f) \tag{297}$$

$$\begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad \begin{tikzcd} C \arrow[d, "g"'] \\ D \end{tikzcd} \qquad \begin{tikzcd} A^\star \times C^\star \arrow[r, "zip_{A,C}"] \arrow[d, "f^\star \times g^\star"'] & (A \times C)^\star \arrow[d, "(f \times g)^\star"] \\ B^\star \times D^\star \arrow[r, "zip_{B,D}"'] & (B \times D)^\star \end{tikzcd} \qquad (f \times g)^\star \circ zip = zip \circ (f^\star \times g^\star) \tag{298}$$

and show that neither of these operators is monic.
□

Exercise 21 Check the following equalities:

$$\begin{aligned} \dagger \circ \langle f, |_S \circ f \rangle &= f && (299) \\ rng \circ (i \rightharpoonup B) &= rng && (300) \\ (A \rightharpoonup f)(\sigma \cup \tau) &= (A \rightharpoonup f)(\sigma) \cup (A \rightharpoonup f)\tau && (301) \\ (A \rightharpoonup (f \circ \pi_1)) \circ \bowtie_n &= (A \rightharpoonup f) \circ \pi_1 && (302) \\ rng \circ (A \rightharpoonup f) \circ mkf &= 2^{f \circ \pi_2} && (303) \end{aligned}$$

□

Exercise 22 Prove that the following property holds with respect to mkf (158,229) and an arbitrary (but suitably well-typed) predicate α:

$$(A \rightharpoonup \alpha) \circ mkf = 2^{\alpha \circ \pi_2} \tag{304}$$

□

Exercise 23 Recall invariant inv-AMS in Remark 3 and

- write it as a $\mathbf{Set}$-predicate defined on $Accounts$ (247);
- "push it" downto the calculation of $Accounts_5$.

□

Exercise 24 Let

$$\begin{aligned} mkr &: (A \rightharpoonup B) \longrightarrow 2^{A \times B} \\ mkr\,\sigma &\stackrel{\text{def}}{=} \{(a, \sigma\, a) \mid a \in dom\,\sigma\} \end{aligned}$$

be a function which converts finite mappings into relational form. Prove or contradict the following facts about mkr:

$$\begin{aligned} mkf \circ mkr &= id && (305) \\ rng \circ (A \rightharpoonup f) &= 2^{f \circ \pi_2} \circ mkr && (306) \end{aligned}$$

□

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 $\unlhd$-reasoning in order to convert

$$2^A \unlhd \ldots$$

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 44.

We start from (185), a law enabling a data domain to be refined by adding extra information to it. In this case:

$$A \unlhd_{\pi_2} n \times A \tag{307}$$

Then we impose the following local invariant on $n \times A$:

$$\phi(i, a) \stackrel{\mathrm{def}}{=} i = H\, a \tag{308}$$

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

$$A \unlhd_{\pi_2|\phi} (n \times A)_\phi \tag{309}$$

by Theorem 9. By applying the powerset functor $\lambda X.2^X$ to (309) we obtain

$$2^A \unlhd_{f_1} (2^{n \times A})_{\Phi_1} \tag{310}$$

for

$$\begin{array}{rcl} \Phi_1 & = & \lambda s.(\forall (i, a) \in s : i = H\, a) \\ f_1 & = & \lambda s.\{a \in A \mid\ (i, a) \in s\} \end{array}$$

We now resort to the following instance of the "curry/uncurry" law of exponentials (139), for $A = 2$:

$$2^{n \times A} \cong (2^A)^n \tag{311}$$

The result of "pushing" Φ_1 along step (311) is concrete invariant Φ_2 in

$$(2^{n \times A})_{\Phi_1} \cong_{f_2} ((2^A)^n)_{\Phi_2} \tag{312}$$

which can be calculated via Theorem 8:

$$\Phi_2 \;=\; \lambda t . \forall i \in n : (\forall a \in t\, i : i = H\, a) \tag{313}$$

Also,

$$f_2 \;=\; \lambda t . \{(i,a) \mid\; i \in n \wedge a \in t\, i\}$$

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:

$$\begin{aligned}
f_1(f_2\, t) \;&=\; \{a \in A \mid\; (i,a) \in \{(i',a') \mid\; i' \in n \wedge a' \in t\, i'\}\} \\
&=\; \{a \in A \mid\; i \in n \wedge a \in t\, i\} \\
&=\; \{a \in A \mid\; \exists i \in n : a \in t\, i\} \\
&=\; \{a \in A \mid\; \bigvee_{i \in n} a \in t\, i\} \\
&=\; \bigcup_{i \in n} \{a \in A \mid\; a \in t\, i\} \\
&=\; \bigcup_{i \in n} t\, i
\end{aligned} \tag{314}$$

The intended "hashing effect" has thus already been achieved.

Collision Handling

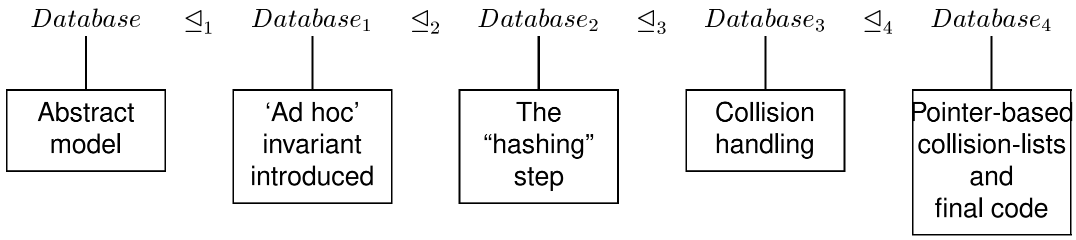
The task of reifying 2^A "recurs" in the co-domain of $n \to 2^A$ — a kind of "fractal" effect on this type of data structuring.

Comments:

- This problem is identified in the literature under the headings *overflow handling* or *collision handling* [HS78].
- See remarks 45 to 47 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:



(315)

Operation Refinement

— Merging 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*

REMARK 44: **Details of proposed hash table calculation**

Step (311) is obtained via the "curry/uncurry" law of exponentials (139), established by the $uncurry$ bijection (233). For $A = 2$ we get the following version of $uncurry$:

$$uncurry\,t \stackrel{\mathrm{def}}{=} \{(b,c) \mid b \in B \wedge c \in t\,b\} \tag{316}$$

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:

$$\begin{array}{rcl} f_2 & = & \lambda t.uncurry\,t \\ & = & \lambda t.\{(i,a) \mid i \in n \wedge a \in t\,i\} \end{array}$$

and

$$\begin{array}{rcl} \Phi_2 & = & \Phi_1 \circ uncurry \\ & = & \lambda t.\Phi_1(\{(i,a) \mid i \in n \wedge a \in t\,i\}) \\ & = & \lambda t.\forall i \in n : (\forall a \in t\,i : i = H\,a) \end{array} \tag{317}$$

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 (317) one has

$$i = (H\,a) \wedge j = H\,a$$

that is, $i = j$ thus contradicting the starting assumption $i \neq j$. □

Exercise 25 Derive the "set-theoretical" version of $uncurry$ which is given by (316). **Hint**: resort to the following isomorphisms explicitly:

$$2^A \underset{\lambda s.(\lambda a.a \in s)}{\overset{\lambda p.\{a \in A \mid p\,a\}}{\rightleftarrows}} \mathcal{P}\,A \tag{318}$$

□

Exercise 26 Recall Exercise 19, in which four local invariants were added to the PPD data model specification of p.72. Resort to Theorem 8 to calculate the weakest impact of these invariants, one at a time, into $RPPD$ (the final relational data model obtained on p.76).

□

REMARK 45: **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

$$n \to (m \to 2^A) \tag{319}$$

under a more elaborate invariant involving n possibly different sub-hashing functions H_i ($i = 1, \ldots, n$):

$$\Phi_3 \quad = \quad \lambda t. \forall i \in n : (\forall j \in m : (\forall a \in (t\, i) j : i = H\, a \wedge j = H_i\, a)) \tag{320}$$

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 (139) — where the hash function is

$$H'\, a = (H\, a, H_{H\, a}\, a)$$

Typically $m < n$, and collision sets get smaller, but 2^A still recurs in (319).
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 (174), turning hash tables into arrays of sequences of collisions:

$$(n \to A^\star)_{\Phi_3}$$

The corresponding invariant Φ_3 is obtained via Theorem 3, 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.64. By Theorem 8 one has,

$$(2^A)^n \trianglelefteq_{elems^n} (A^\star)^n \quad \Rightarrow \quad ((2^A)^n)_{\Phi_2} \trianglelefteq_{elems^n} ((A^\star)^n)_{\underbrace{\Phi_2 \circ elems^n}_{\Phi_3}}$$

that is,

$$\begin{array}{rcl} \Phi_3 & = & \Phi_2 \circ elems^n \\ & = & \lambda t. \forall i \in n : (\forall a \in elems(t\, i) : i = H\, a) \end{array}$$

$A^\star$ is not readily available in programming languages like C or PASCAL. Its reification towards such programming environments has been studied earlier on — recall remarks 34 and 39 — 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

$$X \quad \cong \quad 1 + A \times X \text{ /*\textit{finite lists on} } A \text{ */} \tag{321}$$

Then recursion is *removed* by reifying the recursive pattern of (321) 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":

$$A^\star \quad \trianglelefteq \quad \underbrace{(K \rightharpoonup A \times (K + 1))}_{\text{heap}} \times \underbrace{(K + 1)}_{\text{pointer}} \tag{322}$$

Calculation proceeds in the remark which follows.

REMARK 46: **Hash-table collision handling: a detailed calculation (2)**

The abstraction function at (322) is:

$$f_{(322)}(\sigma, x) \stackrel{\rm def}{=} \begin{cases} x = i_2\, NIL & \Rightarrow \quad [\,] \\ x = i_1\, k & \Rightarrow \quad \begin{array}{ll} let & a = \pi_1(\sigma\, k) \\ & x' = \pi_2(\sigma\, k) \\ in & [a] +\!\!+\, f_{(322)}(\sigma, x') \end{array} \end{cases}$$

which is coupled to the following low-level invariant, ensuring acyclic pointer definedness,

$$\phi_{(322)}(\sigma, x) \stackrel{\rm def}{=} \gamma(x, \sigma, \emptyset)$$

where:

$$\gamma(x, \sigma, C) \stackrel{\rm def}{=} \begin{cases} x = i_2\, NIL & \Rightarrow \quad \text{TRUE} \\ x = i_1\, k \wedge k \notin dom\, \sigma \vee k \in C & \Rightarrow \quad \text{FALSE} \\ x = i_1\, k \wedge k \in dom\, \sigma & \Rightarrow \quad \gamma(\pi_2(\sigma\, k), \sigma, C \cup \{k\}) \end{cases} \tag{323}$$

Concerning $(A^\star)^n$ we will have:

$$(A^\star)^n \quad \trianglelefteq \quad (K \rightharpoonup A \times (K+1)) \times (K+1)^n \tag{324}$$

(leading to an array of pointers to linked-lists sharing the same heap structure) where

$$f_{(324)}(\sigma, t) \stackrel{\rm def}{=} \left(\begin{array}{c} i \\ f_{(322)}(\sigma, t\, i) \end{array} \right)_{i \in n}$$

and

$$\phi_{(324)}(\sigma, t) \stackrel{\rm def}{=} \forall i \in n : \phi_{(322)}(\sigma, t\, i)$$

The overall invariant at this point,

$$\Phi_4 \quad = \quad \phi_{(324)} \wedge \Phi_3 \circ f_{(324)}$$

can be systematically calculated. First, we have:

$$\begin{array}{ll} \Phi_4(\sigma, t) = & \\ & \forall i \in n : \phi_{(322)}(\sigma, t\, i) \wedge \Phi_3 \circ f_{(324)} \\ = & (\forall i \in n : \gamma(t\, i, \emptyset)) \wedge \forall i \in n : (\forall a \in elems\, f_{(322)}(\sigma, t\, i) : i = H\, a) \\ = & \forall i \in n : \gamma(t\, i, \sigma, \emptyset) \wedge (\forall a \in elems\, f_{(322)}(\sigma, t\, i) : i = H\, a) \\ = & \forall i \in n : \gamma(t\, i, \sigma, \emptyset) \wedge \varphi(t\, i, \sigma, i) \end{array}$$

where φ is easily shown to be the following predicate:

$$\varphi(k, \sigma, i) \stackrel{\rm def}{=} \begin{cases} k = NIL & \Rightarrow \quad \text{TRUE} \\ \neg(k = NIL) & \Rightarrow \quad \begin{array}{ll} let & a' = \pi_1(\sigma\, k) \\ & k' = \pi_2(\sigma\, k) \\ in & i = H\, a \wedge \varphi(k', \sigma, i) \end{array} \end{cases}$$

The remaining reasoning is presented in the remark which follows.

REMARK 47: **Hash-table collision handling: a detailed calculation (3)**

Further "fold-unfold" reasoning will lead to

$$\Phi_4(\sigma, t) \quad = \quad \forall i \in n : \Phi_{4aux}(t\, i, \sigma, i, \{\}) \tag{325}$$

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:

$$\Phi_{4aux}(k, \sigma, i, C) \stackrel{\rm def}{=} \left\{ \begin{array}{lll} k = NIL & \Rightarrow & \text{TRUE} \\ \begin{array}{c} k \neq NIL \\ \wedge \\ k \notin dom\, \sigma \\ \vee \\ k \in C \end{array} & \Rightarrow & \text{FALSE} \\ k \in dom\, \sigma & \Rightarrow & \begin{array}{ll} let & a' = \pi_1(\sigma\, k) \\ & k' = \pi_2(\sigma\, k) \\ in & i = H\, a \wedge \Phi_{4aux}(k', \sigma, i, C \cup \{k\}) \end{array} \end{array} \right.$$

Putting everything together, we have calculated the following data structure:

$$((K \rightharpoonup A \times (K+1)) \times (K+1)^n)_{\Phi_4} \tag{326}$$

The final step would be to encode (326) (with some extra "syntactic sugar") in a commercial programming language, *e.g.* C

```
typedef  struct Collision {
         A Synonym;
         struct Collision *Next;
         };                                        (327)
typedef  struct Collision *Bucket;
typedef  Bucket Database4[n]; /* hash table */
```

or PASCAL

```
type  Collision  =  record
                     Synonym:  A;
                     Next:  ^ Collision
                    end;                           (328)
      Bucket     =  ^ Collision;
      Database4  =  (* hash table *)
                    array[1..n] of Bucket;
```

in the documentation of which Φ_4 (325) should be recorded, at conceptual level.

Exercise 27 Suppose that law (175) was used for the purpose of handling hash-table collision sets, that is, one has:

$$(\mathbb{N} \rightharpoonup A)^n$$

instead of $(A^\star)^n$. Calculate abstraction invariant components f_3, Φ_3 for this situation.
□

Exercise 28 A constant function is an obvious "bad" choice for hash function, *e.g.* $H\, a \stackrel{\rm def}{=} i_0$, for $i_0 \in n$. Show that, in this case, Φ_2 (317) reduces to

$$\lambda t. \forall i \in n : i = i_0 \vee t\, i = \emptyset$$

□

$$a = \mathsf{af}\ c \wedge \mathsf{dti}\ c\ , \qquad (17.1)$$

> *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 specialized 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 *choosing* coupling invariants.
- SETS' main purpose is that of *calculating* (instead of choosing) 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.

In the rest of this course we will describe another alternative; that of merging SETS with Bird-Moor's ALGEBRA OF PROGRAMMING [BdM97].

ALGEBRA OF PROGRAMMING is the most recent contribution to a 20 year-old field of research called **program transformation**. Recall *e.g.* the FOLD/UNFOLD CALCULUS [BD77, Dar82] and Backus' FP algebra of programs [Bac78].

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 $\mathbf{Set}$-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:

$$\begin{tikzcd} A \arrow[r, "\sigma"] & B \\ A_1 \arrow[u, "f"] & B_1 \arrow[u, "g"'] \end{tikzcd}$$

The idea is to "close" the above diagram,

$$\begin{tikzcd} A \arrow[r, "\sigma"] & B \\ A_1 \arrow[u, "f"] \arrow[r, "\sigma_1"'] & B_1 \arrow[u, "g"'] \end{tikzcd}$$

so that σ_1 "simulates" the behaviour of σ at concrete-level,

$$\forall a_1 \in A_1 : g(\sigma_1\, a_1) = \sigma(f\, a_1) \tag{329}$$

that is, the diagram commutes.

We will regard σ_1 as the "unknown" of equation (329).

Note that there may be many solutions for σ_1 in equation (329) — the typical "one-abstract to many-concrete" relationship of (functional) refinement.

"Simulation" amounts to exhibiting the same observational behaviour:

$$\sigma\, a \stackrel{\rm def}{=} \begin{array}{ll} let & a_1 \in \{a' \in A_1 \mid f\, a' = a\} \\ in & g(\sigma_1\, a_1) \end{array}$$

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 ...

$$\begin{array}{rcl} \sigma_1\, a_1 & = & g^{-1}(\sigma(f\, a_1)) \\ & = & \ldots \\ & \vdots & \\ & = & \ldots \sigma_1 \ldots a_1 \ldots \text{ /*}\textit{expression free of } f \textit{ and } g \text{ */} \end{array}$$

For non-injective g there may exist more than one solution and idea is to transform refinement equation (329) so that f eventually gives place to g and g is eventually "pulled" up to the outermost place on the right-hand 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:

$$\sigma_1\, a_1 \stackrel{\rm def}{=} \ldots E(\ldots \sigma_1 \ldots a_1 \ldots)$$

(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:

Code 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 (329) 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) $\mathbf{Set}$-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

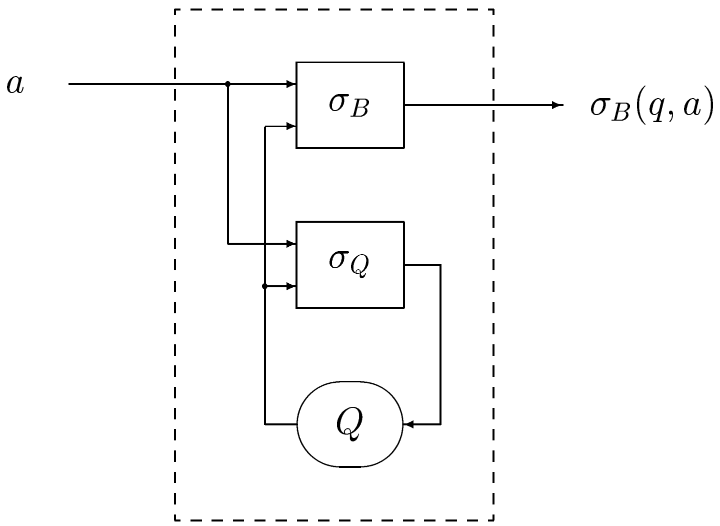
$$\text{post-}\sigma \;:\; Q \times A \times Q \times B \longrightarrow 2$$

(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 <u>result</u> function σ_B and a <u>state</u> 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

$$\begin{tikzcd}
A \times Q \arrow[r, "\sigma_B"] & B \\
A_1 \times Q_1 \arrow[u, "f \times f_Q"] \arrow[r, "x"'] & B_1 \arrow[u, "g"']
\end{tikzcd}$$

and

$$
\begin{tikzcd}
A \times Q \arrow[r, "\sigma_Q"] & Q \\
A_1 \times Q_1 \arrow[u, "f \times f_Q"] \arrow[r, "x"'] & Q_1 \arrow[u, "f_Q"']
\end{tikzcd}
$$

but please note:

- In many situations A and B stay the same ($f = id_A$ and $g = id_B$) 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 (329) to this situation, for f_Q the state abstraction function.

Leaving invariants and pre-conditions aside, for the moment, we will have:

$$
\begin{array}{l}
\forall q_1, q_1' \in Q_1, a_1 \in A_1, b_1' \in B_1 : \\
\qquad \text{post-}\sigma_1(q_1, a_1, q_1', b_1') = \\
\qquad \text{post-}\sigma(f_Q\, q_1', f\, a_1, f_Q\, q', g\, b_1')
\end{array}
\tag{330}
$$

But logical equivalence ($=$) in (330) is too strong: it should be relaxed to implication ($\Rightarrow$),

$$
\begin{array}{l}
\forall q_1, q_1' \in Q_1, a_1 \in A_1, b_1' \in B_1 : \\
\qquad \text{post-}\sigma_1(q_1, a_1, q_1', b_1') \Rightarrow \\
\qquad \text{post-}\sigma(f_Q\, q_1', f\, a_1, f_Q\, q', g\, b_1')
\end{array}
\tag{331}
$$

since increasing determinism is a valid refinement decision (*cf.* 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 10).

Examples of Operation Refinement

Example 1 — Calculating a Typical Database Update Operation

Consider the following functionality over $Accounts$:

$$AccId \times Amount \times Accounts \xrightarrow{credit} Accounts$$

where $credit$ is specified by

$$credit(k, a, q) \stackrel{\text{def}}{=} \begin{cases} k \in dom\, q & \Rightarrow \quad q \dagger \left(\begin{array}{c} k \\ (\pi_1(q\, k), a + \pi_2(q\, k)) \end{array} \right) \\ k \notin dom\, q & \Rightarrow \quad q \end{cases}$$

Informally, $credit$ leaves the database unchanged in case k identifies a nonexistent account. Otherwise, amount a is credited in account k.

Refinement diagram:

$$\begin{tikzcd} AccId \times Amount \times Accounts \arrow[r, "credit"] & Accounts \\ AccId \times Amount \times Accounts_5 \arrow[r, "credit_5"'] \arrow[u, "id \times id \times (248)"] & Accounts_5 \arrow[u, "(248)"'] \end{tikzcd}$$

where (248) indicates abstraction function f of equations (248).

This illustrates a typical "interface-preserving" reification in which only the "type of interest" $Account$ (*cf.* state) is refined.

We want to solve refinement equation

$$credit(k, a, (248)x) \quad = \quad (248)(credit_5(k, a, x)) \tag{332}$$

for $credit_5$.

Going pointfree, we re-write $credit$ in functorial and curried form,

$$\begin{array}{rcl} credit_{k,a} & : & Accounts \longrightarrow Accounts \\ credit_{k,a} & \stackrel{\text{def}}{=} & \dagger \circ \langle id, (AccId \rightharpoonup id \times add_a) \circ |_{\{k\}} \rangle \end{array} \tag{333}$$

where add_a abbreviates $+ \circ \langle id, \underline{a} \rangle$ and $|_{\{k\}}$ abbreviates $| \circ \langle id, \underline{\{k\}} \rangle$.

Written in this way, it is easy to check that $credit_{k,a}$ is an instance of the very common "selective update" functional,

$$\begin{array}{ccc} B & & A \rightharpoonup B \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{U}_S\, f} \\ B & & A \rightharpoonup B \end{array}$$

defined thus:

$$\mathsf{U}_S\, f \stackrel{\text{def}}{=} \dagger \circ \langle id, (A \rightharpoonup f) \circ |_S \rangle \tag{334}$$

(make $A = AccId$, $f = id \times add_a$ and $S = \{k\}$).

Instead of doing the job for the **specific** case of $credit$, it will be worthwhile generalizing the calculation to $\mathsf{U}_S\, f$ so that a **class** of operation refinements is addressed in a single go.

By the way, some properties of U_S:

$$\begin{aligned}
\mathsf{U}_S\, id &= id && (335)\\
\mathsf{U}_\emptyset\, f &= id && (336)\\
\mathsf{U}_S\,(f \circ g) &= (\mathsf{U}_S\, f) \circ (\mathsf{U}_S\, g) && (337)\\
(A \rightharpoonup \underline{c}) \circ \mathsf{U}_S\, f &= A \rightharpoonup \underline{c} && (338)\\
(\mathsf{U}_S\, f) \circ (A \rightharpoonup g) &= (A \rightharpoonup g) \circ (\mathsf{U}_S\, h) \quad \text{if} \quad f \circ g = g \circ h && (339)
\end{aligned}$$

Moreover, remarks 48 and 49 show that diagram

$$\begin{array}{ccc} A \rightharpoonup C \times 2^B & \xrightarrow{\mathsf{U}_S(f \times id)} & A \rightharpoonup C \times 2^B \\ {\scriptstyle (252)}\uparrow & & \uparrow{\scriptstyle (252)} \\ (A \rightharpoonup C) \times 2^{A \times B} & \xrightarrow[(\mathsf{U}_S f) \times id]{} & (A \rightharpoonup C) \times 2^{A \times B} \end{array} \tag{340}$$

commutes.

The product of $\mathsf{U}_S\, f$ and id ("parallel application") at the implementation-level means that only the first data-structure $A \rightharpoonup C$ of $(A \rightharpoonup C) \times 2^{A \times B}$ will be subject to update.

In the case of $credit$, this is the "amounts table", as expected.

Since

$$(248) = (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ (mkf \times id)$$

REMARK 48: **Generic Selective Update Calculations (1)**

We have seen that $credit_{k,a} = \mathsf{U}_{\{k\}}(id \times add_a)$. Other operations on $Amounts$ will be instances of (334), for instance, the operation which adds further account holders to an account:

$$\begin{array}{rcl} addAccHolders_{k,s} & : & Accounts \longrightarrow Accounts \\ addAccHolders_{k,s} & \stackrel{\mathrm{def}}{=} & \dagger \circ \langle id, (AccId \rightharpoonup \cup \circ \langle id, \underline{s} \rangle \times id) \circ |_{\{k\}} \rangle \end{array}$$

In this remark we address the problem of refining $\mathsf{U}_S(f \times g)$, meaning to apply such generic refinement solutions to $credit$ or other instances of $\mathsf{U}_S(f \times g)$.
Our first result concerns the refinement of $\mathsf{U}_S(f \times g)$ under law (168):

$$\begin{tikzcd} A \rightharpoonup C \times D \arrow[r, "\mathsf{U}_S(f \times g)"] & A \rightharpoonup C \times D \\ (A \rightharpoonup C) \times (A \rightharpoonup D) \arrow[u, "\bowtie"] \arrow[r, "x"'] & (A \rightharpoonup C) \times (A \rightharpoonup D) \arrow[u, "\bowtie"'] \end{tikzcd}$$

We have:

$$\begin{array}{ll} \mathsf{U}_S(f \times g) \circ \bowtie & = \dagger \circ \langle id, (A \rightharpoonup f \times g) \circ |_S \rangle \circ \bowtie \\ = & \quad \{ \text{ by } \times\text{-fusion (42) } \} \\ & \dagger \circ \langle \bowtie, (A \rightharpoonup f \times g) \circ |_S \circ \bowtie \rangle \\ = & \quad \{ \text{ by permutative law (203) } \} \\ & \dagger \circ \langle \bowtie, (A \rightharpoonup f \times g) \circ \bowtie \circ (|_S \times |_S) \rangle \\ = & \quad \{ \bowtie \text{ is binatural transformation (188) } \} \\ & \dagger \circ \langle \bowtie, \bowtie \circ ((A \rightharpoonup f) \times (A \rightharpoonup g)) \circ (|_S \times |_S) \rangle \\ = & \quad \{ \text{ by } \times\text{-absroption (43) } \} \\ & \dagger \circ (\bowtie \times \bowtie) \circ \langle id, ((A \rightharpoonup f) \times (A \rightharpoonup g)) \circ (|_S \times |_S) \rangle \\ = & \quad \{ \text{ by (190) } \} \\ & \bowtie \circ \langle \dagger \circ (\pi_1 \times \pi_1), \dagger \circ (\pi_2 \times \pi_2) \rangle \circ \langle id, ((A \rightharpoonup f) \times (A \rightharpoonup g)) \circ (|_S \times |_S) \rangle \\ = & \quad \{ \text{ twice (42) and (43) } \} \\ & \bowtie \circ \langle \dagger \circ \langle \pi_1, \pi_1 \circ ((A \rightharpoonup f) \times (A \rightharpoonup g)) \circ (|_S \times |_S) \rangle, \dagger \circ \langle \pi_2, \pi_2 \circ ((A \rightharpoonup f) \times (A \rightharpoonup g)) \circ \\ & (|_S \times |_S) \rangle \rangle \\ = & \quad \{ \times \text{ is bifunctor, (32) and (33) } \} \\ & \bowtie \circ \langle \dagger \circ \langle \pi_1, ((A \rightharpoonup f) \circ |_S) \circ \pi_1 \rangle, \dagger \circ \langle \pi_2, ((A \rightharpoonup g) \circ |_S) \circ \pi_2 \rangle \rangle \\ = & \quad \{ \text{ (42) twice } \} \\ & \bowtie \circ \langle \dagger \circ \langle id, (A \rightharpoonup f) \circ |_S \rangle \circ \pi_1, \dagger \circ \langle id, (A \rightharpoonup f) \circ |_S \rangle \circ \pi_2 \rangle \\ = & \quad \{ \text{ definition (334) twice } \} \\ & \bowtie \circ \langle \mathsf{U}_S\, f \circ \pi_1, \mathsf{U}_S\, g \circ \pi_2 \rangle \\ = & \quad \{ \text{ by (34) } \} \\ & \bowtie \circ (\mathsf{U}_S\, f \times \mathsf{U}_S\, g) \end{array}$$

We obtain solution $x = \mathsf{U}_S\, f \times \mathsf{U}_S\, g$, that is, fact

$$\mathsf{U}_S(f \times g) \circ \bowtie \quad = \quad \bowtie \circ (\mathsf{U}_S\, f \times \mathsf{U}_S\, g) \qquad (341)$$

holds, a result which, in a sense, extends natural transformation (188). Therefore, selective update commutes with $\times$ under data refinement law (168) — via $\bowtie$ — meaning that both the f-update and the g-update can run "in parallel".

REMARK 49: **Generic Selective Update Calculations (2)**

Our next refinement exercise addresses the implementation of $\mathsf{U}_S(f \times g)$ under data refinement captured by abstraction map (252),

$$\begin{tikzcd}
A \rightharpoonup C \times 2^B \arrow[r, "\mathsf{U}_S(f\times g)"] & A \rightharpoonup C \times 2^B \\
(A \rightharpoonup C) \times 2^{A\times B} \arrow[u, "(252)"] \arrow[r, "x"'] & (A \rightharpoonup C) \times 2^{A\times B} \arrow[u, "(252)"']
\end{tikzcd}$$

— recall Remark 31. We will be particularly interested in the situation where $g = id$.
Our refinement equation is, then,

$$\mathsf{U}_S(f \times g) \circ (252) \quad = \quad (252) \circ x$$

Let us abbreviate $((A \rightharpoonup \underline{\emptyset}) \times collect)$ to β in (252). We have:

$$\begin{array}{rcl}
\mathsf{U}_S(f \times g) \circ (252) & = & \mathsf{U}_S(f \times g) \circ \bowtie \circ \langle \pi_1, \dagger \circ \beta \rangle \\
& = & \qquad \{ \text{ by (341)} \} \\
& & \bowtie \circ (\mathsf{U}_S\, f \times \mathsf{U}_S\, g) \circ \langle \pi_1, \dagger \circ \beta \rangle \\
& = & \qquad \{ \text{ for } g = id \ \} \\
& & \bowtie \circ (\mathsf{U}_S\, f \times \mathsf{U}_S\, id) \circ \langle \pi_1, \dagger \circ \beta \rangle \\
& = & \qquad \{ \text{ by (335) } \} \\
& & \bowtie \circ (\mathsf{U}_S\, f \times id) \circ \langle \pi_1, \dagger \circ \beta \rangle \\
& = & \qquad \{ \text{ by } \times\text{-absorption (43) and identity } \} \\
& & \bowtie \circ \langle (\mathsf{U}_S\, f) \circ \pi_1, \dagger \circ \beta \rangle \\
& = & \qquad \{ \text{ by (32) via the introduction of } id \ \} \\
& & \bowtie \circ \langle \pi_1 \circ (\mathsf{U}_S\, f \times id), \dagger \circ \beta \rangle \\
& = & \qquad \{ \text{ by (344) } \} \\
& & \bowtie \circ \langle \pi_1 \circ (\mathsf{U}_S\, f \times id), \dagger \circ \beta \circ (\mathsf{U}_S\, f \times id) \rangle \\
& = & \qquad \{ \text{ by } \times\text{-fusion (42) } \} \\
& & \underbrace{\bowtie \circ \langle \pi_1, \dagger \circ \beta \rangle}_{(252)} \circ \underbrace{(\mathsf{U}_S\, f \times id)}_{x}
\end{array}$$

In summary:

$$\mathsf{U}_S(f \times id) \circ (252) \quad = \quad (252) \circ (\mathsf{U}_S\, f \times id) \tag{342}$$

holds, that is, generic refinement diagram (340) commutes.

Exercise 29 In the context of the calculations above, check the following equalities:

$$\dagger \circ \langle f, |_S \circ f \rangle \quad = \quad f \tag{343}$$

$$\dagger \circ \beta \circ (\mathsf{U}_S\, f \times id) \quad = \quad \dagger \circ \beta \tag{344}$$

□

Exercise 30 Prove fact (339). (**Suggestion:** resort, among others, to facts (208) and (210).)
□

some extra calculation effort is needed:

$$
\begin{array}{ll}
& credit_{k,q} \circ (248) \\
= & \quad \{ \text{ expanding definitions } \} \\
& \mathsf{U}_{\{k\}}(id \times add_a) \circ (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ (mkf \times id) \\
= & \quad \{ \ (id \times add_a) \circ \langle \pi_2, \pi_1 \rangle = (add_a \times id) \text{ by (64), then (339) } \} \\
& (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ \mathsf{U}_{\{k\}}(add_a \times id) \circ (252) \circ (mkf \times id) \\
= & \quad \{ \text{ by commutative diagram (340) } \} \\
& (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ (\mathsf{U}_{\{k\}} add_a \times id) \circ (mkf \times id) \\
= & \quad \{ \text{ bifunctor } \times \text{ commutes with composition and identity (5) } \} \\
& (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ ((\mathsf{U}_{\{k\}} add_a) \circ mkf \times id) \\
= & \quad \{ \text{ expanding selective update definition (334) and composing } \} \\
& (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ (\dagger \circ \langle mkf, (AccId \rightharpoonup add_a) \circ |_{\{k\}} \circ mkf \rangle \times id) \\
= & \quad \{ \text{ by (206) } \} \\
& (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ \\
& (\dagger \circ \langle mkf, (AccId \rightharpoonup add_a) \circ mkf \circ sel_{(i_1\ \{k\})} \rangle \times id) \\
= & \quad \{ \text{ by (207) } \} \\
& (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ (\dagger \circ \langle mkf, mkf \circ 2^{AccId \times add_a} \circ sel_{(i_1\ \{k\})} \rangle \times id) \\
= & \quad \{ \text{ by (204) } \} \\
& (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ \\
& (\dagger \circ (mkf \times mkf) \circ \langle id, 2^{AccId \times add_a} \circ sel_{(i_1\ \{k\})} \rangle \times id) \\
= & \quad \{ \ \textit{cf. } \times\text{-absorption (43) } \} \\
& (AccId \rightharpoonup \langle \pi_2, \pi_1 \rangle) \circ (252) \circ (mkf \times id) \circ \\
& (update \circ \langle id, 2^{AccId \times add_a} \circ sel_{(i_1\ \{k\})} \rangle \times id) \\
= & \quad \{ \text{ definition of (248) } \} \\
& (248) \circ (\underbrace{update \circ \langle id, 2^{AccId \times add_a} \circ sel_{(i_1\ \{k\})} \rangle}_{\lambda r. sqlUpdate(r, 2, add_a, \lambda n. n=k)} \times id) \\
= & \quad \{ \ \textit{cf. } \text{CAMILA library } \texttt{sql.cam}\text{, in this context } \} \\
& (248) \circ (\lambda r. [\![\texttt{UPDATE r SET Amount=Amount+a WHERE AccId=k }]\!] \times id)
\end{array}
$$

In summary, we have

$$credit_{k,a} \circ (248) =$$
$$(248) \circ (\lambda r. [\![\texttt{UPDATE r SET Amount=Amount+a WHERE AccId=k}]\!] \times id)$$

that is,

$$credit(k, a, (\sigma, \rho)) \stackrel{\mathrm{def}}{=}$$
$$([\![\texttt{UPDATE } \sigma \texttt{ SET Amount=Amount+a WHERE AccId=k}]\!], \rho)$$

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.

Comments:

- A formal semantics of SQL, which has to become explicit at this stage, has been defined by [Rod93] in the context of **Set**-notation.
- 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.

Example 2 — A Local Invariant Dependent Calculation

Our example concerning local invariants will be the refinement of the $find$ operation (296) at hash-table level, that is, under abstraction function f (314) and concrete invariant Φ_2 (317).

Refinement diagram is:

$$\begin{tikzcd} A \times 2^A \arrow[r, "find"] & 2 \\ A \times ((2^A)^n)_{\Phi_2} \arrow[u, "id_A \times (f_1 \circ f_2)"] \arrow[r, "find_2"'] & 2 \arrow[u, "id_2"'] \end{tikzcd}$$

leading to refinement equation:

$$find_2(a, t) = find(a, f_1(f_2\, t))$$

REMARK 50: **A theorem concerning local invariant preservation**

Theorem 10. *(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,*

$$\forall a \in A : \alpha\, a \Rightarrow \alpha(\sigma\, a) \tag{345}$$

Then the reification σ' of σ implied by reification step (161),

$$\begin{tikzcd} A \arrow[r, "\sigma"] & A \\ B_\phi \arrow[u, "f"] \arrow[r, "\sigma'"'] & B_\phi \arrow[u, "f"'] \end{tikzcd}$$

will still preserve α at B-level.

Proof. σ' is any function satisfying

$$\forall b \in B_\phi : f(\sigma'\, b) = \sigma(f\, b) \tag{346}$$

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 (346). 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 (345). □

Exercise 31 Calculate the relational implementation, at $Amounts_5$ level (see p.69), of the "remove account" operator

$$remAcc(k, \sigma) \stackrel{\rm def}{=} \sigma \setminus \{k\} \tag{347}$$

defined over its ancestor datatype ($Amounts$). That is to say, solve for $remAcc_5$ the following refinement diagram:

$$\begin{tikzcd} AccId \times Amounts \arrow[r, "remAcc"] & Amounts \\ AccId \times Amounts_5 \arrow[u, "id \times (248)"] \arrow[r, "remAcc_5"'] & Amounts_5 \arrow[u, "(248)"'] \end{tikzcd}$$

where **Hint**: It may help to consider relational operator rem (231) for tuple-deletion, whose semantics is opposite to that of sel (55).
□

Reasoning is carried out as follows:

$$\begin{array}{rcl} find_2(a,t) & = & find(a, f_1(f_2\, t)) \\ & = & a \in f_1(f_2\, t) \\ & = & a \in \bigcup_{i \in n} t\, i \\ & = & \bigvee_{i \in n} a \in t\, i \\ & = & \exists i \in n : a \in t\, i \end{array}$$

Now, taking invariant Φ_2 into account, one has:

$$\begin{array}{rcl} \bigvee_{i \in n} a \in t\, i & = & a \in t(H\, a) \vee \bigvee_{i \in n \wedge i \neq H\, a} a \in t\, i \\ & = & (a \in t(H\, a) \vee \bigvee_{i \in n \wedge i \neq H\, a} \mathsf{FALSE} \\ & = & a \in t(H\, a) \\ & = & find(a, t(H\, a)) \end{array}$$

In summary,

$$find_2(a,t) \quad = \quad find(a, t(H\, a))$$

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 $\bigcup_{i \in n} t\, i$, the efficiency gain is obvious.

See remarks 51, 52 and 53 for the calculation of the other operations specified in (296).
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

Consider the fairly academic refinement equation

$$belongs(x, y) = x \in elems\, y \tag{348}$$

relative to the diagram of the $find$ operation,

$$find(x, y) = x \in y$$

REMARK 51: **Hash-table operation synthesis: the $insert$ function**

Refinement diagram:

$$
\begin{tikzcd}
A \times 2^A \arrow[r, "insert"] & 2^A \\
A \times ((2^A)^n)_{\Phi_2} \arrow[u, "id_A \times (f_1 \circ f_2)"] \arrow[r, "insert_2"'] & ((2^A)^n)_{\Phi_2} \arrow[u, "f_1 \circ f_2"']
\end{tikzcd}
$$

Refinement equation:

$$f_1(f_2(insert_2(a,t))) = insert(a, f_1(f_2\,t))$$

Reasoning:

$$
\begin{array}{rcl}
f_1(f_2(insert_2(a,t))) & = & insert(a, f_1(f_2\,t)) \\
& = & \{a\} \cup f_1(f_2\,t) \\
& = & \{a\} \cup \bigcup_{i \in n} t\,i \\
& = & \{a\} \cup t(H\,a) \cup \bigcup_{i \in n - \{H\,a\}} t\,i \\
& = & t'(H\,a) \cup \bigcup_{i \in n - \{H\,a\}} t\,i
\end{array}
$$

where t' is the one-entry mapping

$$t' \quad = \quad \left(\begin{array}{c} H\,a \\ \{a\} \cup t(H\,a) \end{array} \right)$$

Then,

$$
\begin{array}{rcl}
t'(H\,a) \cup \bigcup_{i \in n - \{H\,a\}} t\,i & = & \bigcup_{i \in n} \left\{ \begin{array}{rcl} i = H\,a & \Rightarrow & t'\,i \\ \neg(i = H\,a) & \Rightarrow & t\,i \end{array} \right. \\
& = & \bigcup_{i \in n} \left\{ \begin{array}{rcl} i \in dom\,t' & \Rightarrow & t'\,i \\ \neg(i \in dom\,t') & \Rightarrow & t\,i \end{array} \right. \\
& = & \bigcup_{i \in n} (t \dagger t')i \\
& = & f_1(f_2(t \dagger t'))
\end{array}
$$

In summary,

$$f_1(f_2(insert_2(a,t))) \quad = \quad f_1(f_2(t \dagger \left(\begin{array}{c} H\,a \\ \{a\} \cup t(H\,a) \end{array} \right)))$$

or, by removing $f_1 \circ f_2$ from both sides of the equality,

$$insert_2(a,t) \quad = \quad t \dagger \left(\begin{array}{c} H\,a \\ insert(a, t(H\,a)) \end{array} \right)$$

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 (315).

REMARK 52: **Hash-table operation synthesis: the $initialDb$ and $remove$ functions**

Refinement diagram for $initialDb$:

$$\begin{tikzcd}
{} \arrow[r, "initialDb"] & 2^A \\
{} \arrow[u] \arrow[r, "initialDb_2"'] & ((2^A)^n)_{\Phi_2} \arrow[u, "f_1 \circ f_2"']
\end{tikzcd}$$

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:

$$\begin{tikzcd}
A \times 2^A \arrow[r, "remove"] & 2^A \\
A \times ((2^A)^n)_{\Phi_2} \arrow[u, "id_A \times f_1 \circ f_2"] \arrow[r, "remove_2"'] & ((2^A)^n)_{\Phi_2} \arrow[u, "f_1 \circ f_2"']
\end{tikzcd}$$

Reasoning:

$$\begin{aligned}
f_1(f_2(remove_2(a, t))) &= remove(a, f_1(f_2\, t)) = f_1(f_2\, t) - \{a\} \\
&= \bigcup_{i \in n} t\, i - \{a\} = (\bigcup_{i \in n} t\, i) \cap \overline{\{a\}} \\
&= \bigcup_{i \in n} (t\, i \cap \overline{\{a\}}) = \bigcup_{i \in n} (t\, i - \{a\}) \\
&= (t(H\, a) - \{a\}) \cup \bigcup_{i \in n - \{H\, a\}} (t\, i - \{a\}) = t'(H\, a) \cup \bigcup_{i \in n - \{H\, a\}} \underbrace{t\, i - \{a\}}_{(*)}
\end{aligned}$$

(to be continued in Remark 53)

in the “sets replaced by lists” refinement:

$$\begin{tikzcd}
A \times 2^A \arrow[r, "find"] & 2 \\
A \times A^\star \arrow[u, "id_A \times elems"] \arrow[r, "belongs"'] & 2 \arrow[u, "id_2"']
\end{tikzcd}$$

We want to calculate an iterative version of $belongs$.

CALCULATION — STEP 1

Start by currying (348) on the first parameter:

$$\begin{tikzcd}
2^A \arrow[r, "find_x"] & 2 \\
A^\star \arrow[u, "elems"] \arrow[r, "belongs_x"'] & 2 \arrow[u, "id_2"']
\end{tikzcd} \qquad belongs_x = find_x \circ elems$$

Recall that $elems = (\![\ \underline{\emptyset}, puts\ ]\!)_{\mathsf{F}}$, for $\mathsf{F}\,X = 1 + A \times X$:

$$\begin{tikzcd}
A^\star \arrow[r, shift left, "\iota^{-1}"] \arrow[d, "elems = (\![\ \underline{\emptyset}, puts\ ]\!)_{\mathsf{F}}"'] & 1 + A \times A^\star \arrow[l, shift left, "\iota"] \arrow[d, "1 + A \times elems"] \\
2^A & 1 + A \times 2^A \arrow[l, "{[\ \underline{\emptyset}, puts\ ]}"]
\end{tikzcd}$$

Then compose this with F-algebra $(2, [\ \underline{\mathsf{FALSE}}, test_x\ ])$:

$$\begin{tikzcd}
A^\star \arrow[d, "elems"] \arrow[dd, bend right, "belongs_x"'] & 1 + A \times A^\star \arrow[l, "{[\ nil, cons\ ]}"'] \arrow[d, "\mathsf{F}\, elems"] \\
2^A \arrow[d, "find_x"] & 1 + A \times 2^A \arrow[l, "{[\ \underline{\emptyset}, puts\ ]}"] \arrow[d, "1 + A \times find_x"] \\
2 & 1 + A \times 2 \arrow[l, "{[\ \underline{\mathsf{FALSE}}, test_x\ ]}"]
\end{tikzcd}$$

where $test_x \stackrel{\rm def}{=} \vee \circ (=_x \times id)$.

It is easy to check that the bottom square commutes:

$$find_x \circ [\ \underline{\emptyset}, puts\ ] = [\ \underline{\mathsf{FALSE}}, test_x\ ] \circ (1 + A \times find_x)$$

So, by catamorphism-fusion (150) we have

$$belongs_x = (\![\ \underline{\mathsf{FALSE}}, test_x\ ]\!)_{\mathsf{F}}$$

that is, going pointwise and uncurried:

$$
\begin{array}{l}
belongs(x, y) \stackrel{\rm def}{=} \\
\quad \left\{ \begin{array}{lcl} y = [\,] & \Rightarrow & \mathsf{FALSE} \\ y \neq [\,] & \Rightarrow & x = head\, y \vee belongs(x, tail\, y) \end{array} \right.
\end{array}
\tag{349}
$$

Abstraction function $elems$ has vanished! — so we have completed step 1 of the proposed method.

CALCULATION — STEP 2

Next, we will be concerned with efficiency and want to eliminate recursion in (349). Let $beloop_x$ be any function satisfying the following axiom:

$$
beloop_x(y, b) = b \vee belongs_x\, y \tag{353}
$$

It is easy to check the following instantiation of (353),

$$
beloop_x(y, \mathsf{FALSE}) = belongs_x\, y \tag{354}
$$

since we know that algebraic structure

$$
(\{\mathsf{TRUE}, \mathsf{FALSE}\}; \vee, \mathsf{FALSE})
$$

is a monoid. Moreover, for $y = [\,]$, we have

$$
\begin{array}{rcl}
beloop_x([\,], b) & = & b \vee belongs_x\, [\,] \\
& = & b \vee \mathsf{FALSE} \\
& = & b
\end{array}
\tag{355}
$$

as well as, for $y \neq [\,]$,

$$
\begin{array}{rcl}
beloop_x(y, b) & = & b \vee (x = head\, y \vee belongs_x(tail\, y)) \\
& = & (b \vee (x = head\, y)) \vee belongs_x(tail\, y) \\
& = & beloop_x(tail\, y, (b \vee (x = head\, y)))
\end{array}
\tag{356}
$$

cf. (353). Putting (355) and (356) together, we obtain

$$
\begin{array}{l}
beloop_x(y, b) \stackrel{\rm def}{=} \\
\quad \left\{ \begin{array}{lcl} y = [\,] & \Rightarrow & b \\ y \neq [\,] & \Rightarrow & beloop_x(tail\, y, b \vee (x = head\, y)) \end{array} \right.
\end{array}
\tag{357}
$$

Furthermore, we know that TRUE is the “zero” of $\vee$:

$$
b \vee \mathsf{TRUE} = \mathsf{TRUE} \vee b = \mathsf{TRUE}
$$

REMARK 53: **Hash-table operation synthesis: the $remove$ function (conclusion) and final comments**

(Continued from Remark 52)
Note that t' above is the one-entry mapping

$$t' = \left(\begin{array}{c} H\,a \\ t(H\,a) - \{a\} \end{array} \right)$$

At this point one resorts to invariant Φ_2 (317), 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 (348) above to $t\,i$. The reasoning here-upon is similar to that carried out for $insert$, leading to

$$f_1(f_2(remove_2(a,t))) = f_1(f_2(t \dagger \left(\begin{array}{c} H\,a \\ t(H\,a) - \{a\} \cup t(H\,a) \end{array} \right)))$$

i.e. (by removing $f_1 \circ f_2$ from both sides of the equality)

$$remove_2(a,t) = t \dagger \left(\begin{array}{c} H\,a \\ remove(a, t(H\,a) \end{array} \right)$$

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 (295), is far too simple a model for database files. A model closer to reality is

$$Database' = A \rightharpoonup B \tag{350}$$

where A plays the rôle of a domain of keys and B is the data of interest.
Note that $2^A \cong_{dom} A \rightharpoonup 1$ (167). So it seems that the new model amounts to generalizing 1 in functor $\lambda X.X \rightharpoonup 1$ to any finite B. And indeed each operation σ over (295) can be generalized to some σ' over (350) provided that σ' "preserves σ along the dom morphism", that is, if σ is such that

$$dom\,\sigma'(x, \dots) = \sigma(dom\,x, \dots)$$

With respect to data refinement, we know that functor $\lambda X.X \rightharpoonup B$ behaves well provided that it is applied to injective functions, see p.30.
Reference [Oli94] includes a thorough treatment of this extension and a generalization of the "hash effect" to arbitrary $\mathbf{Set}$-functors which obey to the following property:

$$\mathsf{G}(n \times A) \trianglelefteq (\mathsf{G}\,A)^n \tag{351}$$

for $n > 0$ and a given hash function $H : A \longrightarrow n$. This opens the hashing optimization to many other data-structures. However, for $\mathsf{G}\,X = X^*$ we apparently have

$$(n \times A)^* \cong (A^*)^n \tag{352}$$

since both sides of (352) 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 (352), the original interleaving of such sequences. See [Oli94] for further discussion on this topic.

So, via (353),

$$beloop_x(y, \mathsf{TRUE}) = \mathsf{TRUE}$$

i.e.

$$(b = \mathsf{TRUE}) \Rightarrow beloop_x(y, b) = b$$

We take advantage of this in the follow up of (357) via (380):

$$\begin{array}{l} beloop_x(y,b) \stackrel{\mathrm{def}}{=} \\ \qquad \left\{ \begin{array}{lcl} y = [\,] & \Rightarrow & \left\{ \begin{array}{rcl} b & \Rightarrow & b \\ \neg b & \Rightarrow & b \end{array} \right. \\ y \neq [\,] & \Rightarrow & \left\{ \begin{array}{rcl} b & \Rightarrow & b \\ \neg b & \Rightarrow & beloop_x(tail\, y, b \vee (x = head\, y)) \end{array} \right. \end{array} \right. \end{array}$$

that is,

$$\begin{array}{l} beloop_x(y,b) \stackrel{\mathrm{def}}{=} \\ \qquad \left\{ \begin{array}{rcl} y = [\,] \wedge b & \Rightarrow & b \\ y = [\,] \wedge \neg b & \Rightarrow & b \\ y \neq [\,] \wedge b & \Rightarrow & b \\ y \neq [\,] \wedge \neg b & \Rightarrow & beloop_x(tail\, y, b \vee (x = head\, y)) \end{array} \right. \end{array}$$

or, at last,

$$\begin{array}{l} beloop_x(y,b) \stackrel{\mathrm{def}}{=} \\ \qquad \left\{ \begin{array}{rcl} y = [\,] \vee b & \Rightarrow & b \\ y \neq [\,] \wedge \neg b & \Rightarrow & beloop_x(tail\, y, (x = head\, y)) \end{array} \right. \end{array} \tag{358}$$

cf. (381).

Finally, we put (354) together with (358) 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)                        (359)
                  };
  }
}
```

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.
- See the program transformation literature for 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

$$\begin{array}{rcl} f & : & \mathsf{F}Y \longrightarrow M \\ f & \stackrel{\mathrm{def}}{=} & p \rightarrow u, \theta \circ \langle d, f \circ e \rangle \end{array} \tag{360}$$

be an arbitrary function where the following abstract operators occur,

$$\begin{array}{rcl} p & : & \mathsf{F}Y \longrightarrow 2 \\ e & : & \mathsf{F}Y \longrightarrow \mathsf{F}Y \\ d & : & \mathsf{F}Y \longrightarrow M \\ \theta & : & M \times M \longrightarrow M \\ u & : & 1 \longrightarrow M \end{array}$$

such that $(M;\theta,u)$ is a monoid, that is,

$$\theta\circ\langle i,u\rangle = \theta\circ\langle u,i\rangle = i$$
$$\theta\circ\langle\theta\circ\langle i,j\rangle,k\rangle = \theta\circ\langle i,\theta\circ\langle j,k\rangle\rangle$$

hold for arbitrary $\cdots \xrightarrow{i,j,k} M$.

Comments:

- Patterns such as (360) are called *program schemata*. Schema (360), which is called the *linear monadic schema* (LMS), is nothing but the following abstract hylomorphism,

$$\begin{tikzcd}
\mathsf{F}Y \arrow[d, "f"'] \arrow[rr, bend left, "(!+\langle d_1,e\rangle)\circ p?"] & 1+M\times\mathsf{F}Y \arrow[d, "1+M\times f"'] & 1+Y\times\mathsf{F}Y \arrow[l, "1+d_2\times id_{\mathsf{F}Y}"'] \arrow[d, "1+Y\times f"] \\
M & 1+M\times M \arrow[l, "{[\,u,\theta\,]}"] & 1+Y\times M \arrow[l, "1+d_2\times id_M"]
\end{tikzcd}$$

  for $d = d_2\circ d_1$.

- Check that $belongs_x$ matches this schema for

LMS	$belongs_x$
Y	A
$\mathsf{F}Y$	$A^\star$
M	2
p	$=_{[\,]}$
e	$tail$
d_1	$head$
d_2	$=_x$
θ	$\vee$
u	FALSE

Wishing to generalize our calculation of $belongs$ to f, we introduce a function $\mathsf{F}Y\times M \xrightarrow{floop} M$,

$$floop(..y..,r) \stackrel{\rm def}{=} r\,\theta\,f(..y..) \tag{361}$$

which, because $(M;\theta,u)$ is a monoid, will satisfy:

$$f(..y..) = floop(..y..,u) \tag{362}$$

The following reasoning is a consequence of (361), of the substitution property, of rule (378) and of the monoidal properties of $(M; \theta, u)$:

$$
\begin{array}{rcl}
floop(..y.., r) & = & r\,\theta\, f(..y..) \\
& = & r\,\theta \left\{ \begin{array}{rcl} p(..y..) & \Rightarrow & u \\ \neg(p(..y..)) & \Rightarrow & d(..y..)\,\theta\, f(..e\,y..) \end{array} \right. \\
& = & \left\{ \begin{array}{rcl} p(..y..) & \Rightarrow & r\,\theta\, u \\ \neg p(..y..) & \Rightarrow & r\,\theta\,(d(..y..)\,\theta\, f(..e\,y..)) \end{array} \right. \\
& = & \left\{ \begin{array}{rcl} p(..y..) & \Rightarrow & r \\ \neg p(..y..) & \Rightarrow & \underbrace{(r\,\theta\, d(..y..))\,\theta\, f(..e\,y..)}_{A} \end{array} \right.
\end{array}
$$

By instantiation of (361) we get

$$A = floop(..e\,y.., r\,\theta\, d(..y..))$$

In summary, we have

$$
floop(..y.., r) \;=\; \left\{ \begin{array}{rcl} p(..y..) & \Rightarrow & r \\ \neg p(..y..) & \Rightarrow & floop(..e\,y.., r\,\theta\, d(..y..)) \end{array} \right.
$$

which, together with the "loop initialization" given by (362), 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:

$$
\begin{array}{rcl}
floop(..y.., r) & = & \left\{ \begin{array}{rcl} p(..y..) & \Rightarrow & r \\ \neg p(..y..) & \Rightarrow & \left\{ \begin{array}{rcl} r = a & \Rightarrow & a \\ r \neq a & \Rightarrow & floop(..e\,y.., r\,\theta\, d(..y..)) \end{array} \right. \end{array} \right. \\
& = & \left\{ \begin{array}{rcl} p(..y..) \vee r = a & \Rightarrow & r \\ \neg p(..y..) \wedge r \neq a & \Rightarrow & floop(..e\,y.., r\,\theta\, d(..y..)) \end{array} \right.
\end{array}
$$

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 $belongs$ (359) is an obvious instance.

Examples 3.1 — Set and List Browsing

Recall Zermelo-Frænkl set-comprehension formula,

$$\{g\,x \mid\; x \in X \wedge p\,x\} \tag{363}$$

and list comprehension,

$$[g\,x \mid x \leftarrow L \wedge p\,x] \tag{364}$$

given $A \xrightarrow{g} B$, $X \subseteq A$, $L \in A^*$ and "filter" $A \xrightarrow{p} 2$.

It can be shown (*e.g.* by induction) that

- hylomorphism

$$\begin{tikzcd}
2^A \arrow[d, "f"'] \arrow[rr, bend left, "(!+gets)\circ{=_\emptyset}?"] & 1 + 2^B \times 2^A \arrow[d, "1+2^B\times f"] & 1 + A \times 2^A \arrow[l, "1+d_2\times id_{2^A}"'] \arrow[d, "1+A\times f"] \\
2^B & 1 + 2^B \times 2^B \arrow[l, "{[\,\underline{\emptyset},\cup\,]}"] & 1 + A \times 2^B \arrow[l, "1+d_2\times id_{2^B}"]
\end{tikzcd}$$

  where

$$d_2 = p \rightarrow sin \circ g, \underline{\emptyset}$$

for $sin\, x \stackrel{\mathrm{def}}{=} \{x\}$ — that is, recursive function

$$
f\, y \stackrel{\mathrm{def}}{=}
\left\{
\begin{array}{lll}
y = \emptyset & \Rightarrow & \emptyset \\
y \neq \emptyset & \Rightarrow & \begin{array}{ll} let & e \in y \\ & a = \left\{ \begin{array}{rcl} p\, e & \Rightarrow & \{g\, e\} \\ \neg(p\, e) & \Rightarrow & \emptyset \end{array} \right. \\ in & a \cup f(y - \{e\}) \end{array}
\end{array}
\right.
$$

when applied to X, yields the same set as ZF-formula above;

- hylomorphism

$$
\begin{tikzcd}
A^\star \arrow[rr, bend left, "(!+\langle head,tail\rangle)\circ=_{[\,]}?"] \arrow[d, "f"'] & 1 + B^\star \times A^\star \arrow[d, "1+B^\star\times f"'] & 1 + A \times A^\star \arrow[l, "1+d_2\times id_{A^\star}"'] \arrow[d, "1+A\times f"] \\
B^\star & 1 + B^\star \times B^\star \arrow[l, "{[\,\underline{[\,]},+\!\!+\,]}"'] & 1 + A \times B^\star \arrow[l, "1+d_2\times id_{B^\star}"']
\end{tikzcd}
$$

where

$$
d_2 = p \rightarrow sinl \circ g, \underline{[\,]}
$$

for $sinl\, x \stackrel{\mathrm{def}}{=} [x]$ — that is, recursive function

$$
f\, y \stackrel{\mathrm{def}}{=}
\left\{
\begin{array}{lll}
y = [\,] & \Rightarrow & [\,] \\
y \neq [\,] & \Rightarrow & \begin{array}{ll} let & x = head\, y \\ & a = \left\{ \begin{array}{rcl} p\, x & \Rightarrow & [g\, x] \\ \neg(p\, x) & \Rightarrow & [\,] \end{array} \right. \\ in & a +\!\!+ f(tail\, y) \end{array}
\end{array}
\right.
$$

when applied to L, yields the same list as formula (364) above.

Moreover, both recursive functions above are instances of linear-monadic scheme (360).

So ... they "are" loops!

Simple instantiation of (360) will show that second f above, 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,g(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(g(x)); }
}
```

Of course, a similar `while`-loop schema can be developed for (363).

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.72 is that which computes the *part explosion* of one equipment according to its production tree:

$$
\begin{array}{rcl}
explode & : & \#Equip \times PPD \longrightarrow (\#Comp \rightharpoonup I\!N) \\
explode(\#e, \sigma) & \stackrel{\rm def}{=} & let \quad \tau = apply(\sigma, \#e) \\
& & in \quad \bigoplus_{x \in dom\,\tau} \left\{ \begin{array}{lcl} x = i_1\,k & \Rightarrow & \begin{pmatrix} k \\ \tau\,x \end{pmatrix} \\ x = i_2\,k & \Rightarrow & (\tau\,x) \otimes explode(k, \sigma) \end{array} \right.
\end{array}
$$

where

- $\oplus$ is multiset ("bag") union:

$$
\begin{array}{rcl}
x \oplus y & \stackrel{\rm def}{=} & \quad x \setminus (dom\,y) \\
& & \cup \;\; y \setminus (dom\,x) \\
& & \cup \;\; \begin{pmatrix} a \\ (x\,a) + (y\,a) \end{pmatrix}_{a \in dom\,x \cap dom\,y}
\end{array}
\qquad (365)
$$

- $\otimes$ is called multiset external multiplication (by analogy with vector spaces, see Exercise 32):

$$
\begin{array}{rcl}
\otimes & : & I\!N \times (A \rightharpoonup I\!N) \longrightarrow (A \rightharpoonup I\!N) \\
n \otimes \sigma & \stackrel{\rm def}{=} & (A \rightharpoonup (\lambda a.n \times a))\sigma
\end{array}
\qquad (366)
$$

- $apply$ is the following auxiliary function:

$$
\begin{array}{rcl}
apply & : & PPD \times \#Equip \longrightarrow (\#Unid \rightharpoonup I\!N) \\
apply(\sigma, k) & \stackrel{\rm def}{=} & let \quad \rho = \pi_3\,\sigma \\
& & in \quad \left\{ \begin{array}{lcl} k \in dom\,\rho & \Rightarrow & \rho\,k \\ k \notin dom\,\rho & \Rightarrow & (\;\;) \end{array} \right.
\end{array}
$$

Eager to implement $explode$ at $RPPD$-level, suppose that from its refinement diagram we have already arrived at

$$
\begin{array}{rcl}
explode_R & : & \#Equip \times RPPD \longrightarrow (\#Comp \rightharpoonup I\!N) \\
explode_R(\#e, \gamma) & \stackrel{\rm def}{=} & let \quad \tau = apply_R(\gamma, \#e) \\
 & & in \quad \bigoplus_{x \in dom\,\tau} \left\{ \begin{array}{lcl} x = i_1\,k & \Rightarrow & \left(\begin{array}{c} k \\ \tau\,x \end{array} \right) \\ x = i_2\,k & \Rightarrow & \tau\,x \otimes explode_R(k, \gamma) \end{array} \right.
\end{array}
\tag{367}
$$

where γ is a 4-tuple $(\gamma_1, \gamma_2, \gamma_3, \gamma_4)$, recall p.76 and see Exercise 35.

Note that only PPD was refined — a "state-oriented" data refinement once again.

QUESTION

How do we implement $explode_R$? How do we implement recursion on top the the underlying relational model?

CALCULATION

The first move is to realize that

$$
explode_R(\#e, \gamma) \;=\; aux(apply_R(\gamma, \#e), \gamma) \tag{368}
$$

holds, where

$$
aux(\tau, \gamma) \stackrel{\rm def}{=} \left\{ \begin{array}{lcl}
\tau = (\;\;) & \Rightarrow & (\;\;) \\
\tau \neq (\;\;) & \Rightarrow & let \quad x \in dom\,\tau \\
 & & \qquad\; y = \left\{ \begin{array}{lcl} x = i_1\,k & \Rightarrow & \left(\begin{array}{c} k \\ \tau\,x \end{array} \right) \\ x = i_2\,k & \Rightarrow & (\tau\,x) \otimes explode_R(k, \gamma) \end{array} \right. \\
 & & in \quad\; y \oplus aux(\tau \setminus \{x\}, \gamma)
\end{array} \right.
$$

is the linear monadic version of $\bigoplus_{x \in \ldots} \ldots$ — see (371) and (372) in a remark to follow, for $\varphi = \oplus$.

However, because of the occurrence of $explode_R$ in the schema, one is still far away from iteration.

It is easy to check that the following properties

$$n \otimes aux(\tau,\gamma) \;=\; aux(n \otimes \tau,\gamma) \tag{369}$$
$$aux(\tau_1,\gamma) \oplus aux(\tau_2,\gamma) \;=\; aux(\tau_1 \oplus \tau_2,\gamma) \tag{370}$$

follow from the underlying multiset algebra (see Exercise 32).

Then, from (368) and (369) 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 re-write aux's body into:

$$\left\{\begin{array}{lcl} x = i_1\, k & \Rightarrow & \left(\begin{array}{c} k \\ \tau\, x \end{array}\right) \oplus aux(\tau \setminus \{x\},\gamma)\\ x = i_2\, k & \Rightarrow & aux(\tau\, x \otimes apply_R(\gamma,k),\gamma) \oplus aux(\tau \setminus \{x\},\gamma)\end{array}\right.$$

Now apply (370) 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

$$\left\{\begin{array}{lcl} x = i_1\, k & \Rightarrow & \left(\begin{array}{c} k \\ \tau\, x \end{array}\right)\\ x = i_2\, k & \Rightarrow & (\;\;) \end{array}\right. \oplus \underbrace{\left\{\begin{array}{lcl} x = i_1\, k & \Rightarrow & aux(\tau \setminus \{x\},\gamma)\\ x = i_2\, k & \Rightarrow & aux(\tau\, x \otimes apply_R(\gamma,k) \oplus \tau \setminus \{x\},\gamma)\end{array}\right.}_{X}$$

simply because $(\;\;)$ is the unit of $\oplus$ and according to the following rule

$$f \circ \langle (p \rightarrow q, r), (p \rightarrow s, t) \rangle \;=\; p \rightarrow f \circ \langle q, s\rangle, f \circ \langle r, t\rangle$$

combining (378) and (379). 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 (378). Eventually, we find that aux instantiates the linear monadic schema (360) 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 $\#Comp \rightharpoonup I\!N$, we stop at this "pseudo-C" formulation of our target `while`-loop,

$$
\begin{array}{l}
\{\ \texttt{Parts}\quad\ \ \texttt{r = ();} \\
\quad \texttt{Structure y' = y;} \\
\quad \texttt{while}\quad\ \ \texttt{(y' != ())} \\
\qquad\qquad\quad \{\ \texttt{Unit x = choice(dom(y'));} \\
\qquad\qquad\quad\ \ \texttt{Quantity n = y'(x);} \\
\qquad\qquad\quad\ \ \texttt{r = r}\ \oplus \left\{\begin{array}{lcl} x = i_1\,k & \Rightarrow & \left(\begin{array}{c} k \\ n \end{array}\right) \\ x = i_2\,k & \Rightarrow & (\ ) \end{array}\right. \quad \texttt{;} \\
\qquad\qquad\quad\ \ \texttt{y' = y'}\ \setminus\{x\} \oplus \left\{\begin{array}{lcl} x = i_1\,k & \Rightarrow & (\ ) \\ x = i_2\,k & \Rightarrow & n \otimes apply_R(\gamma,k) \end{array}\right. \quad \texttt{;} \\
\qquad\qquad\quad \}\texttt{;} \\
\}
\end{array}
$$

where data-types `Parts` and `Structure` model $\#Comp \rightharpoonup I\!N$ and $\#Unit \rightharpoonup I\!N$, respectively.

Putting things together — recall (368) — 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));

    }
}
```

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.

A full implementation will require a relational database-level 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 this into account.

Pragmatics of Program Calculation

Impact on Relational Database Design

REMARK 54: **Monoids as a basis for recursion introduction and elimination**

Let $(M;\theta,\epsilon)$ be an Abelian monoid, that is, θ is associative and commutative and ϵ is the unit of θ. Given some $X \xrightarrow{\varphi} M$, it will make sense to think of construction

$$2^X \xrightarrow{\lambda x.\Theta_{e\in x}\varphi\, e} M \tag{371}$$

obeying to the following properties:

$$\begin{array}{rcll} \Theta_{e\in\emptyset}\varphi\, e & = & \epsilon & \\ \Theta_{e\in x}\varphi\, e & = & (\varphi\, e_0)\theta\Theta_{e\in x-\{e_0\}}\varphi\, e & \quad\quad \text{for } e_0 \in x \end{array}$$

It can be shown that $\lambda x.\Theta_{e\in x}\varphi\, e$ is the same as hylomorphism

$$\begin{tikzcd} 2^X \arrow[d, "f"'] \arrow[rr, bend left, "(!+gets)\circ{=_\emptyset}?"] & 1+M\times 2^X \arrow[d, "1+M\times f"'] & 1+X\times 2^X \arrow[l, "1+\varphi\times M_{2^X}"] \arrow[d, "1+X\times f"] \\ M & 1+M\times M \arrow[l, "{[\,\epsilon,\theta\,]}"] & 1+X\times M \arrow[l, "1+\varphi\times M_M"] \end{tikzcd}$$

that is, the following recursive schema:

$$\begin{array}{rcl} f: 2^X & \rightarrow & M \\ f\,x & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lcll} x=\emptyset & \Rightarrow & \epsilon & \\ x\neq\emptyset & \Rightarrow & let & e\in x \\ & & in & (\varphi\, e)\theta f(x-\{e\}) \end{array} \right. \end{array} \tag{372}$$

This is called the θ-reduction algorithmic scheme and it can be regarded as a recursive implementation of (371). But, because it is an instance of the linear monadic scheme (360), recursion can be eliminated and thus an iterative implementation of the schema be obtained.

Exercise 32 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*:

$$\mathsf{Mset}\, A \quad\cong\quad A \rightharpoonup I\!N \tag{373}$$

Therefore, the empty mapping $(\quad)$ represents the empty multiset and set-union generalizes to multiplicity addition, that is, given two multisets x and y in $\mathsf{M}set\; A$, their union, denoted by $x \oplus y$, is the mapping given by

$$\begin{array}{rcll} x\oplus y & \stackrel{\text{def}}{=} & & x\setminus dom\, y \\ & & \cup & y\setminus dom\, x \\ & & \cup & \left(\begin{array}{c} a \\ (x\,a)+(y\,a) \end{array} \right)_{a\in dom\, x\cap dom\, y} \end{array} \tag{374}$$

1. Show that $\oplus$ can be alternatively defined by:

$$x\oplus y \stackrel{\text{def}}{=} x\dagger y\dagger \left(\begin{array}{c} a \\ (x\,a)+(y\,a) \end{array} \right)_{a\in dom\, x|(dom\, y)} \tag{375}$$

2. Show that $\mathsf{M}set\; A$ together with $\oplus$ and the empty mapping form an Abelian (commutative) monoid.
3. Show that multiset external multiplication (366) obeys the following properties

$$n\otimes(m\otimes\sigma) = (n\times m)\otimes\sigma \tag{376}$$
$$n\otimes(\sigma\oplus\tau) = (n\otimes\sigma)\oplus(n\otimes\tau) \tag{377}$$

□

REMARK 55: **A quick reference guide to program transformation**

Program transformation calculi arise 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]. For current standards on program transformation see [BdM97].

We start with a few properties involving conditionals.

Function application distributes *wrt.* conditional

$$f \circ (p \rightarrow q, r) \quad = \quad p \rightarrow f \circ q, f \circ r \tag{378}$$

Construction may distribute *wrt.* conditional

$$\langle (p \rightarrow q, r), (p \rightarrow s, t) \rangle \quad = \quad p \rightarrow \langle q, s \rangle, \langle r, t \rangle \tag{379}$$

Note that this is a special case of the so-called "banana-split" law of catamorphisms [BdM97].

Other rules concerning conditionals

$$p \rightarrow q, q \quad = \quad q \tag{380}$$

$$\left\{ \begin{array}{ccc} p & \Rightarrow & q \\ r & \Rightarrow & q \\ t & \Rightarrow & s \end{array} \right. \quad = \quad \left\{ \begin{array}{rcc} (p \vee r) & \Rightarrow & q \\ t & \Rightarrow & s \end{array} \right. \tag{381}$$

$$(p \rightarrow q, r) \rightarrow s, t \quad = \quad p \rightarrow (q \rightarrow s, t), (r \rightarrow s, t) \tag{382}$$

$$\left\{ \begin{array}{rcl} p & \Rightarrow & \left\{ \begin{array}{rcc} q & \Rightarrow & a \\ \neg(q) & \Rightarrow & b \end{array} \right. \\ \neg(p) & \Rightarrow & x \end{array} \right. \quad = \quad \left\{ \begin{array}{rcc} (p \wedge q) & \Rightarrow & a \\ (p \wedge \neg q) & \Rightarrow & b \\ \neg p & \Rightarrow & x \end{array} \right. \tag{383}$$

Lazy disjunction and conjunction

$$\begin{array}{rcl} \vee \circ \langle p, q \rangle & \leq & p \rightarrow \underline{\mathrm{TRUE}}, q \\ \wedge \circ \langle p, q \rangle & \leq & p \rightarrow q, \underline{\mathrm{FALSE}} \end{array} \tag{384}$$

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 a recursive function of scheme

$$f \stackrel{\mathrm{def}}{=} p \rightarrow g, h \circ f \circ e$$

(to be continued in Remark 56.)

REMARK 56: **A quick reference guide to program transformation (conclusion)**

Wherever f is a subexpression of a larger expression

$$\phi \circ f \circ a$$

the "**unfold** f"-transformation means to replace f's invocation by f's body, that is, $\phi \circ f \circ a$ expands to

$$\phi \circ (p \rightarrow g, h \circ f \circ e) \circ a$$

which will lead to

$$(p \rightarrow \phi \circ g, \phi \circ h \circ f \circ e) \circ a \tag{385}$$

by application of law (378).
Unfolding thus means to read

$$\phi(f\,x) = \phi(\left\{ \begin{array}{ccc} p\,x & \Rightarrow & g\,x \\ \neg(p\,x) & \Rightarrow & h(f(e\,x)) \end{array} \right.)$$

from left to right. The opposite transformation, which from right to left makes it possible to re-write (385) 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 33 Recall the factorial function,

$$y! = \left\{ \begin{array}{ccc} y = 0 & \Rightarrow & 1 \\ \neg(y = 0) & \Rightarrow & y \times (y-1)! \end{array} \right.$$

Frame it into the linear monadic schema (360) and calculate its iterative, `while`-loop version.
□

Exercise 34 Build f_φ (372) 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 35 Recalling the refinement of PPD into $RPPD$ (p.76), calculate $apply_R$ (367).
□

- 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.
- 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 — students often report 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 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 — results already available concerning **LTS**s (“labelled transition systems”).
- Integration of SETS with other refinement calculi (on-going Ph.D. thesis).
- Development of a genetic-algorithm-geared SETS animator (recent 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 the **IFAD Toolbox**.
- Further integration of SETS with Morgan’s and Bird-Moor’s calculi (based on comparative work).
- Development of a set of tools supporting SETS-based **reverse engineering** — a R&D consortium (KARMA) has been set up involving us and 3 software houses.
- 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? Try [McL95].

References

[ABNO97a] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. Bringing CAMILA and SETS together — the bams.cam and ppd.cam camila toolset demos. Technical report, DI/UM, Braga, December 1997. Technical Report, 45 p.

[ABNO97b] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Formal software engineering supported by functional programming. In A. De Giusti, J. Diaz, and P. Pesado, editors, *Proc. II Conf. Latino Americana de Programación Funcional (CLaPF97)*, pages 1343–1358, La Plata, Argentina, October 1997. Centenario UNLP.

[ABNO97c] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Prototyping and refinement of constructive specifications, 1997. In AMAST'97: Sixth International Conference on Algebraic Methodology and Software Technology, 13–17 December 1997, Macquarie University, Sydney, Australia.

[AH88] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. Technical Report CS 3/880, University of Sussex, 1988.

[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.

[Bar97] L. S. Barbosa. A note on SETS functorial foundations, March 1997. DI/UM Internal Research Note.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.

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

[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.

[FL97] J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques*. Cambridge University Press, 1^{st} edition, 1997.

[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.).

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in Lecture Notes in Computer Science. Springer, 1996.

[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.

[McC63] J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *Proc. of* IFIP *62*, pages 21–28, Amsterdam-London, 1963. North-Holland Pub. Company.

[McL95] C. McLarty. *Elementary Categories, Elementary Toposes*. Oxford Logic Guides n.º 21. Calendron Press, Oxford, 1^{st} edition, 1995.

[Mil89] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989. C. A. R. Hoare.

[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 May 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`.

[Par98] A. Pardo. Monadic corecursion — definition, fusion laws, and applications. *Electronic Notes in Theoretical Computer Science*, 11, 1998.

[Rod93] C. J. Rodrigues. Sobre o desenvolvimento formal de bases de dados. Master's thesis, University of Minho, 1993. (In Portuguese).

[Rod97] C. J. Rodrigues. A very general monotonicity result in SETS, July 1997. DI/UM Internal Research Note.

[Spi89] J. M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989. C. A. R. Hoare.

[TM95] A. Takano and E. Meijer. Shortcut to deforestation in calculational form. In *Proc. FPCA'95*, 1995.

[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).