# Data Transformation by Calculation

José N. Oliveira

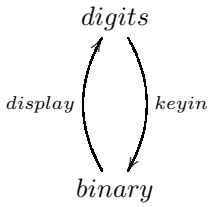
Dep. Informática, Universidade do Minho, 4700-320 Braga, Portugal,
jno@di.uminho.pt

**Abstract.** This tutorial addresses the foundations of data-model transformation. A catalog of transformations is presented which includes abstraction and representation relations and associated constraints, all expressed in an algebraic style via the pointfree-transform, a technique resembling the Laplace transform in mathematics: predicates are converted to binary relation terms (of the algebra of programming) in a 2-level style encompassing both data and operations. Data-calculation, which also includes transformation of recursive data models into "flat" database schemes, has been in use at Minho as alternative to standard database design and is the foundation of the "2LT bundle" of tools available from the UMinho Haskell libraries.

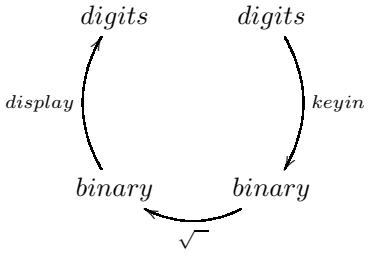
**Keywords:** Theoretical foundations ; refinement ; model driven engineering ; reusable theories.

## 1 Introduction

Watch yourself using a pocket calculator: once a digit key is pressed, the corresponding digit is displayed on the LCD display:




This simple situation illustrates the main ingredients of everyday interaction with machines: the abstract objects we have in mind (eg. digits, numbers, etc) need to be *represented* inside machines, before these can perform calculations for us (eg. square root):



To be useful, a calculator should behave properly: if the digit displayed is not always the one whose key was just pressed; or *nothing* at all is displayed; or even the required operation (such as triggered by square root key) is not properly calculated, the calculator is faulty and cannot be trusted.

When using machines such as computers or calculators, one is *subcontracting* mechanical services. Inside the machine, the same subcontracting process happens again and again: complex routines accomplish their tasks by subcontracting (simpler) routines, and so and so forth. So, the data representation process illustrated above for the (interaction with a) pocket calculator happens inside machines every time a routine is called: input data are to be made available in the appropriate format to the subcontracted routine, the result of which may need to change format again before it reaches its caller.

Such data $represent/retrieve$ processes (analogue to the $keyin/display$ process above) happen an uncountable number of times even in simple software systems. *Subcontracting* thus being the essence of computing (as it is of any organized society), much trouble is to be expected once one or more $represent/retrieve$ processes fail: the whole service as subcontracted from outside is likely to collapse.

Three kinds of fault have been identified above: loss of data, confusion among data and wrong calculation. The first two have to do with *data representation* and the third with *data calculation*. Preventing them from happening and disturbing the prodigious number of software subcontracts code designers are asked to implement is the main aim of this tutorial.

We will see that most of the work has to do with *data transformation*, a technique which the average programmer is not aware of being using when writing, in an 'ad hoc' way, cumbersome middleware code to "bridge the gap" between two different technology layers. The other part of the story — ensuring the overall correctness of software subcontracts — has to do with *data refinement*, a well established branch of the software sciences which is concerned with the relationship between (stepwise) specification and implementation of software.

This tutorial notes are organized as follows. Sections 2 and 3 present the overall spirit of our approach and introduce a simple case study which will be taken as running example. Section 4 is a review of the binary relation notation and calculus (referred to as the *pointfree (PF) transform*). Section 5 shows how to denote the meaning of data in terms of such a unified notation. Section 6 expresses data impedance mismatch in the PF-style. Sections 7 to 9 illustrate the approach in the context of (database) relational modeling. Recursive data modeling is addressed from section 10 onwards. Once impedance of recursive data is dealt with, we show how to handle cross-paradigm impedance by calculation (section 11) and how to transcribe operations from recursive to flat data models (section 12). Section 13 and appendix A are concerned with tools and practical issues. Finally, section 14 concludes and points out a number of research directions in the field.

## 2 On data representation

The theoretical foundation of *data representation* can be written in few words: what matters is the *no loss / no confusion* principle hinted above. Let us explain what this

means by writing $b\ R\ a$ to denote the fact that *datum* $b$ *represents datum* $a$, and the converse fact $a\ R^\circ\ b$ to denote that $a$ *is the datum represented by* $b$. The use of definite article "*the*" instead of "*a*" in the previous sentence is already a symptom of the **no confusion** principle: we want $b$ to represent *only one* datum of interest. So $R$ should be *injective*:

$$\langle \forall\ a, a', b\ ::\ b\ R\ a\ \wedge\ b\ R\ a' \Rightarrow a = a' \rangle \tag{1}$$

The **no loss** principle means that no data is lost in the representation process or, in other words, that every datum of interest $a$ is representable by some $b$:

$$\langle \forall\ a\ ::\ \langle \exists\ b\ ::\ b\ R\ a \rangle \rangle \tag{2}$$

In discrete maths this means that $R$ should be totally defined, or *entire* in our terminology [1].

It follows that the retrieval relation $R^\circ$ is *surjective* (as converses of entire relations always are) and *simple* (as converses of injective relations always are). By a simple relation we mean what is normally referred to as a *partial function* [2].

In general, it is useful to give some freedom to the retrieval relation, provided that it bears the desired properties (simplicity and surjectivenss) and that representation and retrieval are *connected* to each other. This last property is written as

$$\langle \forall\ a, b\ ::\ b\ R\ a \Rightarrow a\ F\ b \rangle \tag{3}$$

where $F$ denotes the chosen retrieval relation. This enforces retrieval relations always *larger* than the converses of their chosen representations,

$$R^\circ \subseteq F \tag{4}$$

thus establishing $R^\circ$ as the *least* retrieval relation associated with representation $R$. (In our starting scenario, $F$ was chosen to be *exactly* $R^\circ$.) Note that we could also have expressed implication (3) as

$$R \subseteq F^\circ \tag{5}$$

which is equivalent to (4), meaning that $F^\circ$ is the largest representation one can connect to a given retrieval relation $F$.

To express the fact that $(R, F)$ is a *connected* representation/retrieve pair we will draw a diagram of the form

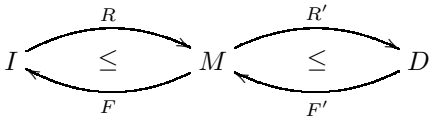
$$A \underset{F}{\overset{R}{\rightleftarrows}} C \qquad \leq \tag{6}$$

[1] Our choice of terminology is that of [10], a well-known textbook in programming using calculational techniques.

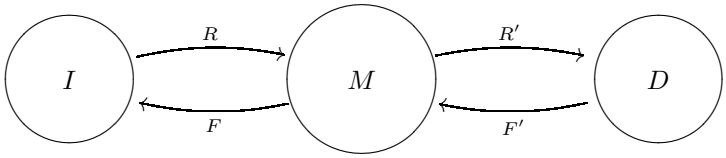
[2] The two rules of thumb "converse of *injective* is *simple*" and "converse of *entire* is *surjective*" will be studied in section 4.

where $A$ is the datatype of data *to be represented* and $C$ is the chosen datatype of representations. In the data refinement literature, $A$ is often referred to as *the abstract type* and $C$ as *the concrete one*, because $C$ contains more information than $A$, which is *ignored* by $F$ (a non-injective relation in general). This explains why $F$ is also referred to as the *abstraction relation* in a $(R, F)$ pair.

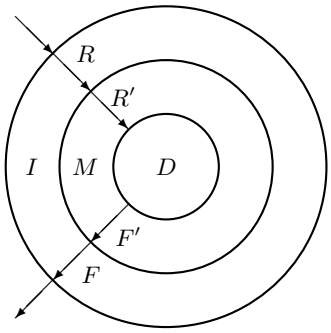
In general, it will make sense to connect more than one layer of abstraction as in, for instance,




where letters $I$, $M$ and $D$ have been judiciously chosen so as to suggest the words *interface*, *middleware* and *dataware*, respectively. In fact, data become "more concrete" as they go down the traditional layers of software architecture: the contents of interactive, handy objects at the interface level (often pictured as trees, combo boxes and the like) become pointer structures (eg. in C++/C#) as they descend to the middleware, from where they are channeled to the data level, where they live as persistent database records. A popular picture of the diagram above is

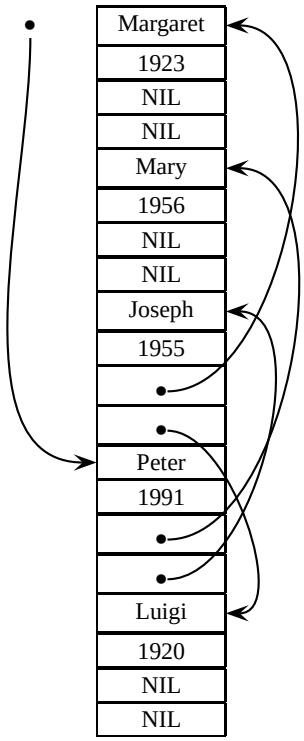



where $I, M$ and $D$ can be seen as communicating processes, represented by circles. The layered-architecture depicted in




(7)

is also popular. Despite the different geometry, the meaning of the two drawings is the same.




(a)

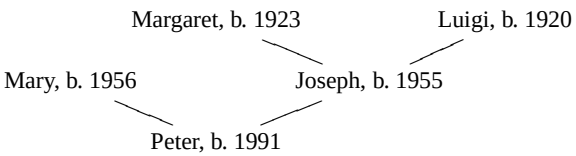
| ID | Name | Birth |
|---|---|---|
| 1 | Joseph | 1955 |
| 2 | Luigi | 1920 |
| 3 | Margaret | 1923 |
| 4 | Mary | 1956 |
| 5 | Peter | 1991 |

| ID | Ancestor | ID |
|---|---|---|
| 5 | Father | 1 |
| 5 | Mother | 4 |
| 1 | Father | 2 |
| 1 | Mother | 3 |

(b)

**Fig. 1.** Middleware (a) and dataware (b) formats for family tree sample data (8)

As an example, consider an interface ($I$) providing direct manipulation of pedigree trees, so common nowadays in genealogy websites:




(8)

Trees — which are the users' mental model of inductive structures — become pointer structures (Fig. 1a) once channeled to the middleware ($M$). For archival purposes, such structures are eventually buried into the dataware level ($D$) in the form of very concrete, persistent records (rows) of database files (tables), cf. eg. Fig. 1b.

## 3 Context and Motivation

Once materialized in some technology (eg. XML, C/C++/Java, SQL, etc), the layers of (7) stay apart of each other in different programming *paradigms* (eg. markup languages, object-orientation, relational databases, etc) each requiring its own skills and programming techniques.

As we have seen above, different data models can be compared via abstraction/ representation pairs. These are expected to be more complex once the two models under comparison belong to different paradigms. This kind of complexity is a measure of the *impedance mismatches between the various data-modeling and data-processing paradigms*, in the words of reference [35] where a thorough account is given of the many problems which hinder software technology in this respect. Quoting [35]:

> *Whatever programming paradigm for data processing we choose, data has the tendency to live on the other side or to eventually end up there. (...) This myriad of inter- and intra-paradigm data models calls for a good understanding of techniques for mappings between data models, actual data, and operations on data. (...)*
> *Given the fact that IT industry is fighting with various impedance mismatches and data-model evolution problems for decades, it seems to be safe to start a research career that specifically addresses these problems.*

Reference [35] goes further in identifying three main ingredients (levels) in *mapping scenarios*:

- the *type-level* mapping of a source data model to a target data model;
- two maps ("map forward" and "map backward") between source / target data;
- the *transcription level* mapping of source operations into target operations.

Clearly, inequation (6) can be seen as a succinct presentation of the two first ingredients, the former being captured be the $\leq$-ordering on data models and the latter by the $(F, R)$ pair of relations. The third can easily be captured by putting two instances of (6) together, in a way such that the input and output types of a given operation, say $O$, are *wrapped* by forward and backward data maps:

$$\begin{array}{ccc} A & \underset{F}{\overset{R}{\leq}} & C \\ {\scriptstyle O}\downarrow & & \downarrow{\scriptstyle P} \\ B & \underset{F'}{\overset{R'}{\leq}} & D \end{array} \tag{9}$$

The (safe) transcription of $O$ into $P$ can be formally stated by ensuring that the picture is a commutative diagram. A typical situation arises when $A$ and $C$ are the same, and $O$ is regarded as a state-transforming operation in a software component, eg. one of its CRUD ("Create, Read, Update and Delete") operations. Then the diagram will ensure correct refinement of such an operation across the change of state representation.

The theory behind diagrams such as (9) is known as *data refinement*, and it is among the most studied formalisms in software design theory, being available from several textbooks [3].

The fact that state-of-the-art software technologies don't enforce such formal design principles in general leads to the unsafe technology which we live on today, which is hindered by permanent cross-paradigm impedance mismatch, loose (untyped) data mappings, unsafe CRUD operation transcription, etc. Why is this so? Why isn't data refinement widespread? Perhaps because it is hard to understand and apply in practice. This is true: refinement is far too complex a discipline for most software practitioners, a fact which is mirrored on its prolific terminology — cf. *downward*, *upward* refinement [25], *forwards*, *backwards* refinement [25, 62, 41], *S,SP,SC*-refinement [19] and so on. Another defect of these theories is their reliance on *invent & verify (proof)* development strategies which are hard to master and get involved once facing "real-sized" problems. What can we do about this?

The approach we propose to follow in this tutorial is different from the standard in two respects: first, we adopt a *transformational* strategy as opposed to invention-followed-by-verification; second, we adopt a *calculational* approach throughout our data transformation steps. What do we mean by "calculational"?

Let us briefly review some background. The idea of using mathematics to reason about and transform programs is an old one and can be traced back to the times of McCarthy's work on the foundations of computer programming [39], of Floyd's work on program meaning [22] and of Paterson and Hewitt's *comparative schematology* [55]. A so-called *program transformation* school was already active in the mid 1970s, see for instance references [14, 17]. But program transformation becomes *calculational* only after the inspiring work of J. Backus in his *algebra of (functional) programs* [6] where the emphasis is put on the algebra of language combinators rather than on the $\lambda$-notation and its variables, or points. This is why this calculus is said to be *point-free*.

Intensive research on the (pointfree) program calculational approach in the last thirty years has led to the discipline of *algebra of programming* [10, 4]. The priority of this discipline has been, however, mostly on reasoning about *algorithms* rather than *data structures*. Our own attempts to set up a *calculus of data structures* date back to [44–46] where the $\leq$-ordering and associated calculus are defined. The approach, however, was not agile enough. It is only after its foundations are stated in the pointfree style [47, 48] that elegant calculations can be performed to calculate data representations. References [2, 15] describe tools which have been developed to (partly) automate this kind of data-level calculation.

The purpose of this tutorial is to provide the reader with strategies and tools for smoothing data impedance mismatch by calculation. Before we go into describing the rules of the calculus, we will present (in the section which follows) some background on the pointfree transform, while settling basic notation conventions. (Readers familiar with the binary relation calculus, as presented for instance in [10], will want to skip the section which follows; readers completely unfamiliar with these matters may want to skim through it in a first reading and to revisit it more thoroughly every time it's needed.)

[3] See eg. [31, 42, 18].

## 4 Introducing the Pointfree Transform

By *pointfree transform* [54] ("PF-transform" for short) we essentially mean the conversion of predicate logic formulæ into binary relations by removing bound variables and quantifiers — a technique which, initiated by De Morgan in the 1860s [56], eventually lead to what is known today as the *algebra of programming* [10, 4]. As explained in [54], the PF-transform is analogous to a reasoning strategy known as the *Laplace transform* applicable to integral / differential equations [34].

Theories "refactored" via the PF-transform become more general, more structured and simpler [52–54]. Elegant expressions replace lengthy formulæ and easy-to-follow calculations replace pointwise proofs with lots of "$\cdots$" notation, case analyses and natural language explanations for "obvious" steps.

The main principle of the PF-transform is that *"everything is a binary relation"* once logical expressions are PF-transformed; one thereafter resorts to the powerful calculus of binary relations [10, 4] until a solution for the problem is found, which is mapped back to logics if required.

*Relations.* Let $B \xleftarrow{R} A$ denote a binary relation on datatypes $A$ (source) and $B$ (target). We will say that $B \longleftarrow A$ is the *type* of $R$ and write $b\ R\ a$ to mean that pair $(b, a)$ is in $R$. Of course, relation type declarations $B \xleftarrow{R} A$ and $A \xrightarrow{R} B$ are equivalent.

The underlying partial order on relations is written $R \subseteq S$ (read: *"$R$ is at most $S$"*), meaning that $S$ is either more defined or less deterministic than $R$,

$$R \subseteq S \equiv \langle \forall\ b, a\ ::\ b\ R\ a \Rightarrow b\ S\ a \rangle \tag{10}$$

for $R, S$ of the same type.

$R \cup S$ denotes the union of two relations and $\top$ is the largest relation of its type. Its dual is $\bot$, the smallest such relation (the empty one). Equality on relations can be established by $\subseteq$-antisymmetry:

$$R = S \equiv R \subseteq S\ \wedge\ S \subseteq R \tag{11}$$

Relations can be combined by three basic operators: composition ($R \cdot S$), converse ($R^\circ$) and meet ($R \cap S$). $R^\circ$, the *converse* of $R$ is such that $a(R^\circ)b$ iff $bRa$ holds. Meet corresponds to set-theoretical intersection and composition is defined in the usual way: $b(R \cdot S)c$ holds wherever there exists some mediating $a$ such that $bRa\ \wedge\ aSc$. Thus we get another basic rule of the PF-transform:

$$b(R \cdot S)c \equiv \langle \exists\ a\ ::\ bRa\ \wedge\ aSc \rangle \tag{12}$$

Note how converse commutes with composition

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{13}$$

and with itself:

$$(R^\circ)^\circ = R \tag{14}$$

All these basic operators are $\subseteq$-monotonic. Composition is the basis of (sequential) factorization. Everywhere $T = R \cdot S$ holds, the replacement of $T$ by $R \cdot S$ will be referred to as a "factorization" and that of $R \cdot S$ by $T$ as "fusion". Every relation $B \xleftarrow{R} A$ allows for two trivial factorizations, $R = R \cdot id_A$ and $R = id_B \cdot R$ where, for every $X$, $id_X$ is the identity relation mapping every element of $X$ onto itself. (As a rule, subscripts will be dropped wherever types are implicit or easy to infer.)

*Coreflexives and orders.* Some standard terminology arises from the $id$ relation: a (endo) relation $A \xleftarrow{R} A$ (often called an *order*) will be referred to as *reflexive* iff $id \subseteq R$ holds and as *coreflexive* iff $R \subseteq id$ holds.

Coreflexive relations are fragments of the identity relation which model predicates or sets. The PF-transform of a (unary) *predicate* $p$ is the coreflexive $[\![p]\!]$ such that

$$b[\![p]\!]a \equiv (b = a) \ \wedge \ (p\ a) \tag{15}$$

that is, the relation that maps every $a$ which satisfies $p$ (and only such $a$) onto itself. The PF-meaning of a set $S$ is $[\![\lambda a.a \in S]\!]$, that is, $b[\![S]\!]a \equiv (b = a) \ \wedge \ a \in S$. (The $[\![\ ]\!]$ brackets will be omitted wherever clear from the context.)

Preorders are reflexive, transitive relations, where $R$ is transitive iff $R \cdot R \subseteq R$ holds. Partial orders are anti-symmetric preorders, where $R$ being anti-symmetric means $R \cap R^\circ \subseteq id$. A preorder $R$ is an *equivalence* if it is symmetric, that is, if $R = R^\circ$.

*Taxonomy.* Converse is of paramount importance in establishing a wider taxonomy of binary relations. Let us first define two important notions: the *kernel* of a relation $R$, $\mathsf{ker}\ R \stackrel{\text{def}}{=} R^\circ \cdot R$ and its dual, the *image* of $R$: $\mathsf{img}\ R \stackrel{\text{def}}{=} R \cdot R^\circ$. From (13, 14) one immediately draws

$$\mathsf{ker}\,(R^\circ) = \mathsf{img}\ R \tag{16}$$

$$\mathsf{img}\,(R^\circ) = \mathsf{ker}\ R \tag{17}$$

Kernel and image lead to some useful terminology:

| | *Reflexive* | *Coreflexive* |
|---|---|---|
| $\mathsf{ker}\ R$ | entire $R$ | injective $R$ |
| $\mathsf{img}\ R$ | surjective $R$ | simple $R$ |

(18)

In words: a relation $R$ is said to be *entire* (or total) iff its kernel is reflexive; and *simple* (or functional) iff its image is coreflexive. Dually, $R$ is *surjective* iff $R^\circ$ is entire, and $R$ is *injective* iff $R^\circ$ is simple.

Recall that these four classes have already been mentioned in section 2. So it may be useful to check formulæ (1,2) against the definitions captured by (18). We shall do it for (2) as warming-up exercise in pointfree-to-pointwise conversion:

$$\begin{array}{ll} & R \text{ is entire} \\ \equiv & \quad \{\ (18)\ \} \\ & \mathsf{ker}\ R \text{ is reflexive} \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ definitions } \} \\
& id \subseteq R^\circ \cdot R \\
\equiv & \{ \text{ going pointwise, for all } c, a \ \} \\
& c = a \ \Rightarrow \ \langle \exists\, b \ :: \ c\, R^\circ b \ \wedge \ b\, Ra \rangle \\
\equiv & \{ \text{ substitution } c := a \text{ ; converse } \} \\
& \langle \exists\, b \ :: \ b\, R\, a \ \wedge \ b\, Ra \rangle \\
\equiv & \{ \ p \ \wedge \ p \equiv p \ \} \\
& \langle \exists\, b \ :: \ b\, R\, a \rangle
\end{array}
$$

*Exercise 1.* Derive (1) from (18).
□

*Exercise 2.* Resort to (16,17,18) and (18) to prove the following four rules of thumb:

- converse of *injective* is *simple* (and vice-versa)
- converse of *entire* is *surjective* (and vice-versa)
- smaller than injective (simple) is injective (simple)
- larger than entire (surjective) is entire (surjective)

(Recall that two of these have been already used in section 2.)
□

A relation is said to be a *function* iff it is both simple and entire. Following a widespread convention, functions will be denoted by lowercase characters (eg. $f$, $g$, $\phi$) or identifiers starting with lowercase characters, and function application will be denoted by juxtaposition, eg. $f\ a$ instead of $f(a)$. Thus $bfa$ means the same as $b = f\ a$. Another means of denoting functions is the use of sections $(a\theta)$ and $(\theta b)$ of a binary operator $\theta$, whose meaning is as follows:

$$f = (a\theta) \ \equiv \ f\ b = a\ \theta\ b \tag{19}$$

$$g = (\theta b) \ \equiv \ g\ a = a\ \theta\ b \tag{20}$$

The overall taxonomy of binary relations is pictured in Fig. 2 where, further to the standard classification, we add *representations* and *abstractions*. These are the relation classes involved in $\leq$-rules (6), as we have seen already. Because of $\subseteq$-antisymmetry, $\mathsf{img}\ F = id$ wherever $F$ is an *abstraction* and $\mathsf{ker}\ R = id$ wherever $R$ is a *representation*.

Bijections (also referred to as isomorphisms) are functions, abstractions and representations at the same time. A particular isomorphism is $id$, which also is the smallest equivalence relation on a particular data domain. So, $b\ id\ a$ means the same as $b = a$.

*Functions and relations.* The interplay between functions and relations is a rich part of the binary relation calculus [10]. For instance, the PF-transform rule which follows, involving two functions $(f, g)$ and an arbitrary relation $R$

$$b(f^\circ \cdot R \cdot g)a \equiv (f\ b)R(g\ a) \tag{21}$$

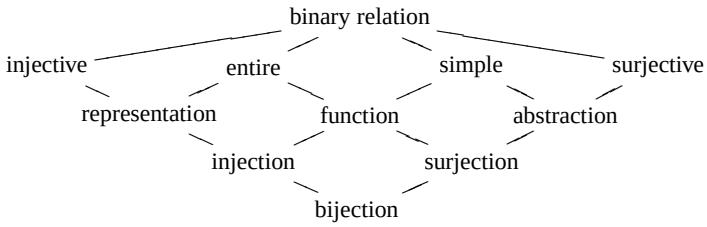



**Fig. 2.** Binary relation taxonomy

plays a prominent rôle in the PF-transform [3]. For instance, the pointwise definition of the kernel of a function $f$, $b(\ker f)a \equiv f\ b = f\ a$, stems from (21), whereby it is easy to see that $\top$ is the kernel of every constant function, $1 \xleftarrow{!} A$ included. (Function ! — read "!" as "bang" — is the unique function of its type, where 1 denotes the singleton type.)

Given two preorders $\leq$ and $\sqsubseteq$, one may relate arguments and results of pairs of functions $f$ and $g$ in, essentially, two ways:

$$f \cdot \sqsubseteq \ \subseteq \ \leq \cdot g \tag{22}$$

$$f^{\circ} \cdot \sqsubseteq \ = \ \leq \cdot g \tag{23}$$

As we shall see shortly, (22) is equivalent to $\sqsubseteq \ \subseteq \ f^{\circ} \cdot \leq \cdot g$. For $f = g$, this establishes $\sqsubseteq$ to $\leq$ monotonicity, thanks to (21). Both $f, g$ in pattern (23) are monotone and said to be *Galois connected*, $f$ (resp. $g$) being referred to as the *lower* (resp. *upper*) adjoint of the connection. By introducing variables in both sides of (23) via (21), we obtain (for all $a$ and $b$)

$$(f\ b) \sqsubseteq a \ \equiv \ b \leq (g\ a) \tag{24}$$

For further details on the rich theory of Galois connections (GCs) with examples and applications see [1, 3].

Galois connections in which the two preorders are relation inclusion ($\leq, \sqsubseteq \ := \ \subseteq, \subseteq$) are particularly interesting because the two adjoints are relational combinators and the connection itself is their universal property. The following table lists connections which are relevant for this tutorial, using section notation (19,20) to denote upper and lower adjoints:

| $(f\ R) \subseteq S \equiv R \subseteq (g\ S)$ | | | |
|---|---|---|---|
| **Description** | $f$ | $g$ | **Obs.** |
| Converse | $(\_)^\circ$ | $(\_)^\circ$ | |
| *Shunting* rule | $(h\cdot)$ | $(h^\circ\cdot)$ | NB: $h$ is a function |
| "Converse" *shunting* rule | $(\cdot h^\circ)$ | $(\cdot h)$ | NB: $h$ is a function |
| Left-division | $(R\cdot)$ | $(R \setminus\ )$ | read "$R$ under ..." |
| Right-division | $(\cdot R)$ | $(\ /\ R)$ | read "... over $R$" |
| range | $\rho$ | $(\cdot \top)$ | lower $\subseteq$ restricted to coreflexives |
| domain | $\delta$ | $(\top \cdot)$ | lower $\subseteq$ restricted to coreflexives |
| difference | $(\_ - R)$ | $(R \cup\ )$ | |

(25)

It is instructive to recover properties of the relation calculus from (25). For instance, the entry marked "*Shunting* rule" leads to

$$h \cdot R \subseteq S \equiv R \subseteq h^\circ \cdot S \tag{26}$$

for all $h, R$ and $S$. By taking converses, one gets another entry, namely

$$R \cdot h^\circ \subseteq S \equiv R \subseteq S \cdot h \tag{27}$$

These equivalences are popularly known as "shunting rules" [10]. The fact that *at most* and equality coincide in the case of functions

$$f \subseteq g \ \equiv\ f = g \ \equiv\ f \supseteq g \tag{28}$$

is among many beneficial consequences of these rules.

It should be mentioned that some rules in (25) appear in the literature under different guises and usually not identified as GGs [4]. For a thorough presentation of the relational calculus in terms of GCs see [4]. There are *many* advantages in such an approach. Further to the systematic tabulation of operators (of which the table above is just a sample), GCs have a rich algebra of properties. For instance,

- the two adjoints $f$ and $g$ in a GC are monotonic;
- lower adjoint $f$ commutes with join and upper-adjoint $g$ commutes with meet, wherever these exist;
- two cancellation laws hold, *left-cancellation*

$$b \leq g(f\ b) \tag{29}$$

and *right-cancellation*

$$f\ (g\ a) \sqsubseteq a \tag{30}$$

It may happen that a cancellation law holds up to equality, eg.

$$f\ (g\ a) = a \tag{31}$$

In this case the connection is said to be *perfect* on the particular side [1].

*Exercise 3.* Prove that relational composition preserves *all* relational classes in the taxonomy of Fig. 2.
□

[4] For instance, the *shunting* rule is called *cancellation law* in [62].

*Simplicity.* Simple relations (vulg. partial functions) will be particularly relevant in the sequel because of their ubiquity in software modeling. In particular, they will be used to model data *identity* and any kind of data structure "embodying a functional dependency" [52] such as eg. relational database tables, memory segments (both static and dynamic) and so on.

In the same way simple relations generalize functions (Fig. 2), *shunting* rules (26, 27) generalize to

$$S \cdot R \subseteq T \equiv (\delta\, S) \cdot R \subseteq S^\circ \cdot T \tag{32}$$

$$R \cdot S^\circ \subseteq T \equiv R \cdot \delta\, S \subseteq T \cdot S \tag{33}$$

for $S$ simple. These rules involve the *domain* operator ($\delta$) whose GC

$$\delta\, R \subseteq \Phi \equiv R \subseteq \top \cdot \Phi \tag{34}$$

is also listed in table (25), for $\Phi$ coreflexive.

We will draw $B \xleftarrow{R} A$ (or $A \xrightarrow{R} B$) to indicate that $R$ is simple. Later on we will need to describe finite simple relations at pointwise level. The notation we shall adopt for this purpose is borrowed from VDM [31], where it is known as *mapping comprehension*. This notation exploits the applicative nature of a simple relation $S$ by writing $b\ S\ a$ as $a \in dom\ S\ \wedge\ b = S\ a$, where $\wedge$ should be understood non-strict on the right argument [5] and $dom\ S$ is the set-theoretic version of coreflexive $\delta\ S$ above, that is,

$$\delta\ S = [\![dom\ S]\!] \tag{35}$$

holds (cf. the isomorphism between sets and coreflexives). Then relation $S$ itself is written

$$\{a \mapsto S\ a \mid a \in dom\ S\}$$

and projection $f \cdot S \cdot g^\circ$ as

$$\{g\ a \mapsto f(S\ a) \mid a \in dom\ S\} \tag{36}$$

provided $g$ is injective (thus ensuring simplicity).

*Exercise 4.* Show that the union of two simple relations $M$ and $N$ is simple *iff* the following condition holds:

$$M \cdot N^\circ \subseteq id \tag{37}$$

Suggestion: resort to universal property

$$(R \cup S) \subseteq X \equiv (R \subseteq X) \wedge (S \subseteq X) \tag{38}$$

Furthermore show that (37) converts to pointwise notation as follows,

$$\langle \forall\ a\ ::\ a \in (dom\ M \cap dom\ N) \Rightarrow (M\ a) = (N\ a) \rangle$$

— a condition known as (map) *compatibility* in VDM terminology [21].
□

[5] VDM embodies a logic of partial functions (LPF) which takes this into account [31].

*Coreflexives again.* The *domain* of a relation is an example of coreflexive relation. It will be useful to order relations with respect to how defined they are:

$$R \preceq S \equiv \delta\, R \subseteq \delta\, S \tag{39}$$

From $\top = \ker !$ one draws another version of (39), $R \preceq S \;\equiv\; ! \cdot R \subseteq ! \cdot S$, which in turn leads to:

$$R \cup S \preceq T \equiv R \preceq T \;\wedge\; S \preceq T \tag{40}$$

$$R \cdot f^{\circ} \preceq S \equiv R \preceq S \cdot f \tag{41}$$

Among many other useful properties of coreflexives we single out the following, which prove very useful in calculations:

$$\Phi \cdot \Psi \;=\; \Phi \cap \Psi \;=\; \Psi \cdot \Phi \tag{42}$$

$$\Phi^{\circ} = \Phi \tag{43}$$

*Summary.* The material in this section is drawn from similar sections in [53, 54], which introduce the reader to the essentials of the PF-transform. While the notation adopted is that of a reference textbook on the subject [10], the presentation of the associated calculus is shorter and enhanced via the use of Galois connections, a strategy inspired by two (still unpublished) textbooks [1, 4]. There is a slight difference, perhaps: by regarding the underlying mathematics as that of a *transform* to be actually used by software engineers in model reasoning, the overall flavour is quite practical and not that of a *fine art* only accessible to the initiated — an aspect of the recent evolution of the calculus already stressed in [33].

The table below provides a summary of the PT-transform rules given so far, where left-hand sides are logical formulæ ($\psi$) and right-hand sides are the corresponding PF equivalents ($[\![\psi]\!]$).

| $\psi$ | $[\![\psi]\!]$ |
|---|---|
| $\langle \forall\, a, b \;::\; b\, R\, a \Rightarrow b\, S\, a \rangle$ | $R \subseteq S$ |
| $\langle \forall\, a \;::\; f\, a = g\, a \rangle$ | $f \subseteq g$ |
| $\langle \forall\, a \;::\; a\, R\, a \rangle$ | $id \subseteq R$ |
| $\langle \exists\, a \;::\; b\, R\, a \;\wedge\; a\, S\, c \rangle$ | $b(R \cdot S)c$ |
| $b\, R\, a \;\wedge\; b\, S\, a$ | $b\,(R \cap S)\,a$ |
| $b\, R\, a \vee b\, S\, a$ | $b\,(R \cup S)\,a$ |
| $(f\, b)\; R\; (g\, a)$ | $b(f^{\circ} \cdot R \cdot g)a$ |
| TRUE | $b \top a$ |
| FALSE | $b \bot a$ |

(44)

*Operator precedence.* In order to save parentheses in relational expressions, we define the following precedence ordering on the relational operators seen so far:

$$\_^{\circ} > \{\delta\,, \rho\} > (\cdot) > \cap > \cup$$

Example: $R \cdot \delta\, S^{\circ} \cap T \cup V$ abbreviates $((R \cdot (\delta\,(S^{\circ}))) \cap T) \cup V$.

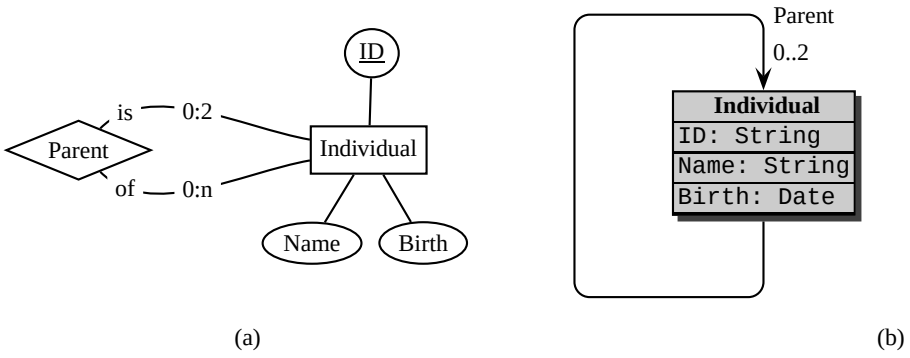



**Fig. 3.** ER and UML diagrams proposed for *genealogies*

## 5 Data structures

One of the main difficulties in studying data structuring is the number of disparate (graphic) notations, programming languages and paradigms one has to deal with. Which should one adopt? While graphical notations (such as the UML [13]) are gaining adepts everyday, it is difficult to be accurate in such notations because their semantics are, as a rule, not formally defined.

Our approach will be rather minimalist: we will *map* such notations to the PF-notation whose rudiments have just been presented. By the word "map" we mean a light-weight approach in this tutorial: presenting a fully formal semantics for the data structuring facilities offered by any commercial language or notation would be a tutorial in itself!

Let us illustrate our approach by resorting to the family tree example given earlier on — recall (8) and Fig. 1. Suppose requirements are to provide CRUD operations on a genealogy database collecting such family trees. How does one go about describing the data model underlying such operations?

The average database designer will approach the model via *entity-relationship* (ER) diagrams. Using the graphical notation proposed by [7], for instance, one could draw the diagram of figure 3(a). But many others will regard this notation too old-fashioned and will propose the UML of figure 3(b) instead.

Uncertain of what such drawings *actually mean*, many a programmer will prefer to go straight into code, eg. C

```
typedef struct Gen {
        char *name          /* name is a string */
        int   birth         /* birth year is a number */
        struct Gen *mother; /* genealogy of mother (if known) */
        struct Gen *father; /* genealogy of father (if known) */
        } ;
```

— which matches with Fig. 1a — or XML, eg.

```
<!-- DTD for genealogical trees -->
<!ELEMENT tree (node+)>
<!ELEMENT node (name, birth, mother?, father?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT birth (#PCDATA)>
<!ELEMENT mother EMPTY>
<!ELEMENT father EMPTY>
<!ATTLIST tree
  ident ID #REQUIRED>
<!ATTLIST mother
  refid IDREF #REQUIRED>
<!ATTLIST father
  refid IDREF #REQUIRED>
```

— or SQL, eg. (fixing some arbitrary sizes for datatypes)

```
CREATE TABLE INDIVIDUAL (
  ID  NUMBER  (10) NOT NULL,
  Name   VARCHAR (80) NOT NULL,
  Birth  NUMBER  (8)  NOT NULL,
  CONSTRAINT INDIVIDUAL_pk PRIMARY KEY(ID)
);

CREATE TABLE ANCESTORS (
  ID       VARCHAR (8)  NOT NULL,
  Ancestor VARCHAR (8)  NOT NULL,
  PID      NUMBER  (10) NOT NULL,
  CONSTRAINT ANCESTORS_pk PRIMARY KEY (ID,Ancestor)
);
```

— which matches with Fig. 1b.

What about functional programmers? By looking at the pedigree tree where we started from (8), an inductive data type inhabited by such trees can be defined, eg. in Haskell,

```
data PTree = Node {
        name   :: [ Char ],
        birth  :: Int     ,
        mother :: Maybe PTree,
        father :: Maybe PTree
        }
```
(45)

whereby (8) would be encoded as data value

```
Node
  {name = "Peter", birth = 1991,
   mother = Just (Node
     {name = "Mary", birth = 1956,
      mother = Nothing,
      father = Nothing}),
   father = Just (Node
```

```
{name = "Joseph", birth = 1955,
 mother = Just (Node
    {name = "Margaret", birth = 1923,
     mother = Nothing, father = Nothing}),
     father = Just (Node
        {name = "Luigi", birth = 1920,
         mother = Nothing, father = Nothing})})}
```

Of course, the same tree can be encoded in XML notation using eg. the DTD which follows

```
<!-- DTD for genealogical trees -->
<!ELEMENT tree (name, birth, tree?, tree?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT birth (#PCDATA)>
```

As well-founded structures, these trees can be pretty-printed as in (8). However, how can one ensure that the same *print-family-tree* operation won't loop forever while retrieving data from eg. Fig. 1b? This would clearly happen if, by mistake, record $\boxed{1 \mid Father \mid 2}$ in Fig. 1b were updated to $\boxed{1 \mid Father \mid 5}$ : $Peter$ would become a descendant of himself!

Several questions are on demand: are all the above data models "equivalent"? If so, in what sense? If not, how can they be ranked in terms of "quality"? How can we tell apart the *essence* of a data model from its technology wrapping?

To answer these questions we need to put some effort in describing the notations involved in terms of a single, simple (ie. technology free) unifying notation. But notation alone (syntax) is not enough: the ability to *reason* in such a notation is essential, otherwise different data models won't be comparable.

These requirements explain why, in what follows, we choose the PF-notation for such a purpose.

*Records* are *inhabitants of products.* Broadly speaking, a database is that part of an *information system* which collects *facts* or *records* of particular situations, which are under permanent retrieval and analytical processing. There may be *millions* of such records archived in a database and so efficiency is an issue. But, what *is* a record?

Any row in the tables of Fig. 1b is a record, ie. *records* a *fact*. For instance, record $\boxed{5 \mid Peter \mid 1991}$ means *Peter, whose ID number is 5, was born in 1991*. However, a mathematician would have written $(5, Peter, 1991)$ instead of *drawing* the tabular stuff; clearly, she/he would have inferred

$$\frac{\begin{array}{c} 5 \in I\!N, \\ Peter \in String, \\ 1991 \in I\!N \end{array}}{(5, Peter, 1991) \in N \times String \times I\!N}$$

So records can be regarded as *tuples* which inhabit *products* of types. Given two types $A$ and $B$, their (Cartesian) product is defined by $A \times B \stackrel{\text{def}}{=} \{(a, b) \mid a \in A \wedge b \in B\}$.

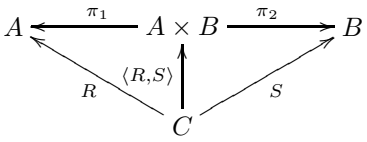
Product datatype $A \times B$ is essential to information processing and is available in virtually every programming language. In Haskell one writes `(A,B)` to denote $A \times B$, for $A$ and $B$ two predefined datatypes. This syntax can be decorated with names, eg.

```
data C = C { first :: A,  second :: B }
```

as was the case of `PTree` introduced earlier on. In the C programming language, the $A \times B$ datatype is realized using "struct"'s, eg.

```
struct {
   A first;
   B second;
};
```

The meaning of a product can be captured by the following diagram, where $R$ and $S$ are relations




(46)

and where the two projections $\pi_1, \pi_2$ are such that

$$\pi_1(a,b) = a \ \wedge \ \pi_2(a,b) = b \tag{47}$$

Given $R, S$, $\langle R, S\rangle$ — read: *"split of $R$ and $S$"* — is a binary relation relating three objects $a, b, c$:

$$(a,b)\langle R,S\rangle c \ \equiv \ a \ R \ c \ \wedge \ b \ S \ c \tag{48}$$

A special case of *split* will be referred to as *relational product*:

$$R \times S \stackrel{\text{def}}{=} \langle R \cdot \pi_1, S \cdot \pi_2\rangle \tag{49}$$

So we can add two more entries to table (44):

| $\psi$ | $[\![\psi]\!]$ |
|---|---|
| $a \ R \ c \ \wedge \ b \ S \ c$ | $(a,b)\langle R,S\rangle c$ |
| $b \ R \ a \ \wedge \ d \ S \ c$ | $(b,d)(R \times S)(a,c)$ |

(50)

As simple example of PF-transformation, we calculate the PF-version of (48):

$$\begin{array}{ll} & (48) \\ \equiv & \quad \{ \text{ projections (47) } \} \\ & (a,b)\langle R,S\rangle c \equiv \pi_1(a,b) \ R \ c \ \wedge \ \pi_2(a,b) \ S \ c \\ \equiv & \quad \{ \text{ (21) } \} \\ & (a,b)\langle R,S\rangle c \equiv (a,b)(\pi_1^\circ \cdot R)c \ \wedge \ (a,b)(\pi_2^\circ \cdot S)c \\ \equiv & \quad \{ \text{ introduce } \cap \text{ (44) ; remove variables } \} \\ & \langle R,S\rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \end{array} \tag{51}$$

Note that binary product can be generalized to *n-ary product* $A_1 \times A_2 \times \ldots \times A_n$ involving projections $\{\pi_i\}_{i=1,n}$ such that $\pi_i(a_1, \ldots, a_n) = a_i$.

*Exercise 5.* Identify which types are isomorphic under the following bijections:

$$flatr(a,(b,c)) = (a,b,c) \tag{52}$$

$$flatl((b,c),d) = (b,c,d) \tag{53}$$

□

*Exercise 6.* Show that the side condition of the following *split-fusion* law [6]

$$\langle R, S\rangle \cdot T = \langle R \cdot T, S \cdot T\rangle \;\Leftarrow\; R \cdot (\mathsf{img}\, T) \subseteq R \vee S \cdot (\mathsf{img}\, T) \subseteq S \tag{54}$$

can be dispensed with (at least) in the following situations: (a) $T$ is simple; (b) $R$ or $S$ are functions.
□

*Exercise 7.* Write the following cancellation law with less symbols assuming that $R \preceq S$ and $S \preceq R$ hold:

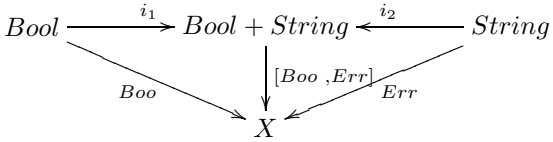
$$\pi_1 \cdot \langle R, S\rangle = R \cdot \delta\, S \;\wedge\; \pi_2 \cdot \langle R, S\rangle = S \cdot \delta\, R \tag{55}$$

□

*Summing up types.* The following is a declaration of a type in Haskell which is inhabited by *either* Booleans or error strings:

```
data X = Boo Bool | Err String
```

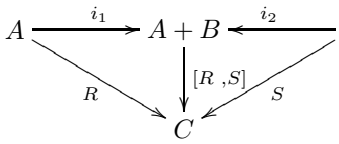
If one queries a Haskell interpreter for the types of the `Boo` and `Err` constructors, one gets two functions which fit in the following diagram




(56)

where $Bool + String$ denotes the sum (disjoint union) of types $Bool$ and $String$, functions $i_1, i_2$ are the necessary *injections* and $[Boo\ , Err]$ is an instance of the *"either"* relational combinator :

$$[R\ , S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \text{ cf. diagram } \tag{57}$$




In pointwise notation, $[R\ , S]$ means

$$c[R\ , S]x \;\equiv\; \langle \exists\, a \;::\; c\, R\, a \;\wedge\; x = i_1 a\rangle \vee \langle \exists\, b \;::\; c\, S\, a \;\wedge\; x = i_2 b\rangle$$

[6] Theorem 12.30 in [1].

In the same way *split* was used above to define relational product $R \times S$, *either* can be used to define *relational sums*:

$$R + S = [i_1 \cdot R \, , i_2 \cdot S] \tag{58}$$

As happens with products, $A + B$ can be generalized to $n$*-ary sum* $A_1 + A_2 + \ldots + A_n$ involving $n$ injections $\{i_i\}_{i=1,n}$.

In most programming languages, sums are not primitive and need to be programed on purpose, eg. in C (using unions)

```
struct {
   int tag; /* eg. 1,2 */
   union {
      A ifA;
      B ifB;
   } data;
};
```

where explicit integer tags are made available to model injections $i_1, i_2$.

*(Abstract) pointers.* A particular example of a datatype sum is $1 + A$, where $A$ is an arbitrary type and 1 is the singleton type. The "amount of information" in this kind of structure is that of a pointer in C/C++: one "pulls a rope" and either gets nothing (1) or something useful of type $A$. In such a programming context "nothing" above means a predefined value NIL. This analogy supports our preference in the sequel for NIL as canonical inhabitant of datatype 1. In fact, we will refer to $1 + A$ (or $A + 1$) as the "pointer to $A$" datatype [7]. This corresponds to the `Maybe` type constructor in Haskell.

*Polynomial types, grammars and languages.* Types involving arbitrary nesting of products and sums are called *polynomial* types, eg. $1 + A \times B$ (the "pointer to struct" type). These types are useful in describing the abstract contents of generative grammars (expressed in BNF notation) once non-terminal symbols are identified with types and terminal symbols are filtered. The conversion is synthetized by the following table,

| BNF NOTATION | | POLYNOMIAL NOTATION |
|---|---|---|
| $\alpha \mid \beta$ | $\mapsto$ | $\alpha + \beta$ |
| $\alpha\beta$ | $\mapsto$ | $\alpha \times \beta$ |
| $\epsilon$ | $\mapsto$ | $1$ |
| $a$ | $\mapsto$ | $1$ |

(59)

applicable to the right hand side of BNF-productions, where $\alpha, \beta$ range over sequences of terminal or non-terminal symbols, $\epsilon$ stands for *empty* and $a$ ranges over terminal symbols. For instance, production

$$X \to \epsilon \mid a \; A \; X$$

[7] Note that we are abstracting from the reference/dereference semantics of a *pointer* as understood in C-like programming languages. This is why we refer to $1 + A$ as an *abstract* pointer. The explicit introduction of references (pointers, keys, identities) is deferred to section 10.

(where $X, A$ are non-terminals and $a$ is terminal) leads to equation

$$X = 1 + A \times X \tag{60}$$

which has $A^\star$ — the "sequence of $A$" datatype — as (least) solution. Since $1 + A \times X$ can also be regarded as instance of the "pointer to struct" pattern, once can encode the same equation as the following (suitably sugared) type declaration in C:

```
typedef struct x {
  A data;
  struct x *next;
} Node;

typedef Node *X;
```

*Recursive types.* Both the interpretation of grammars [60] and the analysis of datatypes with pointers lead to systems of polynomial equations, that is, to mutually recursive datatypes [61]. For instance, the two *typedef*s above lead to

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

It is the substitution of $Node$ by $A \times X$ in the second equation which gives raise to (60). There is a slight detail, though: in dealing with recursive types one needs to replace *equality* of types by *isomorphism* of types, a concept to to be dealt with later on (see section 6). So, for instance, the *PTree* datatype illustrated above in the XML and Haskell syntaxes is captured by the equation

$$PTree \cong Ind \times (PTree + 1) \times (PTree + 1) \tag{61}$$

where $Ind = Name \times Birth$ packages the information relative to the name and birth year datatypes, which don't participate in the recursive machinery and are, in a sense, *parameters* of the model. Thus one may write

$$PTree \cong \mathsf{G}(Ind, PTree) \tag{62}$$

where $\mathsf{G}$ captures the particular pattern of recursion chosen to model family trees

$$\mathsf{G}(X, Y) \stackrel{\text{def}}{=} X \times (Y + 1) \times (Y + 1)$$

($X$ refers to the parametric information and $Y$ to the inductive part.)

All the examples given so far illustrate the rôle of abstract data patterns in bridging over two different programming paradigms and notations. This will be particularly apparent from what follows.

*Membership.* The idea of typing data structures is that of controlling how data are stored in a computer so as to be properly fetched. Every instance of a datatype is thus a *data container* of some type and the typing mechanism should help in retrieving data from containers using some form of *membership* relation.

Sets are perhaps the best known data containers and purport a very intuitive notion of membership: everybody knows what $a \in S$ means, wherever $a$ is of type $A$ and $S$ of type $\mathcal{P}A$ (read: "the powerset of $A$"). Sentence $a \in S$ already tells us that (set) membership has type $A \xleftarrow{\in} \mathcal{P}A$. Now, the "linked list" model presented above is also a *container type*, the intuition being that $a$ belongs (or occurs) in list $l \in A^\star$ iff it can be found in the set of all its elements, $a \in elems\, l$, where $elems$ is the obvious "list setify" function. In the case of lists, membership has type $A \xleftarrow{\in} A^\star$ (note the overloading of symbol $\in$). But even product $A \times A$ has membership: $a$ is a member of a pair $(x, y)$ of type $A \times A$ iff it can be found in either sides of that pair, that is $a \in (x, y)$ means $a = x \vee a = y$. So it makes sense to define a *generic* notion of membership, able to fully explain the overloading of symbol $\in$ above.

Datatype membership has been extensively studied [26, 10, 53]. Below we deal with polynomial type membership, which is what it required in this tutorial. A polynomial type expression may involve the composition, product, or sum of other polynomial types, plus the identity ($\mathsf{Id}\ X = X$) and constant types ($\mathsf{F}\ X = K$), where $K$ is any basic datatype, eg. the Booleans, the natural numbers, etc. Generic membership is defined, in the PF-style, over the structure of polynomial types as follows:

$$\in_{\mathsf{K}} \stackrel{\text{def}}{=} \bot \tag{63}$$

$$\in_{\mathsf{Id}} \stackrel{\text{def}}{=} id \tag{64}$$

$$\in_{\mathsf{F}\times\mathsf{G}} \stackrel{\text{def}}{=} (\in_{\mathsf{F}} \cdot \pi_1) \cup (\in_{\mathsf{G}} \cdot \pi_2) \tag{65}$$

$$\in_{\mathsf{F}+\mathsf{G}} \stackrel{\text{def}}{=} [\in_{\mathsf{F}}, \in_{\mathsf{G}}] \tag{66}$$

$$\in_{\mathsf{F}\cdot\mathsf{G}} \stackrel{\text{def}}{=} \in_{\mathsf{G}} \cdot \in_{\mathsf{F}} \tag{67}$$

*Exercise 8.* Calculate the membership of type $\mathsf{F}\ X = X \times X$ and convert it to pointwise notation, so as to confirm the intuition above that $a \in (x, y)$ holds iff $a = x \vee a = y$.
□

Generic membership will be of help in specifying data structures which depend on each other by some form of *referential integrity* constraint. Before showing this, we need to introduce the important notion of *reference*, or *identity*.

*Identity.* Base clause (64) above clearly indicates that, sooner or later, equality plays its rôle when checking for polynomial membership. And equality of complex objects may be cumbersome to express and expensive to calculate. Moreover, checking two objects for equality based on their properties alone may not work: it may happen that two physically different objects have the same properties, eg. two employees with exactly the same age, name, born in the same place, etc.

This *identification* problem has a standard solution: one associates to the objects in a particular collection *identifiers* which are unique in that particular context, cf. eg. identifier *ID* in Fig. 1b. So, instead of storing a collection of objects of, say, type $A$ in a set of type $\mathcal{P}A$, one stores an association of unique names to the original objects, usually thought of in tabular format — as is the case in Fig. 1b.

However, thinking in terms of *tabular relations* expressed by sets of tuples where particular attributes ensure unique identification (vulg. *keys*), as is typical of database

theory [38], is not sufficiently general and actually quite cumbersome for reasoning purposes. References [50, 52] show that relational *simplicity* [8] is what matters in unique identification. So it suffices to regard collections of uniquely identified objects $A$ as simple relations of type

$$K \rightharpoonup A \tag{68}$$

where $K$ is a nonempty datatype of *keys*, or identifiers. For the moment, no special requirements are put on $K$. Later on, $K$ will be asked to provide for a countably infinite supply of identifiers, that is, to behave such as *natural number objects* do in category theory [40].

Below we show that simplicity and membership are what is required of our PF-notation to capture the semantics of data modeling (graphical) notations such as *Entity-Relationship* diagrams and UML class diagrams.

*Entity-relationship diagrams.* As the name tells, Entity-Relationship data modeling involves two basic concepts: *entities* and *relationships*. Entities correspond to *nouns* in natural language descriptions: they describe classes of objects which have identity and exhibit a number of properties or attributes. Relationships can be thought of as *verbs*: they record (the outcome of) actions which engage different entities.

A few notation variants and graphical conventions exist for these diagrams. For its flexibility, we stick to the *generic entity-relationship* (GER) proposal of [24]. Figure 4 depicts a GER diagram involving two entities: **Book** and **Borrower**. The latter possesses attributes *Name*, *Address*, *Phone* and identity *PID*. As anticipated above where discussing how to model object identity, the semantic model of **Borrower** is a simple relation of type

$$T_{PID} \rightharpoonup T_{Name} \times T_{Address} \times T_{Phone}$$

where by $T_a$ we mean the type where attribute $a$ takes values from. For notation economy, we will drop the $T_{..}$ notation and refer to the type of attribute $a$ by mentioning $a$ alone:

$$Borrowers = PID \rightharpoonup Name \times Address \times Phone$$

Entity **Book** has a multivalued attribute (*Author*) imposing at most 5 authors. The semantics of such attributes can be also captured by (nested) simple relations:

$$Books = ISBN \rightharpoonup Title \times (5 \rightharpoonup Author) \times Publisher \tag{69}$$

Note the use of number $5$ to denote the initial segment of the natural numbers ($\mathbb{N}$) up to 5, that is, set $\{1, 2, ..., 5\}$.

Books can be reserved by borrowers and there is no limit to the number of books the latter can reserve. The outcome of a reservation at a particular date is captured by relationship *Reserved*. Relationship formal semantics are also simple relations, this time involving the identities of the entities engaged. In this case:

$$Reserved = ISBN \times PID \rightharpoonup Date$$

[8] Recall that a relation is simple wherever its image is coreflexive (18).

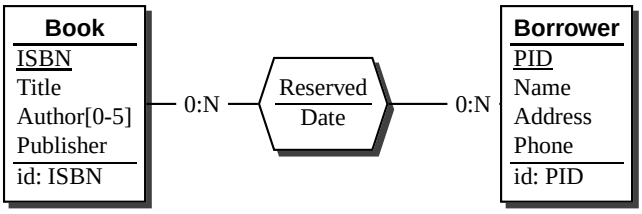



**Fig. 4.** Sample of GER diagram (adapted from [24]).

Altogether, the diagram specifies the following datatype inhabited by triples of simple relations:

$$Db = Books \times Borrowers \times Reserved$$

In summary, Entity-Relationship diagrams describe data models which are concisely captured by simple binary relations. But we are not done yet: the semantics of the problem include the fact that only *existing* books can be borrowed by *known* borrowers. So one needs to impose a semantic constraint (invariant) on datatype $Db$ which, written pointwise, goes as follows

$$\begin{array}{l} \phi(M, N, R) \stackrel{\text{def}}{=} \\ \quad \langle \forall\, i, p, d \;::\; d\, R\, (i, p) \Rightarrow \langle \exists\, x \;::\; x\, M\, i \rangle \;\wedge\; \langle \exists\, y \;::\; y\, M\, p \rangle \rangle \end{array} \tag{70}$$

where $i, p, d$ range over $ISBN, PID$ and $Date$, respectively.

Constraints of this kind, which are implicitly assumed when interpreting *relationships* in these diagrams, are known as *integrity constraints*. Being invariants at the semantic level, they bring along with them the problem of ensuring their preservation by the corresponding CRUD operations. Worse than that, their definition in the predicate calculus is not agile enough for calculation purposes. Is there any alternative?

Space constraints preclude presenting the calculation which would show (70) *equivalent* to the following, much more concise PF-definition:

$$\phi(M, N, R) \stackrel{\text{def}}{=} R \cdot \in^{\circ} \preceq M \;\wedge\; R \cdot \in^{\circ} \preceq N \tag{71}$$

cf. diagram

$$\begin{array}{ccccc} ISBN & \xleftarrow{\in = \pi_1} & ISBN \times PID & \xrightarrow{\in = \pi_2} & PID \\ \downarrow{\scriptstyle M} & & \downarrow{\scriptstyle R} & & \downarrow{\scriptstyle N} \\ Title \times (5 \rightharpoonup Author) \times Publisher & & Date & & Name \times Address \times Phone \end{array}$$

To understand (71) and the diagram, the reader must recall the definition of the $\preceq$ ordering (39) — which compares the domains of two relations — and inspect the

types of the two memberships, $ISBN \xleftarrow{\in=\pi_1} ISBN \times PID$ in the first instance and $PID \xleftarrow{\in=\pi_2} ISBN \times PID$ in the second. We check the first instance, the second being similar:

$$
\begin{array}{ll}
& ISBN \xleftarrow{\in} ISBN \times PID \\
= & \quad \{ \text{ polynomial decomposition, membership of product (65) } \} \\
& (\in_{\mathsf{Id}} \cdot \pi_1) \cup (\in_{\mathsf{PID}} \cdot \pi_2) \\
= & \quad \{ \text{ (63) and (64) } \} \\
& id \cdot \pi_1 \cup \bot \cdot \pi_2 \\
= & \quad \{ \text{ trivia } \} \\
& \pi_1
\end{array}
$$

Multiplicity labels 0:N in the diagram of Fig. 4 indicate that there is no limit to the number of books a borrower can reserve. Now suppose the library decrees the following rule: *borrowers can have at most one reservation active*. In this case, label 0:N on the Books side must be restricted to 0:1. These so-called many-to-one relationships are once again captured by simple relations, this time of a different shape:

$$Reserved = PID \rightharpoonup ISBN \times Date \tag{72}$$

Note how clever use of simple relations dispenses with explicit cardinality invariants, which would put spurious weight on the data model. However, referential integrity is still to be maintained. The required pattern is once again nicely built up around membership,

$$\phi(M, N, R) \stackrel{\text{def}}{=} (\in \cdot R)^\circ \preceq M \ \wedge \ R \preceq N \tag{73}$$

cf.

$$
\begin{array}{ccc}
 & ISBN \times Date \xleftarrow{R} PID & \\
 & \Big\downarrow{\scriptstyle \in=\pi_1} \qquad \Big\downarrow{\scriptstyle \in=id} & \\
Title \times (5 \rightharpoonup Author) \times Publisher & \xleftarrow{M} ISBN \qquad PID \xrightarrow{N} & Name \times Address \times Phone
\end{array}
$$

Note the similarity in shape between these diagrams and the corresponding Entity-Relationship diagrams. The main advantage of the former resides in the richer semantics which enables formal reasoning, as we shall see in the sequel.

*UML class diagrams.* UML class diagrams and GERs are very similar. Entities become known as *classes* and *relationships* as *associations*. Moreover, class boxes publish the types of the attributes (including default values, if provided), and the signatures of the methods local to each class. Although these extensions represent a step forward, they don't change the relational semantics we presented above for GERs, as far as data modeling is concerned. So we move on.

*Name spaces and "heaps"*. Relational database referential integrity can be shown to be an instance of a more general issue which traverses computing from end to end: *name space* referential integrity (NSRI). The are so many instances of NSRI that *genericity* is the only effective way to address the topic [9] For this one *has to resort* to membership is its full genericity.

The issue is that, whatever programming language is adopted, one is faced with some ubiquitous syntactic ingredients: (a) source code is made of units; (b) units refer to other units; (c) units need to be named.

For instance, a software package is a (named) collection of modules, each module being made of (named) collections of data type declarations, of variable declarations, function declarations etc. Moreover, the package won't compile in case name spaces don't integrate with each other. Other examples of name spaces requiring NSRI are XML DTDs, grammars (where nonterminals play the rôle of names), etc.

In general, one is lead to heterogeneous (typed) collections of (mutually dependent) *name spaces*, nicely modeled as simple relations again

$$N_i \rightharpoonup \mathsf{F}_i(T_i, N_1, \ldots, N_j, \ldots, N_n)$$

where $\mathsf{F}_i$ is a parametric type describing the particular pattern which expresses how names of type $N_i$ depend on names of types $N_j$ ($j = 1, n$) and where $T_i$ aggregates all types which don't participate in NSRI.

Assuming that all such $\mathsf{F}_i$ have membership, we can draw diagram

$$\begin{array}{ccc} N_i & \xrightarrow{S_i} & \mathsf{F}_i(T_i, N_1, \ldots, N_i, \ldots, N_n) \\ & \searrow^{\in_{i,j} \cdot S_i} & \downarrow{\scriptstyle \in_{i,j}} \\ & & N_j \end{array}$$

where $\in_{i,j} \cdot S_i$ is a name-to-name relation, or *dependence graph*. Overall NSRI will hold iff

$$\langle \forall\, i, j \;::\; (\in_{i,j} \cdot S_i)^\circ \preceq S_j \rangle \tag{74}$$

Of course, this includes self referential integrity as a special case ($i = j$).

NSRI also shows up at low level, where data structures such as *caches* and *heaps* can also be thought of as name spaces: at such a low level, names are memory addresses (natural numbers). For instance,

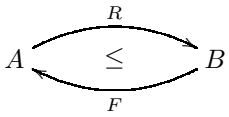
$$I\!N \xrightarrow{H} \mathsf{F}\,(T, I\!N)$$

models a heap "of shape" $\mathsf{F}$ where $T$ is some datatype of interest. A heap satisfies NSRI iff it has no dangling pointers. We shall be back to this model of heaps when discussing how to deal with recursive data models (section 10).

[9] For further dimensions and implications of *naming* see eg. Robin Milner's interesting essay *What's in a name? (in honour of Roger Needham)* available from `http://www.cl.cam.ac.uk/řm135`.

## 6 Data impedance mismatch expressed in the PF-style

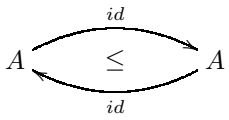
Now that both the PF-notation has been presented and that its application to describing the semantics of data structures has been illustrated, we are better positioned to restate and study diagram (6):




This expresses the *data impedance mismatch* between two data models $A$ and $B$ as witnessed by a *connected* representation/abstraction pair $(R, F)$. Formally, this means that:

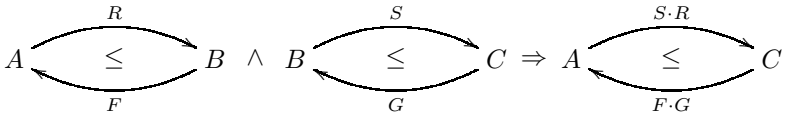
- $R$ is a representation ($\ker R = id$)
- $F$ is an abstraction ($\mathsf{img}\ F = id$)
- $R$ and $S$ are connected: $R \subseteq F^\circ$.

The least impedance mismatch possible happens between a datatype and itself,




(75)

a situation in which there is *no impedance* at all. Another way to read (75) is to say that the $\leq$-ordering on data is *reflexive*. It turns up that $\leq$ is also *transitive*,




(76)

that is, data impedances compose. The calculation of (76) is immediate: composition respects abstractions and representations (recall exercise 3) and $(F \cdot G, S \cdot R)$ are connected:

$$
\begin{array}{ll}
& S \cdot R \subseteq (F \cdot G)^\circ \\
\equiv & \quad \{ \text{ converses (13) } \} \\
& S \cdot R \subseteq G^\circ \cdot F^\circ \\
\Leftarrow & \quad \{ \text{ monotonicity } \} \\
& S \subseteq G^\circ \ \wedge \ R \subseteq F^\circ \\
\equiv & \quad \{ \text{ since } S, G \text{ and } R, F \text{ are assumed connected } \} \\
& \text{TRUE}
\end{array}
$$

*Right-invertibility.* A most beneficial consequence of (6) is the *right-invertibility* property, written

$$F \cdot R = id \tag{77}$$

Written in predicate logic, (77) expands to

$$\langle \forall\, a', a \;::\; \langle \exists\, b \;::\; a'\; F\; b \;\wedge\; b\; R\; a \rangle \;\equiv\; a' = a \rangle \tag{78}$$

The PF-calculation of (77) is easy:

$$\begin{array}{ll}
& F \cdot R = id \\
\equiv & \quad \{ \text{ equality of relations (11) } \} \\
& F \cdot R \subseteq id \;\wedge\; id \subseteq F \cdot R \\
\equiv & \quad \{ \; \mathsf{img}\, F = id \text{ and } \mathsf{ker}\, R = id \; \} \\
& F \cdot R \subseteq F \cdot F^{\circ} \;\wedge\; R^{\circ} \cdot R \subseteq F \cdot R \\
\equiv & \quad \{ \text{ converses } \} \\
& F \cdot R \subseteq F \cdot F^{\circ} \;\wedge\; R^{\circ} \cdot R \subseteq R^{\circ} \cdot F^{\circ} \\
\Leftarrow & \quad \{ \; (F\cdot) \text{ and } (R^{\circ}\cdot) \text{ are monotone (25) } \} \\
& R \subseteq F^{\circ} \;\wedge\; R \subseteq F^{\circ} \\
\equiv & \quad \{ \text{ trivia } \} \\
& R \subseteq F^{\circ} \\
\equiv & \quad \{ \text{ (6) ensures } R \text{ and } F \text{ connected } \} \\
& \text{TRUE}
\end{array}$$

Clearly, this *right-invertibility* property is what matters in data representation: $id \subseteq F \cdot R$ ensures the **no loss** principle and $F \cdot R \subseteq id$ ensures the **no confusion** principle.

*Exercise 9.* The reader may be interested to compare the calculation just above with the corresponding proof carried out at pointwise level using quantified logic expressions. This will amount to showing that (78) is entailed by the *pointwise* statement of $(F, R)$ as a connected abstraction/representation pair. (Good luck!)
□

While (as we have seen) $F \cdot R = id$ is entailed by (6), the converse entailment *does not hold*: $F \cdot R = id$ ensures $R$ a representation and $F$ surjective, but simple. It may be also the case that $F \cdot R = id$ and $R \subseteq F^{\circ}$ does not hold, as the following counter-example shows: $R = !^{\circ}$ and $\bot \subset F \subset !$.

*Exercise 10.* Consider two data structuring patterns: *"pointer to struct"* $(A \times B + 1)$ and *"pointer in struct"* $((A + 1) \times B)$. The question is: which of these data patterns represents the other? We suggest the reader to check the validity of

$$A \times B + 1 \;\underset{f}{\overset{R}{\leq}}\; (A + 1) \times B \tag{79}$$

where

$$R \stackrel{\text{def}}{=} [i_1 \times id \,,\langle i_2, !^\circ\rangle] \tag{80}$$

and $f = R^\circ$, that is

$$f(i_1\ a, b) = i_1(a, b) \tag{81}$$
$$f(i_2\ \text{NIL}, b) = i_2\ \text{NIL} \tag{82}$$

where NIL denotes the unique inhabitant of type 1.
□

A particular case where right-invertibility is *equivalent* to (6) happens wherever both the abstraction and the representation are functions, say $f, r$:

$$A \underset{f}{\overset{r}{\leq}} C \qquad \equiv \qquad f \cdot r = id \tag{83}$$

That $f \cdot r = id$ equivales $r \subseteq f^\circ$ and entails $f$ surjective and $r$ injective is easy to calculate:

$$\begin{array}{ll} & f \cdot r = id \\ \equiv & \quad \{\ (28)\ \} \\ & f \cdot r \subseteq id \\ \equiv & \quad \{\ \text{shunting (26)}\ \} \\ & r \subseteq f^\circ \\ \Rightarrow & \quad \{\ \text{composition is monotonic}\ \} \\ & f \cdot r \subseteq f \cdot f^\circ\ \wedge\ r^\circ \cdot r \subseteq r^\circ \cdot f^\circ \\ \equiv & \quad \{\ f \cdot r = id\ ;\ \text{converses}\ \} \\ & id \subseteq f \cdot f^\circ\ \wedge\ r^\circ \cdot r \subseteq id \\ \equiv & \quad \{\ \text{definitions}\ \} \\ & f\ \text{surjective}\ \wedge\ r\ \text{injective} \end{array}$$

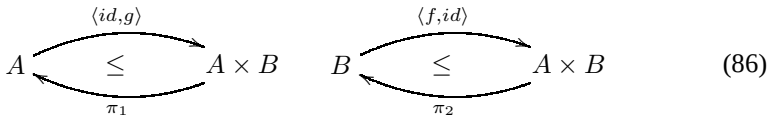
The right invertibility property can be a handy way of spotting $\leq$ rules. For instance, the following cancellation properties of product and sum hold [10]:

$$\pi_1 \cdot \langle f, g\rangle = f\ ,\ \pi_2 \cdot \langle f, g\rangle = g \tag{84}$$
$$[g\ , f] \cdot i_1 = g\ ,\ [g\ , f] \cdot i_2 = f \tag{85}$$

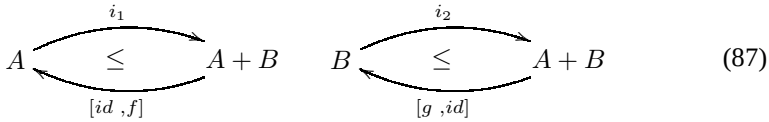
Suitable instantiation of $f$, $g$ to the identity function in both lines above lead to

$$\pi_1 \cdot \langle id, g\rangle = id\ ,\ \pi_2 \cdot \langle f, id\rangle = id$$
$$[id\ , f] \cdot i_1 = id\ ,\ [g\ , id] \cdot i_2 = id$$

Thus we get — via (83) — the following $\leq$-rules for free:

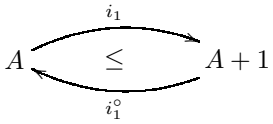



(86)




(87)

The rules above tell the two projections surjective and the two injections injective (as expected). At programming level, they ensure that adding entries to a `struct` or (disjoint) `union` is a valid representation strategy, provided functions $f, g$ are supplied by default [15]. Alternatively, they can be replaced by the top relation $\top$ (meaning a *don't care* representation strategy). In the case of (87), even $\bot$ will work instead of $f, g$, leading, for $A = 1$, to the standard representation of datatype $A$ by a "*pointer* to $A$":




*Exercise 11.* Show that $[id\ ,\bot] = i_1^\circ$ and that $[\bot\ , id] = i_2^\circ$. (Easy.)
□

*Isomorphic data types.* As instance of (83) consider the case of $f$ and $r$ such that both


(88)

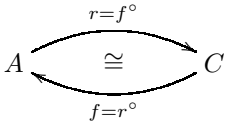
hold. This equivales

$$\begin{array}{ll} & r \subseteq f^\circ \ \wedge \ f \subseteq r^\circ \\ \equiv & \quad \{ \text{ converses ; (11) } \} \\ & r = f^\circ \end{array} \tag{89}$$

So $r$ (a function) is the converse of another function $f$. This means that both are bijections (isomorphisms) — recall recalling Fig. 2 — since

$$f \text{ is an isomorphism} \equiv f^\circ \text{ is a function} \tag{90}$$

In a diagram:




(91)

Isomorphism $A \cong C$ corresponds to *minimal* impedance mismatch between types $A$ and $C$ in the sense that, although the format of data changes, data conversion in both ways is wholly recoverable. That is, the isomorphic types $A$ and $C$ are "abstractly" the same. Here is a trivial example

$$A \times B \underset{swap}{\overset{swap}{\cong}} B \times A \tag{92}$$

where $swap$ is the name given to polymorphic function $\langle \pi_2, \pi_1 \rangle$. This isomorphism establishes the *commutativity* of $\times$, whose translation into practice is obvious: one can change the order in which the entries in a `struct` (eg. in C) are listed; swap the order of two columns in a spreadsheet, etc.

Naturally, the question arises: how can one be *certain* that $swap$ is an isomorphism? A constructive, rather elegant way is to follow the advice in (90), which invites one to calculate the converse of $swap$,

$$\begin{array}{ll} & swap^\circ \\ = & \quad \{ \text{ (51) } \} \\ & (\pi_1^\circ \cdot \pi_2 \cap \pi_2^\circ \cdot \pi_1)^\circ \\ = & \quad \{ \text{ converses } \} \\ & \pi_2^\circ \cdot \pi_1 \cap \pi_1^\circ \cdot \pi_2 \\ = & \quad \{ \text{ (51) again } \} \\ & swap \end{array}$$

which is $swap$ again. So $swap$ is its own inverse and therefore an isomorphism.

*Exercise 12.* The calculation just above was too simple. To recognize the power of (90), prove the associative property of disjoint union,

$$A + (B + C) \underset{f=[id+i_1\ ,i_2 \cdot i_2]}{\overset{r}{\cong}} (A + B) + C \tag{93}$$

by calculating *the function* $r$ which is the converse of $f$.

Appreciate the elegance of this strategy when compared to what is conventional in discrete maths: to prove $f$ bijective, one would have to either prove $f$ injective and surjective, or *invent* its converse $f^\circ$ and prove the two cancellations $f \cdot f^\circ = id$ and $f^\circ \cdot f = id$. (A lot of work!) □

*Exercise 13.* The following are known facts involving sums and products:

$$A \times (B \times C) \cong (A \times B) \times C \tag{94}$$
$$A \cong A \times 1 \tag{95}$$
$$A \cong 1 \times A \tag{96}$$
$$A + B \cong B + A \tag{97}$$
$$C \times (A + B) \cong C \times A + C \times B \tag{98}$$

Guess the relevant isomorphism pairs.
□

*Exercise 14.* Show that (90) holds, for $f$ a function (of course).
□

*Relation transposes.* Once again let us have a look at isomorphism pair $(f, r)$ in (89), this time to introduce variables in the equality:

$$\begin{array}{ll} & r = f^\circ \\ \equiv & \quad \{ \text{ introduce variables } \} \\ & \langle \forall\ a, c\ ::\ c\ r\ a \equiv c\ (f^\circ)\ a \rangle \\ \equiv & \quad \{ \text{ (21) } \} \\ & \langle \forall\ a, c\ ::\ c = r\ a \equiv (f\ c) = a \rangle \end{array}$$

This is a pattern shared by many (pairs of) operators in the relational calculus, as is the case of eg. (omitting the universal quantifier)

$$k = \Lambda R \ \equiv \ R = \ \in \cdot k \tag{99}$$

where $\Lambda$ converts a binary relation into *the corresponding* set-valued function [10], of

$$k = tot\ S \ \equiv \ S = i_1^\circ \cdot k \tag{100}$$

where $tot$ *totalizes* a simple relation $S$ into *the corresponding* "Maybe-function" [10], and of

$$k = curry\ f \ \equiv \ f = \underbrace{ap \cdot (k \times id)}_{uncurry\ k} \tag{101}$$

where $curry$ converts a two-argument function $f$ into *the corresponding* unary function, for $ap(g, x) = g\ x$.

These properties of $\Lambda$, $tot$ and $curry$ are normally referred to as *universal properties*, because of their particular pattern of universal quantification. Novice readers will find them more palatable once further (quantified) variables are introduced on their right hand sides:

$$\begin{array}{rcl} k = \Lambda R & \equiv & \langle \forall\ b, a\ ::\ b\ R\ a \equiv b \in (k\ a) \rangle \\ k = tot\ S & \equiv & \langle \forall\ b, a\ ::\ b\ S\ a \equiv (i_1 b) = k\ a \rangle \\ k = curry\ f & \equiv & \langle \forall\ b, a\ ::\ f(b, a) = (k\ b) a \rangle \end{array}$$

[10] See [53]. This corresponds to the view that simple relations are "possibly-undefined" functions. Recall that $A + 1 \xleftarrow{i_1^\circ} A$ is the membership of $Maybe$.

In summary, $\Lambda$, $tot$ and $curry$ are all isomorphisms. Here they are expressed by $\cong$-diagrams,

$$\begin{array}{c} A \to B \underset{(\in\cdot)}{\overset{\Lambda}{\cong}} (\mathcal{P}B)^A \quad A \rightharpoonup B \underset{untot=(i_1^\circ\cdot)}{\overset{tot}{\cong}} (B+1)^A \\ B^{C\times A} \underset{uncurry}{\overset{curry}{\cong}} (B^A)^C \end{array} \tag{102}$$

where the exponential notation $B^A$ describes the datatype of all functions from $A$ to $B$.

*Exercise 15.* (For Haskell programmers) Inspect the type of `flip lookup` and relate it to that of $tot$. (NB: `flip` is available from `GHC.Base` and `lookup` from `GHC.ListA`.)

*Exercise 16.* The following is a well-known isomorphism involving functions.

$$(B \times C)^A \underset{\langle -,- \rangle}{\overset{\langle (\pi_1 \cdot),(\pi_2 \cdot)\rangle}{\cong}} B^A \times C^A \tag{103}$$

Write down the *universal property* captured by (103).
□

*Exercise 17.* Relate the following function $p2p$ (read: *"pair to power"*)

$$(p2p\ p)b = if\ \ b\ \ then\ \ (\pi_1\ p)\ \ else\ \ (\pi_2\ p) \tag{104}$$

with isomorphism

$$A \times A \ \ \cong \ \ A^2 \tag{105}$$

□

Since exponentials are inhabited by functions and these are special cases of relations, there must be combinators which express functions in terms of relations and vice versa. Isomorphisms $\Lambda$ and $tot$ (99, 100) already establish relationships of this kind. Let us see two more which will prove useful in calculations to follow.

*"Relational currying"*. Consider isomorphism

$$(C \to A)^B \underset{(\underline{\ })}{\overset{(\overline{\ })^\circ}{\cong}} B \times C \to A \tag{106}$$

and associated universal property,

$$k = \overline{R} \equiv \langle \forall\ a, b, c\ ::\ a\ (k\ b)\ c \equiv a\ R\ (b, c) \rangle \tag{107}$$

where we suggest that $\overline{R}$ be read "$R$ transposed". $\overline{R}$ is thus a relation-valued function which expresses a kind of *selection/projection* mechanism: given some particular $b_0$, $\overline{R}\ b_0$ selects the "sub-relation" of $R$ of all pairs $(a, c)$ related to $b_0$.

This extension of *currying* to relations is a direct consequence of (99):

$$\begin{array}{ll} & B \times C \to A \\ \cong & \quad \{\ \Lambda/(\in\cdot)\ (99,\ 102)\ \} \\ & \exp B \times C(\mathcal{P}A) \\ \cong & \quad \{\ curry/uncurry\ \} \\ & \exp B(\exp C(\mathcal{P}A)) \\ \cong & \quad \{\ \text{exponentials preserve isomorphisms}\ \} \\ & (C \to A)^B \end{array}$$

The fact that, for simple relations, one could have resorted above to the $Maybe$-transpose (100) instead of the power transpose (99), leads to the conclusion that relational "currying" preserves simplicity:

$$(C \rightharpoonup A)^B \underset{(\bar{\_})}{\overset{(\bar{\_})^\circ}{\cong}} B \times C \rightharpoonup A \tag{108}$$

Since all relations are simple, we can use notation convention (36) in the following pointwise definition of $\overline{M}$:

$$\overline{M}\ b = \{c \mapsto M(b', c) \mid (b', c) \in dom\ M\ \wedge\ b' = b\} \tag{109}$$

This rule will play its rôle in multiple (foreign) key synthesis.

*Sets are fragments of* "bang". We have already seen that sets can be modeled by coreflexive relations, which are simple. *Characteristic* functions are another way to represent sets:

$$2^A \underset{\lambda S.\lambda a.a \in S}{\overset{\lambda p.\{a \in A \mid p\ a\}}{\cong}} \mathcal{P}A \qquad \text{cf.} \quad p = (\in S)\ \equiv\ S = \{a \mid p\ a\} \tag{110}$$

Here we see the correspondence between set comprehension and membership testing expressed by 2-valued functions, ie. predicates.

By combining the *tot/untot* isomorphism (102) with (110) we obtain

$$\mathcal{P}A \underset{dom}{\overset{s2m}{\cong}} A \rightharpoonup 1 \tag{111}$$

where $s2m\ S = ! \cdot [\![S]\!]$ and $dom$ is defined by (35). This shows that that every fragment of "bang" $(!)$ models a set [11].

*Exercise 18.* Show that "obvious" facts such as $S = \{a \mid a \in S\}$ and $p\ x \equiv x \in \{a \mid p\ a\}$ stem from (110). Investigate other properties of set-comprehension notation which can be drawn from (110).
□

*Relators and $\leq$-monotonicity.* A lesson learned from (83) is that everywhere one finds a surjection one finds a particular $\leq$-rule. For instance, predicate $f\ n \stackrel{\text{def}}{=} n \neq 0$ over the integers is surjective (onto the Booleans). Thus Booleans can be represented by integers, $2 \leq \mathbb{Z}$ — a fact C programmers know very well. Of course, one expects this *"to scale up"*: any data structure involving the Booleans (eg. sets of Booleans) can be represented by a similar structure involving integers (eg. sets of integers). However, what does the word "similar" mean in this context? Typically, when building a tree of integers, a C programmer looks at this tree and "sees" the tree with the same geometry where the integers have been replaced by their $f$ images.

In general, let $A$ and $B$ be such that $A \leq B$ and let $\mathsf{G}\,X$ denote a type parametric on $X$. We want to be able to scale up the $A$-into-$B$ representation to structures of type $\mathsf{G}$:

$$A \underset{F}{\overset{R}{\leq}} B \quad \Rightarrow \quad \mathsf{G}\,A \underset{\mathsf{G}\,F}{\overset{\mathsf{G}\,R}{\leq}} \mathsf{G}\,B \tag{112}$$

The questions arise: does this hold for *any* parametric type $\mathsf{G}$ we can think of? what do $\mathsf{G}\,R$ and $\mathsf{G}\,F$ actually mean? Let us check. First of all, we investigate conditions for $(\mathsf{G}\,F, \mathsf{G}\,R)$ to be connected:

$$\begin{array}{ll} & \mathsf{G}\,R \subseteq (\mathsf{G}\,F)^\circ \\ \Leftarrow & \quad \{\ \text{assume } \mathsf{G}(X^\circ) \subseteq (\mathsf{G}\,X)^\circ, \text{ for all } X\ \} \\ & \mathsf{G}\,R \subseteq \mathsf{G}(F^\circ) \\ \Leftarrow & \quad \{\ \text{assume monotonicity of } \mathsf{G}\ \} \\ & R \subseteq F^\circ \\ \equiv & \quad \{\ R \text{ is assumed connected to } F\ \} \\ & \text{TRUE} \end{array}$$

Next, $\mathsf{G}\,R$ must be injective:

$$\begin{array}{ll} & (\mathsf{G}\,R)^\circ \cdot \mathsf{G}\,R \subseteq id \\ \Leftarrow & \quad \{\ \text{assume } (\mathsf{G}\,X)^\circ \subseteq \mathsf{G}(X^\circ)\ \} \end{array}$$

[11] Relations at most $(!)$ are referred to as *right-conditions* in [26].

$$
\begin{array}{ll}
 & (\mathsf{G}\,R^\circ)\cdot\mathsf{G}\,R \subseteq id \\
\Leftarrow & \quad \{\ \text{assume } (\mathsf{G}\,R)\cdot(\mathsf{G}\,T) \subseteq \mathsf{G}(R\cdot T)\ \} \\
 & \mathsf{G}(R^\circ\cdot R) \subseteq id \\
\Leftarrow & \quad \{\ \text{assume } \mathsf{G}\,id \subseteq id \text{ and monotonicity of } \mathsf{G}\ \} \\
 & R^\circ\cdot R \subseteq id \\
\equiv & \quad \{\ R \text{ is injective}\ \} \\
 & \textsc{True}
\end{array}
$$

The reader eager to pursue checking the other requirements ($R$ entire, $F$ surjective, etc) will find out that the wish list concerning $\mathsf{G}$ will end up being as follows:

$$\mathsf{G}\,id = id \tag{113}$$

$$\mathsf{G}\,(R\cdot S) = (\mathsf{G}\,R)\cdot(\mathsf{G}\,S) \tag{114}$$

$$\mathsf{G}\,(R^\circ) = (\mathsf{G}\,R)^\circ \tag{115}$$

$$R\subseteq S \Rightarrow \mathsf{G}\,R \subseteq \mathsf{G}\,S \tag{116}$$

These turn up being the properties of a *relator* [5], a concept which extends that of a *functor* to relations: a parametric datatype $\mathsf{G}$ is said to be a relator wherever, given a relation from $A$ to $B$, $\mathsf{G}\,R$ extends $R$ to $\mathsf{G}$-structures. In other words, it is a relation from $\mathsf{G}\,A$ to $\mathsf{G}\,B$

$$\begin{array}{ccc} A & \cdots\cdots & \mathsf{G}\,A \\ {\scriptstyle R}\downarrow & & \downarrow{\scriptstyle \mathsf{G}\,R} \\ B & \cdots\cdots & \mathsf{G}\,B \end{array} \tag{117}$$

which obeys the properties above (it commutes with the identity, with composition and with converse, and it is monotonic). Once $R, S$ above are restricted to functions, the behaviour of $\mathsf{G}$ in (113, 114) is that of a functor, and (115) and (116) become trivial — the former establishing that $\mathsf{G}$ preserves isomorphisms and the latter that $\mathsf{G}$ preserves equality (Leibniz).

It is easy to show that relators preserve all basic properties of relations as in Fig. 2. Two trivial relators are the *identity* relator $\mathsf{Id}$, which is such that

$$\mathsf{Id}\,R = R$$

and the *constant* relator $\mathsf{K}$ (for a given data type $K$) which is such that

$$\mathsf{K}\,R = id_K$$

Relators can also be multi-parametric and we have already seen two of these: product $R\times S$ (49) and sum $R+S$ (58).

The prominence of parametric type $\mathsf{G}\,X = K \rightharpoonup X$, for $K$ a given datatype $K$ of *keys*, leads us to the investigation of its properties as a relator,

$$\begin{array}{ccc} B & \cdots\cdots & K \rightharpoonup B \\ {\scriptstyle R}\downarrow & & \downarrow{\scriptstyle K \rightharpoonup R} \\ C & \cdots\cdots & K \rightharpoonup C \end{array} \tag{118}$$

where we define $K \rightharpoonup R$ as follows:

$$N(K \rightharpoonup R)M \stackrel{\text{def}}{=} \delta\, M = \delta\, N \;\wedge\; N \cdot M^{\circ} \subseteq R \tag{119}$$

So, wherever simple $N$ and $M$ are $(K \rightharpoonup R)$-related, they are equally defined and their outputs are $R$-related. Wherever $R$ is a function $f$, $K \rightharpoonup f$ is a function too defined by projection

$$(K \rightharpoonup f)M = f \cdot M \tag{120}$$

This can be extended to a bi-relator,

$$(g \rightharpoonup f)M = f \cdot M \cdot g^{\circ} \tag{121}$$

provided $g$ is injective (recall (36)).

*Exercise 19.* Show that instantiation $R := f$ in (119) leads to $N \subseteq f \cdot M$ and $f \cdot M \subseteq N$ in the body of (119), and therefore to (120).
□

*Exercise 20.* Show that $K \rightharpoonup \_)$ is a relator.
□

*Indirection and dereferencing.* Indirection is a representation technique whereby data of interest stored in some data structure is replaced by references (pointers) to some global (dynamic) store — recall (68) — where the data is *actually* kept. The representation implicit in this technique involves allocating fresh cells in the global store; the abstraction consists in retrieving data by pointer dereferencing.

The motivation for this kind of representation is well-known: the referent is more expensive to move around than the reference. Despite being well understood and very widely used, dereferencing is a permanent source of errors in programming: it is impossible to retrieve data from a non-allocated reference.

To see how this strategy arises, consider $B$ in (117) the datatype of interest (archived in some parametric container of type $\mathsf{G}$, eg. binary trees of $B$s). Let $A$ be the natural numbers and $R$ be simple. Since relators preserve simplicity, $\mathsf{G}\,R$ will be simple too:

$$\begin{array}{ccc} \mathbb{N} & \cdots\cdots & \mathsf{G}\,\mathbb{N} \\ {\scriptstyle R}\downarrow & & \downarrow{\scriptstyle \mathsf{G}\,R} \\ B & \cdots\cdots & \mathsf{G}\,B \end{array}$$

The meaning of this diagram is that of declaring a generic function (say $rmap$) which, giving $R$ simple, yields $\mathsf{G}\,R$ also simple. So $rmap$ has type

$$(\mathbb{N} \rightharpoonup B) \to \mathsf{G}\,\mathbb{N} \rightharpoonup \mathsf{G}\,B \tag{122}$$

— in the same way the *fmap* function of Haskell class `Functor` has type

```
fmap :: (a -> b) -> (g a -> g b)
```

(Recall that, once restricted to functions, relators coincide with functors.)

From (108) we infer that $rmap$ can be "uncurried" into a simple relation of type

$$((\mathbb{N} \rightharpoonup B) \times \mathsf{G}\,\mathbb{N}) \rightharpoonup \mathsf{G}\,B \tag{123}$$

which, for finite structures, is surjective. Of course we can replace $\mathbb{N}$ above by any data domain, say $K$ (suggestive of *key*) with the same cardinality, that is, such that $K \cong \mathbb{N}$. Then

$$\mathsf{G}\,B \underset{Dref}{\overset{R}{\leq}} (K \rightharpoonup B) \times \mathsf{G}\,K \tag{124}$$

holds, where the abstraction relation $Dref$, which is such that $\overline{Dref} = rmap$, is defined according to (107)

$$y\ Dref\ (S, x) \equiv y(\mathsf{G}\,S)x$$

for $S$ a *store* and $x$ a data structure of pointers (inhabitant of $\mathsf{G}\,K$).

Consider as example the indirect representation of finite lists of $B$s,

$$l'\ Dref\ (S, l) \equiv l'(S^\star)l$$

where

$$l'(S^\star)l \ \equiv\ length\,l' = length\,l\ \wedge\ \langle \forall\, k\ :\ 1 \leq k \leq length\,l :\ (l'\ k) = S(l\ k)\rangle$$

So, wherever $l'S^\star l$ holds, no reference $k$ in $l$ can live outside the domain of the store $S$:

$$k \in l \Rightarrow \langle \exists\, b\ ::\ b\ S\ k\rangle \tag{125}$$

where $\in$ denotes finite list membership. This ensures (by construction) the correctness of the representation $R = Dref^\circ$ in (107): all references can be dereferenced.

*Exercise 21.* Check that (125) is another instance of NSRI (74),

$$(\in \cdot\, \underline{l})^\circ \preceq S$$

where $\underline{l}$ denotes the "everywhere $l$" constant function.
□

*Exercise 22.* Define a representation function $r \subseteq Dref^\circ$ (124) for $\mathsf{G}\,X = X^\star$.
□

## 7 Calculating database schemes from abstract models

Relational schema modeling is central to the "open-ended list of mapping issues" identified in [35]. In this section we develop a number of $\leq$-rules intended for cross-cutting impedance mismatch with respect to relational modeling.

In other words, we intend to provide a practical method for inferring the schema of a database which (correctly) implements a given abstract model, including the stepwise synthesis of the associated abstraction and representation data mappings and concrete invariants. This method will be shown to extend to recursive structures in section 10.

*Relational schemes "relationally".* Broadly speaking, a relational database is a $n$-tuple of tables, where each table is a relation involving value-level tuples. Tuples are vectors of values which inhabit "atomic" data types, that is, which hold data with no further structure. Since many such relations (tables) exhibit *keys*, they can be thought of as *simple relations*. In this context, let

$$RDBT = \prod_{i=1}^{n}(\prod_{j=1}^{n_i} K_j \rightharpoonup \prod_{k=1}^{m_i} D_k) \tag{126}$$

denote the *generic type* of a relational database [2]. Every $RDBT$-compliant tuple $db$ is a collection of $n$ relational tables (index $i = 1, n$) each of which is a mapping from a tuple of *keys* (index $j$) to a tuple of relevant *data* (index $k$). Wherever $m_i = 0$ we have $\prod_{k=1}^{0} D_k \cong 1$, meaning — via (111) — a *finite set* of tuples of type $\prod_{j=1}^{n_i} K_j$. (These are called *relationships* in the standard terminology.) Wherever $n_i = 1$ we are in presence of a singleton relational table. Last but not least, all $K_j$ and $D_k$ are "atomic" types, otherwise $db$ would fail first normal form (1NF) compliance [38].

Compared to what we have seen so far, $RDBT$ (126) is "flat": there are no sums, no exponentials, no room for a single recursive datatype. Thus the mismatch identified in [35]: how does one map structured data, eg. encoded in XML, or a text generated from some grammar, or a collection of object types, into $RDBT$?

We devote the remainder of this section to a number of $\leq$-rules which can be used to transform arbitrary data models into instances of the "flat" $RDBT$. Such rules have the generic pattern $A \leq B$ (of which $A \cong B$ is a special case) where $B$ only contains products and simple relations. So, by successive application of such rules, one is lead — eventually — to an instance of $RDBT$. Note that (106) and (111) are already rules of this kind (from left to right), the latter enabling one to get rid of powersets and the other to get rid of (some forms of) exponentials. Below we present a few more rules of this kind.

*Getting rid of sums.* It can be shown (see eg. [10]) that the *either* combinator $[R\ , S]$ as defined by (57) is an isomorphism. This happens because one can always project a relation $(B + C) \xrightarrow{T} A$ into two components of types $B \xrightarrow{R} A$ and $C \xrightarrow{S} A$,

such that $T = [R\,,S]$. Thus we have

$$(B+C) \to A \;\underset{[\_\,,\_]}{\overset{[\_\,,\_]^\circ}{\cong}}\; (B \to A) \times (C \to A) \tag{127}$$

that is, universal property

$$T = [R\,,S] \;\equiv\; T \cdot i_1 = R \;\wedge\; T \cdot i_2 = S \tag{128}$$

holds.

When applied from left to right, rule (127) can be of help in removing sums from data models: relations whose input types involve sums can always be decomposed into pairs of relations whose types don't involve (such) sums.

Sums are a main ingredient in describing the *abstract syntax* of data. For instance, in the grammar approach to data modeling, alternative branches of a production (in extended BNF notation) map to polynomial sums, recall (59). The application of rule (127) removes such sums with no loss of information (it is an isomorphism), thus reducing the mismatch between abstract syntax and relational database models.

The calculation of (127), which is easily performed via the power-transpose [10], can alternatively be performed via the *Maybe*-transpose [53] — in the case of simple relations — meaning that relational *either* preserves simplicity:

$$(B+C) \rightharpoonup A \;\underset{[\_\,,\_]}{\overset{[\_\,,\_]^\circ}{\cong}}\; (B \rightharpoonup A) \times (C \rightharpoonup A) \tag{129}$$

What about the other (very common) circumstance in which sums occur at the output rather than at the input type of a simple relation? The following sum-elimination rule is applicable to such situations,

$$A \rightharpoonup (B+C) \;\underset{\overset{+}{\bowtie}}{\overset{\triangle_+}{\leq}}\; (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{130}$$

where

$$M \overset{+}{\bowtie} N \overset{\text{def}}{=} i_1 \cdot M \cup i_2 \cdot N \tag{131}$$

$$\triangle_+ M \overset{\text{def}}{=} (i_1^\circ \cdot M, i_2^\circ \cdot M) \tag{132}$$

That $\overset{+}{\bowtie}$ is not entire is easy to check: the union of two simple relations is not always simple. Let us thus calculate the (weakest) pre-condition for simplicity to be maintained:

$$M \overset{+}{\bowtie} N \text{ is simple}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ definition (131) } \} \\
& (i_1 \cdot M) \cup (i_2 \cdot N) \text{ is simple} \\
\equiv & \{ \text{ simplicity of union of simple relations (37) } \} \\
& (i_1 \cdot M) \cdot (i_2 \cdot N)^\circ \subseteq id \\
\equiv & \{ \text{ converses ; shunting (26, 27) } \} \\
& M \cdot N^\circ \subseteq i_1^\circ \cdot i_2 \\
\equiv & \{ \ i_1^\circ \cdot i_2 = \bot \text{ ; (32,33) } \} \\
& \delta\, M \cdot \delta\, N \subseteq \bot \\
\equiv & \{ \text{ coreflexives (42) } \} \\
& \delta\, M \cap \delta\, N \subseteq \bot
\end{array}
$$

Thus, $M \stackrel{+}{\bowtie} N$ is simple iff $M$ and $N$ are domain-disjoint.

*Exercise 23.* Show that $\stackrel{+}{\bowtie} \cdot \triangle_+ = id$ holds. (NB: property

$$id + id = id \tag{133}$$

can be of help in the calculation.)
□

*Getting rid of multivalued types.* Recall the $Books$ type (69) defined earlier on. It deviates from $RDBT$ in the second factor of its target type, $5 \rightharpoonup Author$, whereby book entries are bound to record up to 5 authors. How do we cope with this situation? $Books$ is an instance of the generic relational type

$$A \rightharpoonup (D \times (B \rightharpoonup C))$$

for arbitrary $A, B, C$ and $D$, where entry $B \rightharpoonup C$ generalizes the notion of a multivalued attribute. Our aim in the calculations which follow is to split this relation type in two so as to combine the two keys of types $A$ and $B$:

$$
\begin{array}{ll}
& A \rightharpoonup (D \times (B \rightharpoonup C)) \\
\cong & \{ \ Maybe \text{ transpose (102) } \} \\
& (D \times (B \rightharpoonup C) + 1)^A \\
\leq & \{ \text{ (79) } \} \\
& ((D + 1) \times (B \rightharpoonup C))^A \\
\cong & \{ \text{ splitting (103) } \} \\
& (D + 1)^A \times (B \rightharpoonup C)^A \\
\cong & \{ \ Maybe \text{ transpose (102, 106) } \} \\
& (A \rightharpoonup D) \times (A \times B \rightharpoonup C)
\end{array}
$$

Altogether, we can rely on $\leq$-rule

$$A \rightharpoonup (D \times (B \rightharpoonup C)) \quad \underset{\bowtie_n}{\overset{\triangle_n}{\leq}} \quad (A \rightharpoonup D) \times (A \times B \rightharpoonup C) \tag{134}$$

where the "nested join" operator $\bowtie_n$ is defined by

$$M \bowtie_n N = \langle M, \overline{N} \rangle \tag{135}$$

— recall (108) — and $\triangle_n$ is

$$\triangle_n M = (\pi_1 \cdot M, usc(\pi_2 \cdot M)) \tag{136}$$

where $usc$ (="undo simple currying") is defined in comprehension notation as follows:

$$usc\, M = \{(a, b) \mapsto (M\ a)b \mid a \in dom\ M, b \in dom(Ma)\} \tag{137}$$

(See reference [58] for details about the calculation of this abstraction / representation pair.)

*Example.* Let us see the application of $\leq$-rule (134) to the $Books$ data model (69). We document each step by pointing out the involved abstraction/representation pair:

$$\begin{array}{ll} Books & = \ ISBN \rightharpoonup (Title \times (5 \rightharpoonup Author) \times Publisher) \\ & \cong_1 \qquad \{\ r_1 = id \rightharpoonup \langle\langle \pi_1, \pi_3\rangle, \pi_2\rangle\ ,\ f_1 = id \rightharpoonup \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1\rangle\ \} \\ & \quad\ ISBN \rightharpoonup (Title \times Publisher) \times (5 \rightharpoonup Author) \\ & \leq_2 \qquad \{\ (134)\text{: } r_2 = \triangle_n\ ,\ f_2 = \bowtie_n\ \} \\ & \quad\ (ISBN \rightharpoonup Title \times Publisher) \times (ISBN \times 5 \rightharpoonup Author) \\ & = \ Books_2 \end{array}$$

Since $Books_2$ belongs to the $RDBT$ class of types (assuming $ISBN, Title, Publisher$ and $Author$ atomic) it is directly implementable as a relational database schema.

Altogether, we have been able to calculate a *type-level* mapping between a source data model ($Books$) and a target data model ($Books_2$). To carry on with the *mapping scenario* set up in [35], we need to be able to synthesize the two data maps ("map forward" and "map backward") between $Books$ and $Books_2$. We do this below as an exercise of PF-reasoning followed by pointwise translation.

Following rule (76), which enables composition of representations and abstractions, we synthesize

$$r = \triangle_n \cdot (id \rightharpoonup \langle\langle \pi_1, \pi_3\rangle, \pi_2\rangle)$$

as overall "map forward" representation, and

$$f = (id \rightharpoonup \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1\rangle) \cdot \bowtie_n$$

as overall "map backward" abstraction. Let us transcribe $r$ to pointwise notation:

$$\begin{aligned}
r\ M &= \triangle_n((id \rightharpoonup \langle\langle \pi_1, \pi_3\rangle, \pi_2\rangle)M) \\
&= \quad \{\ (121)\ \} \\
&\quad \triangle_n(\langle\langle \pi_1, \pi_3\rangle, \pi_2\rangle \cdot M) \\
&= \quad \{\ (136)\ \} \\
&\quad (\pi_1 \cdot \langle\langle \pi_1, \pi_3\rangle, \pi_2\rangle \cdot M, usc(\pi_2 \cdot \langle\langle \pi_1, \pi_3\rangle, \pi_2\rangle \cdot M)) \\
&= \quad \{\ \text{exercise 6 ; projections}\ \} \\
&\quad (\langle \pi_1, \pi_3\rangle \cdot M, usc(\pi_2 \cdot M))
\end{aligned}$$

Thanks to (36), the first component in this pair transforms to pointwise

$$\{isbn \mapsto (\pi_1(M\ isbn), \pi_3(M\ isbn)) \mid isbn \in dom\ M\}$$

and the second to

$$\{(isbn, a) \mapsto ((\pi_2 \cdot M)\ isbn)a \mid isbn \in dom\ M, a \in dom((\pi_2 \cdot M)isbn)\}$$

using definition (137).

The same kind of reasoning will lead us to overall abstraction ("map backward") $f$:

$$\begin{aligned}
f(M, N) &= (id \rightharpoonup \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1\rangle)(M \bowtie_n N) \\
&= \quad \{\ (121) \text{ and } (135)\ \} \\
&\quad \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1\rangle \cdot \langle M, \overline{N}\rangle \\
&= \quad \{\ \text{exercise 6 ; projections}\ \} \\
&\quad \langle \pi_1 \cdot \pi_1 \cdot \langle M, \overline{N}\rangle, \pi_2 \cdot \langle M, \overline{N}\rangle, \pi_2 \cdot \pi_1 \cdot \langle M, \overline{N}\rangle\rangle \\
&= \quad \{\ \text{exercise 7}\ \} \\
&\quad \langle \pi_1 \cdot M, \overline{N} \cdot \delta\, M, \pi_2 \cdot M\rangle \\
&= \quad \{\ (109)\ \} \\
&\quad \{isbn \mapsto (\pi_1(M\ isbn), N', \pi_2(M\ isbn)) \mid isbn \in dom\ M\}
\end{aligned}$$

where $N'$ abbreviates $\{n \mapsto N(i, n) \mid (i, n) \in dom\ N\ \wedge\ i = isbn\}$.

The fact that $\overline{N}$ is preconditioned by $\delta\, M$ in the abstraction is a clear indication that any addition to $N$ of authors of books whose $ISBN$ is not in $M$ is doomed to be ignored when 'backward mapping" the data. This explains why a foreign key constraint must be added to any SQL encoding of $Books_2$, eg.:

```
CREATE TABLE BOOKS (
  ISBN      VARCHAR (...) NOT NULL,
  Publisher VARCHAR (...) NOT NULL,
  Title     VARCHAR (...) NOT NULL,
```

```
  CONSTRAINT BOOKS PRIMARY KEY(ISBN)
);

CREATE TABLE AUTHORS (
  ISBN   VARCHAR (...) NOT NULL,
  Count  NUMBER  (...) NOT NULL,
  Author VARCHAR (...) NOT NULL,
  CONSTRAINT AUTHORS_pk PRIMARY KEY (ISBN,Count)
);

ALTER TABLE AUTHORS ADD CONSTRAINT AUTHORS_FK
  FOREIGN KEY (ISBN) REFERENCES BOOKS (ISBN);
```

It can be observed that this constraint is ensured by representation $r$ (otherwise right-invertibility wouldn't take place). Constraints of this kind are known as *concrete invariants*. We discuss this important notion in the section which follows.

## 8 Concrete invariants

Wherever one has a $\leq$-rule (6) one knows that $R$ and $F$ are connected (5), which in turn implies that the range of $R$ is at most the domain of $F$:

$$\rho\, R \subseteq \delta\, F$$

This means that the space of the representation ($B$) can be divided in three parts:

- *inside* $\rho\, R$ — data inside $\rho\, R$ are referred to as *canonical representatives*; the predicate associated to $\rho\, R$, which is the strongest property ensured by the representation, is referred to as the induced *concrete invariant*, or representation invariant.
- *outside* $\delta\, F$ — data outside $\delta\, F$ are *illegal* data: there is no way in which they can be retrieved; we say that the representation model is *corrupted* (using the database terminology) once its CRUD drives data into this zone.
- *inside* $\delta\, F$ *and outside* $\rho\, R$ — this part contains data values which $R$ never generates but which are retrievable and therefore regarded as *legal* representatives; however, if the CRUD of the target model lets data go into this zone, the range of the representation cannot be assumed as concrete invariant.

The following properties of range and domain

$$\rho\,(R \cdot S) = \rho\,(R \cdot \rho\, S) \tag{138}$$

$$\delta\,(R \cdot S) = \delta\,(\delta\, R \cdot S) \tag{139}$$

help in calculating *concrete invariants* induced by $\leq$-chaining (76).

Concrete invariant calculation, which is in general nontrivial, is softened wherever $\leq$-rules are expressed by GCs [12]. In this case, the range of the representation (concrete invariant) can be computed as coreflexive

$$r \cdot f \cap id \tag{140}$$

[12] Of course, these have to be *perfect* (75) on the source (abstract) side.

that is, predicate [13]

$$\phi\ x \stackrel{\text{def}}{=} r(f\ x) = x \tag{141}$$

As illustration of this process, consider law

$$A \to B \times C \underset{\langle -,- \rangle}{\overset{\langle (\pi_1 \cdot),(\pi_2 \cdot) \rangle}{\leq}} (A \to B) \times (A \to C) \tag{142}$$

which expresses the universal property of the *split* operator (a perfect GC):

$$X \subseteq \langle R, S \rangle \equiv \pi_1 \cdot X \subseteq R\ \wedge\ \pi_2 \cdot X \subseteq S \tag{143}$$

Calculation of the concrete invariant induced by (142) follows:

$$\begin{array}{ll} & \phi(R,S) \\ \equiv & \quad \{\ (141)\ \} \\ & (R,S) = (\pi_1 \cdot \langle R,S \rangle, \pi_2 \cdot \langle R,S \rangle) \\ \equiv & \quad \{\ (55)\ \} \\ & R = R \cdot \delta\, S\ \wedge\ S = S \cdot \delta\, R \\ \equiv & \quad \{\ \delta\, X \subseteq \Phi \equiv X \subseteq X \cdot \Phi\ \} \\ & \delta\, R \subseteq \delta\, S\ \wedge\ \delta\, S \subseteq \delta\, R \\ \equiv & \quad \{\ (11)\ \} \\ & \delta\, R = \delta\, S \end{array}$$

In other words: if equally defined $R$ and $S$ are joined and then decomposed again, this will be a lossless decomposition [52].

Similarly, the following concrete invariant can be shown to hold for rule (134) [14]:

$$\phi(M,N) \stackrel{\text{def}}{=} N \cdot \in^{\circ} \preceq M \tag{144}$$

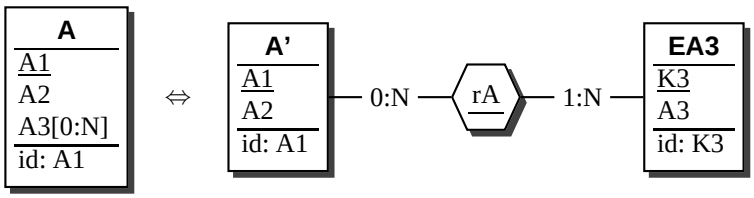
Finally note the very important fact that, in the case of $\leq$-rules supported by perfect GCs, the source datatype is actually *isomorphic* to the subset of the target datatype determined by the *concrete invariant* (as range of the representation function [15]).

## 9 Calculating model transformations

References [24] and [37] postulate a number of model transformation rules (about GERs in the first case and UML in the second) which we are in position to calculate.

[13] See Theorem 5.20 in [1].
[14] See [58] for details.
[15] See the *Unity of opposites* theorem of [4].

We illustrate this process with rule 12.2 of [24], the rule which converts a (multivalued) attribute into an entity type:




(145)

The PF-semantics of entity **A** are captured by simple relations from identity $A_1$ to attributes $A_2$ and $A_3$, this one represented by a powerset due to being [0:N]:

$$A_1 \rightharpoonup A_2 \times \mathcal{P}A_3$$

The main step in the calculation is the creation of the new entity **EA3** by indirection — recall (124) — whereafter we proceed as before:

$$\begin{array}{ll} & A_1 \rightharpoonup A_2 \times \mathcal{P}A_3 \\ \leq_1 & \quad \{\ (124)\ \} \\ & (K_3 \rightharpoonup A_3) \times (A_1 \rightharpoonup A_2 \times \mathcal{P}K_3) \\ \cong_2 & \quad \{\ (111)\ \} \\ & (K_3 \rightharpoonup A_3) \times (A_1 \rightharpoonup A_2 \times (K_3 \rightharpoonup 1)) \\ \leq_3 & \quad \{\ (134)\ \} \\ & (K_3 \rightharpoonup A_3) \times ((A_1 \rightharpoonup A_2) \times (A_1 \times K_3 \rightharpoonup 1)) \\ \cong_4 & \quad \{\ \text{introduce ternary product}\ \} \\ & \underbrace{(A_1 \rightharpoonup A_2)}_{A'} \times \underbrace{(A_1 \times K_3 \rightharpoonup 1)}_{rA} \times \underbrace{(K_3 \rightharpoonup A_3)}_{EA3} \end{array}$$

The overall concrete invariant is

$$\phi(M,R,N) = R \cdot \in^\circ \preceq M \ \wedge\ R \cdot \in^\circ \preceq N \tag{146}$$

— recall eg. (144) — which can be further transformed into:

$$\begin{array}{rl} \phi(M,R,N) = R \cdot \in^\circ \preceq M \ \wedge\ R \cdot \in^\circ \preceq N & \\ \equiv & \quad \{\ (65, 64)\ \} \\ & R \cdot \pi_1^\circ \preceq M \ \wedge\ R \cdot \pi_2^\circ \preceq N \\ \equiv & \quad \{\ (41)\ \} \\ & R \preceq M \cdot \pi_1 \ \wedge\ R \preceq N \cdot \pi_2 \end{array}$$

In words, this means that relationship $R$ (rA in the diagram) must be integrate referentially with $M$ (**A'** in the diagram) on the first attribute of its compound key and with $N$ (**EA3** in the diagram) wrt. the second attribute.

The reader eager to calculate the overall representation and abstraction relations will realize that the former is a relation, due to the fact that there are many ways in which the keys of the newly created entity can be associated to values of the $A3$ attribute. This association cannot be recovered once such keys are abstracted from. So, even restricted by the concrete invariant, the calculated model is surely a valid implementation of the original, but *not* isomorphic to it. Therefore, the rule should not be regarded as bidirectional.

## 10 On the impedance of recursive data models

Recursive data structuring is a source of data impedance mismatch because it is not *directly* supported in every programming environment. While functional programmers regard recursion as *the natural way* to programming, for instance, database programmers don't think in that way: somehow trees have to give room to flat data. Somewhere in between is (pointer-based) imperative programming and object oriented programming: direct support for recursive data structures doesn't exist, but dynamic memory management makes it possible to implement them as heap structures involving pointers or object identities.

In this section we address representation of recursive data structures in terms of non-recursive ones. In a sense, we want to see how to "get away with recursion" [50]. It is a standard result (and every a programmer's experience) that *recursive types using products and sums can be implemented using pointers* [61]. Our challenge is to generalize this result and present it in calculational style.

As we have seen already, recursive (finite) data structures are least solutions to equations of the form $X \cong \mathsf{G}\,X$, where $\mathsf{G}$ is a relator. The standard notation for such a solution is $\mu\mathsf{G}$. (This always exists when $\mathsf{G}$ is *regular* [10], a class which embodies all polynomial $\mathsf{G}$.)

Programming languages which implement datatype $\mu\mathsf{G}$ always do so by *wrapping* it inside some syntax. For instance, the Haskell declaration of datatype `PTree` (45) involves constructor `Node` and selectors `name`, `birth`, `mother` and `father`, which cannot be found in equation (61). But this is precisely why the equation expresses isomorphism and not equality: constructor and selectors participate in two bijections which witness the isomorphism and enable one to construct or inspect inhabitants of the datatype being declared. In general, we draw

$$\mu\mathsf{G} \underset{in}{\overset{out}{\cong}} \mathsf{G}\,\mu\mathsf{G}$$

where $in$ embodies the chosen syntax for constructing inhabitants of $\mu\mathsf{G}$ and $out = in^{\circ}$ embodies the syntax for destructing (inspecting) such inhabitants. For instance, the $in$

bijection associated with `PTree` (45) interpreted as solution to equation (61) is

$$in((n,b),m,f) = Node\ n\ b\ m\ f \tag{147}$$

Programs handling $\mu\mathsf{G}$ can be of essentially two kinds: either they read (parse, inspect) $\mu\mathsf{G}$-structures (vulg. trees) or they actually build such structures. The former kind is known as *folding* and the latter as *unfolding*, and both can be pictured as diagrams nicely exhibiting their recursive (inductive) nature:

$$\begin{array}{ccc} \mu\mathsf{G} & \xrightarrow{out} & \mathsf{G}\,\mu\mathsf{G} \\ {\scriptstyle fold\ R}\downarrow & & \downarrow{\scriptstyle \mathsf{G}(fold\ R)} \\ A & \xleftarrow[R]{} & \mathsf{G}\,A \end{array} \qquad \begin{array}{ccc} \mu\mathsf{G} & \xleftarrow{in} & \mathsf{G}\,\mu\mathsf{G} \\ {\scriptstyle unfold\ R}\uparrow & & \uparrow{\scriptstyle \mathsf{G}(unfold\ R)} \\ A & \xrightarrow[R]{} & \mathsf{G}\,A \end{array} \tag{148}$$

Both $fold$ and $unfold$ are instances of a more general, binary combinator known as *hylomorphism* [10] which is normally expressed using bracketed notation (151) to save parentheses:

$$unfold\ R = [\![\ in, R\ ]\!] \tag{149}$$

$$fold\ S = [\![\ R, out\ ]\!] \tag{150}$$

As fixed points (151), hylomorphisms enjoy a number of so-called *fusion* properties, two of which are listed below for their relevance in calculations to follow [16]:

$$\begin{array}{ccc} C & \xleftarrow{T} & \mathsf{G}\,C \\ {\scriptstyle V}\uparrow & & \uparrow{\scriptstyle \mathsf{G}\,V} \\ B & \xleftarrow{S} & \mathsf{G}\,B \\ {\scriptstyle [\![\ S,H\ ]\!]}\uparrow & & \uparrow{\scriptstyle \mathsf{G}\,[\![\ S,H\ ]\!]} \\ K & \xrightarrow{H} & \mathsf{G}\,K \\ {\scriptstyle R}\uparrow & & \uparrow{\scriptstyle \mathsf{G}\,R} \\ A & \xrightarrow[U]{} & \mathsf{G}\,A \end{array} \qquad [\![\ S,H\ ]\!] = \langle \mu\ X\ ::\ S \cdot (\mathsf{G}\,X) \cdot H \rangle \tag{151}$$

$$V \cdot [\![\ S,H\ ]\!] \subseteq [\![\ T,H\ ]\!] \Leftarrow V \cdot S \subseteq T \cdot (\mathsf{G}\,V) \tag{152}$$

$$[\![\ S,H\ ]\!] \cdot R = [\![\ S,U\ ]\!] \Leftarrow H \cdot R = (\mathsf{G}\,R) \cdot U \tag{153}$$

In (liberal) Haskell syntax we might write the type of the $unfold$ combinator as something like

```
unfold :: (a -> g a) -> a -> mu g
```

assuming only functions involved. If we generalize these to simple relations, we obtain the following type for function $unfold$

$$(A \rightharpoonup \mu\mathsf{G})^{(A \rightharpoonup \mathsf{G}\,A)} \tag{154}$$

which, thanks to (106), "uncurries" into $((A \rightharpoonup \mathsf{G}\,A) \times A) \rightharpoonup \mu\mathsf{G}$.

[16] These and other properties of hylomorphisms arise from the powerful $\mu$-*fusion* theorem [4] once the relational operators involved are identified as lower adjoints in GGs (25).

Let us temporarily assume that there exists a datatype $K$ such that simple relation $Unf$, of type $((K \rightharpoonup \mathsf{G}\,K) \times K) \rightharpoonup \mu\mathsf{G}$ and such that $\overline{Unf} = unfold$, is surjective. Then we are in condition to establish the $\leq$-equation which follows,

$$\mu\mathsf{G} \;\underset{Unf}{\overset{R}{\underset{\leq}{\rightleftarrows}}}\; \underbrace{(K \rightharpoonup \mathsf{G}\,K)}_{\text{“heap”}} \times K \tag{155}$$

where $K$ is regarded as a data type of *“heap addresses”*, or *“pointers”*, and $K \rightharpoonup \mathsf{G}\,K$ a datatype of $\mathsf{G}$-structured *heaps* [17]. So, assertion $t\ Unf\ (H, k)$ means that, if pair $(H, k)$ is in the domain of $Unf$, then the abstract value $t = (unfold\ H)k$ will be retrieved — recall (107). This corresponds to dereferencing $k$ in $H$ and carrying on doing so (structurally) while building (via $in$) the tree which corresponds to such a walk through the heap.

Termination of this process requires $H$ to be free of dangling references — ie. satisfy the NSRI property (74) — and to be referentially acyclic. This second requirement can also be expressed via the membership relation associated with $\mathsf{G}$: relation $K \xleftarrow{\in_{\mathsf{G}} \cdot H} K$ on references must be well-founded.

Jourdan [32] developed a pointwise proof of the surjectiveness of $Unf$ (155) for $K$ isomorphic to the natural numbers and $\mathsf{G}$ polynomial (see more about this in section 14). The representation relation $R$, which should be chosen among the entire sub-relations of $Unf^{\circ}$, is an injective *fold* (since converses of unfolds are folds [10]). Appendix A illustrates a strategy for encoding such folds, in the case of $\mathsf{G}$ polynomial and $K$ the natural-numbers.

“De-recursivation” law (155) generalizes, in the generic PF-style, the main result of [61] and bears some resemblance (at least in spirit) with “defunctionalization” [29], a technique which is used in program transformation and compilation. The genericity of this result and the ubiquity of its translation into practice — cf. name spaces, dynamic memory management, pointers and heaps, database files, object run-time systems, etc — turns it into a very useful device for cross-paradigm transformations. For instance, [50] shows how to calculate a universal SQL representation for arbitrary XML data.

The sections which follow will illustrate this potential, while stressing on genericity [30]. Operations of the *algebra of heaps* such as eg. *defragment*, *garbage-collect* will be stated generically and be shown to be correct with respect to the abstraction relation.

## 11 Cross-paradigm impedance handled by calculation

Let us resume work on the case study started in section 2 and finally show how to map the recursive datatype `PTree` (45) down to a relational model (SQL) via an intermediate heap/pointer representation.

Note that we shall be crossing over three paradigms — functional, imperative and database relational — in a single calculation, using the same (unified) notation:

[17] Technically, this view corresponds to regarding heaps as (finite) relational $\mathsf{G}$-coalgebras.

$$
\begin{array}{ll}
 & PTree \\
\cong_1 & \quad \{\ r_1 = out\ ,\ f_1 = in, \text{ for } \mathsf{G}\,K \stackrel{\text{def}}{=} Ind \times (K+1) \times (K+1) \text{ — cf. (61, 147)}\ \} \\
 & \mu\mathsf{G} \\
\leq_2 & \quad \{\ R_2 = Unf^\circ, F_2 = Unf \text{ — cf. (155)}\ \} \\
 & (K \rightharpoonup Ind \times (K+1) \times (K+1)) \times K \\
\cong_3 & \quad \{\ r_3 = (id \rightharpoonup flatr^\circ) \times id\ ,\ f_3 = (id \rightharpoonup flatr) \times id \text{ — cf. (52)}\ \} \\
 & (K \rightharpoonup Ind \times ((K+1) \times (K+1))) \times K \\
\cong_4 & \quad \{\ r_4 = (id \rightharpoonup id \times p2p) \times id\ ,\ f_4 = (id \rightharpoonup id \times p2p^\circ) \times id \text{ — cf. (105)}\ \} \\
 & (K \rightharpoonup Ind \times (K+1)^2) \times K \\
\cong_5 & \quad \{\ r_5 = (id \rightharpoonup id \times tot^\circ) \times id\ ,\ f_5 = (id \rightharpoonup id \times tot) \times id \text{ — cf. (100)}\ \} \\
 & (K \rightharpoonup Ind \times (2 \rightharpoonup K)) \times K \\
\leq_6 & \quad \{\ r_6 = \triangle_n\ ,\ f_6 = \bowtie_n \text{ — cf. (134)}\ \} \\
 & ((K \rightharpoonup Ind) \times (K \times 2 \rightharpoonup K)) \times K \\
\cong_7 & \quad \{\ r_7 = flatl\ ,\ f_7 = flatl^\circ \text{ — cf. (53)}\ \} \\
 & (K \rightharpoonup Ind) \times (K \times 2 \rightharpoonup K) \times K \\
=_8 & \quad \{\ \text{since } Ind = Name \times Birth \text{ (61)}\ \} \\
 & (K \rightharpoonup Name \times Birth) \times (K \times 2 \rightharpoonup K) \times K \qquad (156)
\end{array}
$$

In summary:

- Step 2 moves from the functional (inductive) to the pointer-based representation. At data level, this corresponds to mapping inductive tree (8) to the heap of Fig. 1a.
- Step 5 starts the move from pointer-based to relational-based representation. Isomorphism (100) between *Maybe*-functions and simple relations (which is the main theme of [53]) provides the relevant data-link between the two paradigms: pointers "become" primary/foreign keys.
- Steps 7 and 8 deliver an RDBT structure (illustrated in Fig. 1b) made up of two tables, one telling the details of each individual, and the other recording its immediate ancestors. The 2-valued attribute in the second table indicates whether the mother or the father of each individual is to be reached. The third factor in (156) is the key which gives access to the root of the original tree.

In practice, a final step is required, translating the relational data into the syntax of the target relational engine (eg. a list of SQL `INSERT` commands for each relation), bringing symmetry to the exercise: in either way (forwards or backwards), data mappings start by *removing* syntax and close by *introducing* syntax.

*Exercise 24.* Let $f_{4:7}$ denote the composition of abstraction functions $f_4 \cdot (\cdots) \cdot f_7$. Show that $(id \rightharpoonup \pi_1) \cdot \pi_1 \cdot f_{4:7}$ is the same as $\pi_1$.
□

## 12 On the transcription level

Our final calculations have to do with what [35] identify as the *transcription level*, the third ingredient of a *mapping scenario*. This has to do with diagram (9): once two pairs of data maps ("map forward" and "map backward") $F, R$ and $F', R'$ have been calculated, so as to represent two datatypes $A$ and $B$, they can be used to transcribe a given source operation $B \xleftarrow{O} A$ into some target operation $D \xleftarrow{P} C$.

How do we establish that $P$ *correctly* implements $O$? Intuitively, $P$ must be such that the performance of $O$ and that of $P$ (*wrapped* within the relevant abstraction and representation relations) cannot be distinguished:

$$O = F' \cdot P \cdot R \tag{157}$$

Equality is, however, much too strong a requirement. In fact, there is no disadvantage in letting the target side of (157) be more defined than the source operation $O$, provided both are simple [18]:

$$O \subseteq F' \cdot P \cdot R \tag{158}$$

Judicious use of (32, 33) will render (158) equivalent to

$$O \cdot F \subseteq F' \cdot P \tag{159}$$

provided $R$ is chosen maximal ($R = F^\circ$) and $F \preceq P$. This last requirement is obvious: $P$ must be prepared to cope with all possible representations delivered by $R = F^\circ$.

In particular, wherever the source operation $O$ is a *query*, ie. such that $F' = id$, (159) shrinks to

$$O \cdot F \subseteq P$$

In words: wherever the source query $O$ delivers a result $b$ for some input $a$, then the target query $P$ must deliver the same $b$ for any target value which represents $a$.

Suppose that, in the context of our running example (pedigree trees), one wishes to transcribe into SQL the query which fetches the name of the person whose pedigree tree is given. In the Haskell data model `PTree`, this is simply the (selector) function $name$. We want to investigate how this function gets mapped to lower levels of abstraction.

The interesting step is $\leq_2$, whereby trees are represented by pointers to heaps. The abstraction relation $Unf$ associated to this step is (co)inductive. Does this entail (co)inductive reasoning? Let us see. Focusing on this step alone, we want to solve equation

$$name \cdot Unf \subseteq Hname$$

[18] Staying within this class of operations is still quite general: it encompasses all deterministic, possibly partial computations. Moreover, within this class, inclusion coincides in fact with the standard definition of *operation refinement* [54].

for unknown $Hname$ — a query of type $((K \rightharpoonup \mathsf{G}\,K) \times K) \rightharpoonup Name$.

Simple relation currying (108) makes this equivalent to finding $Hname$ such that, for every heap $H$, $name \cdot (\overline{Unf}\ H) \subseteq \overline{Hname}\ H$ holds, that is, $name \cdot (unfold\ H) \subseteq \overline{Hname}\ H$. Since both $unfold\ H$ and $\overline{Hname}\ H$ are hylomorphisms, we write them in that way,

$$name \cdot [\![\ in, H\ ]\!] \subseteq [\![\ T, H\ ]\!]$$

so that $T$ becomes the unknown. Then we calculate:

$$\begin{array}{ll} & name \cdot [\![\ in, H\ ]\!] \subseteq [\![\ T, H\ ]\!] \\ \Leftarrow & \quad \{\ \text{fusion (152)}\ \} \\ & name \cdot in \subseteq T \cdot \mathsf{G}(name) \\ \equiv & \quad \{\ name \cdot Node = \pi_1 \cdot \pi_1\ \text{(147) ; expansion of } \mathsf{G}(name)\ \} \\ & \pi_1 \cdot \pi_1 \subseteq T \cdot (id \times (name + id) \times (name + id)) \\ \Leftarrow & \quad \{\ \pi_1 \cdot (f \times g) = f \cdot \pi_1\ \} \\ & T = \pi_1 \cdot \pi_1 \end{array}$$

Thus

$$\begin{array}{lll} \overline{Hname}\ H = [\![\ \pi_1 \cdot \pi_1, H\ ]\!] & & \\ = & \{\ (151)\ \} & \\ & \langle \mu\ X\ ::\ \pi_1 \cdot \pi_1 \cdot (id \times (X + id) \times (Xid)) \cdot H \rangle & \\ = & \{\ \pi_1 \cdot (f \times g) = f \cdot \pi_1\ \} & \\ & \langle \mu\ X\ ::\ \pi_1 \cdot \pi_1 \cdot H \rangle & \\ = & \{\ \text{trivia}\ \} & \\ & \pi_1 \cdot \pi_1 \cdot H & \end{array}$$

Back to *uncurried* format and introducing variables, we get (the post-condition of) $Hname$

$$n\ Hname(H, k) \equiv k \in dom\ H\ \wedge\ n = \pi_1(\pi_1(H\ k))$$

which means what one would expect: pointer $k$ is dereferenced in $H$, whereby a selection of the $Ind$ field takes place, from which the name is finally selected (recall that $Ind = Name \times Birth$).

The exercise of mapping $Hname$ down to the SQL level (156) is similar but less interesting. It will lead us to

$$n\ Rname\ (M, N, k) = k \in dom\ M\ \wedge\ n = \pi_1(M\ k)$$

where $M$ and $K$ are the two relational tables which originated from the heap of step 2. Since $Rname$ can be encoded as

```
SELECT Name FROM M WHERE PID = k
```

under some obvious assumptions concerning the case in which $k$ cannot be found, we are done as far as transcribing $name$ is concerned.

The main ingredient of the exercise we've just completed is the use of fusion property (152). But perhaps it all was *much ado for little*: queries aren't very difficult to transcribe in general. The example we give below is far more eloquent and has to do with housekeeping. Suppose one wants to defragment the heap at level 2 via some reallocation of heap cells. Let $K \xleftarrow{f} K$ be the function chosen to *rename* cell addresses. Recalling (36), defragmentation is easy to model as projection:

$$defragment \ : \ (K \longrightarrow K) \longrightarrow (K \rightharpoonup \mathsf{G}\, K) \longrightarrow (K \rightharpoonup \mathsf{G}\, K)$$
$$defragment\ f\ H \stackrel{\text{def}}{=} (\mathsf{G}\, f) \cdot H \cdot f^{\circ}$$

The correctness of $defragment$ has two facets. First, $H \cdot f^{\circ}$ should remain simple; second, it has to preserve the information stored in $H$: *the pedigree tree recorded in the heap (and pointer) shouldn't change in consequence of a* $defragment$ *operation*. In symbols:

$$t\, Unf\ (defragment\ f\ H, f\ k) \equiv t\, Unf\ (H, k) \tag{160}$$

Let us check (160):

$$\begin{array}{ll}
 & t\, Unf(defragment\ f\ H, f\ k) \ \equiv \ t\, Unf(H, k) \\
\equiv & \quad \{ \text{ (155) ; (149) } \} \\
 & t\, [\![\, in, defragment\ f H \,]\!]\ (f\ k) \ \equiv \ t\, [\![\, in, H \,]\!]\ k \\
\equiv & \quad \{ \text{ go pointfree; definition of } defragment \ \} \\
 & [\![\, in, (\mathsf{G}\, f) \cdot H \cdot f^{\circ} \,]\!] \cdot f = [\![\, in, H \,]\!] \\
\Leftarrow & \quad \{ \text{ fusion property (153) } \} \\
 & (\mathsf{G}\, f) \cdot H \cdot f^{\circ} \cdot f = (\mathsf{G}\, f) \cdot H \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & H \cdot f^{\circ} \cdot f = H \\
\equiv & \quad \{ \text{ since } H \subseteq H \cdot f^{\circ} \cdot f \text{ always holds } \} \\
 & H \cdot f^{\circ} \cdot f \subseteq H
\end{array}$$

So, condition $H \cdot f^{\circ} \cdot f \subseteq H$ (with points:

$$k \in dom\ H \ \wedge \ f\ k = f\ k' \Rightarrow k' \in \ dom\ H \ \wedge \ H\ k = H\ k'$$

for all heap addresses $k, k'$) is sufficient for $defragment$ to preserve the information stored in the heap *and* its simplicity [19]. Of course, any injective $f$ will qualify for safe defragmentation, for every heap.

[19] In fact, $H \cdot f^{\circ} \cdot f \subseteq H$ ensures $H \cdot f^{\circ}$ simple, via (33) and monotonicity.

Some comments are on demand. First of all, and unlike what is common in data refinement involving recursive data structures (see eg. [20] for a comprehensive case study), our calculations above have dispensed with any kind of inductive or coinductive argument. (This fact alone should convince the reader of the advantages of the PF-transform in program reasoning.)

Second, the $defragment$ operation we've just reasoned about is a so-called *representation changer* [28]. These operations (which include garbage collection, etc) are important because they add to efficiency without disturbing the service delivered to the user. In the *mapping scenario* terminology of [35], these correspond to operations which transcribe backwards to the identity function, at source level.

Finally, a comment on CRUD operation transcription. Although CRUD operations in general can be arbitrarily complex, in the process of transcription they split into simpler and simpler middleware and dataware operations which, at the target (eg. database) level end up involving standard protocols for data access [35].

The ubiquity of *simplicity* in data modeling, as shown throughout this tutorial, invites one to pay special attention to the CRUD of this kind of relation. Reference [51] identifies some "design patterns" for simple relations. The one we've used throughout this tutorial is the *identity pattern*. For this pattern, one may provide a succinct specification of the four CRUD operations on a simple relation $M$ as follows:

- $Create(N)$: $M \mapsto N \dagger M$, where argument $N$ embodies the new entries to add to $M$. The use of the override operator $\dagger$ [31, 53] instead of union ($\cup$) ensures simplicity and prevents from writing over existing entries.
- $Read(a)$: deliver $b$ such that $b\ M\ a$, if any.
- $Update(f, \Phi)$: $M \mapsto M \dagger f \cdot M \cdot \Phi$. This is a selective update: the contents of every entry whose key is selected by $\Phi$ get updated by $f$; all the other remain unchanged.
- $Delete(\Phi)$: $M \mapsto M \cdot (id - \Phi)$, where $R - S$ means relational difference (25). All entries whose keys are selected by $\Phi$ are removed.

Space constraints preclude addressing this topic in this tutorial. Reference [51] shows the application of the PF-transform to speed-up reasoning about CRUD preservation of datatype invariants on simple relations. Similar gains are expected in doing the same in the context of CRUD transcription in general.

## 13 2LT: a two-level data transformation toolset

The data transformation calculus which is the subject of this tutorial is currently (partly) animated by the 2LT package of the U.Minho Haskell libraries [15, 9, 16] via (typed) strategic term re-writing using GADTs. More recently, the relational calculus itself has also been addressed in the same way and shown to be applicable to extended static checking [43] in a model-driven engineering context.

At the tutorial we shall review these (rather recent) developments and show how they can be used to scale-up the data transformation/mapping techniques presented to real-size case-studies, mainly by mechanizing repetitive tasks and discharging the (otherwise unmanageable by pen-and-paper) housekeeping.

## 14 Conclusions and future work

This tutorial describes a mathematical approach to the study of data transformation. As main advantages of the approach we point out: (a) a unified and powerful notation to describe data-structures across various programming paradigms, and its (b) associated calculus based on elegant rules which are reminiscent of school algebra; (c) the fact that data impedance mismatch is easily expressed by rules of the calculus which, by construction, offer type-level transformations *together with* well-typed data mappings; (d) the properties enjoyed by such rules, which enable their application in a stepwise, structured way.

The novelty of this approach when compared to previous attempts to lay down the same theory is the use of binary relations to express *both* algorithms and data, in a way which dispenses with inductive proofs and cumbersome reasoning. In fact, most work on the pointfree relation calculus has so far been focused on reasoning about programs (ie. algorithms). Advantages of our proposal to use it *uniformly* for programs *and data* are already apparent at practical level, see eg. on-going work reported in [43].

This style of calculation has been offered to Minho students for several years (in the context of the local tradition on formal modeling) as alternative to standard database design techniques. It is the foundation of the "2LT bundle" of tools available from the UMinho Haskell libraries. However, there is still much work to be done. Below we list a number of items which we propose as prompt topics for research.

*Lenses.* The similarity between abstraction/representation pairs implicit in $\leq$-rules and bi-directional transformations known as *lenses* [27, 12] (developed in the context of the classical *view-update* problem) calls for a PF-calculational approach to the latter. Each lens connects a concrete representation $C$ with an abstract view $A$ on it by means of two functions $A \times C \xrightarrow{put} C$ and $A \xleftarrow{get} C$. (Note the similarity with $(R, F)$-pairs, except for $put$'s additional argument of type $C$.) The possibility of having to extend $put$ and $get$ to relations is open.

*Heaps and pointers at target.* We believe that Jourdan's long, inductive pointwise argument [32] for $\leq$-law (155) can be supplanted by succint pointfree calculation if results developed meanwhile by Gibbons [23] are taken into account. Moreover, the same law should be put in parallel with other related work on calculating with pointers (read eg. [11] and follow the references).

*Separation logic.* Law (155) has a clear connection to shared-mutable data representation and thus with *separation logic* [57]. We are already working on a PF-relational model for this logic [59], which we believe will be useful in better studying and further extending law (155).

*Concrete invariants.* Taking concrete invariants into account is useful because these ensure (for free) properties at target-data level which can be advantageous in the transcription of source operations. The techniques presented in section 8 and further exploited in [58] need further work. Moreover, $\leq$-rules should be able to take invariants into account (a topic suggested but little developed in [49]).

*Mapping scenarios for the UML.* Following the exercise of section 9, a calculational theory of UML mapping scenarios could be developed starting from eg. K. Lano's catalogue [37]. For preliminary work on this subject see eg. [8].

## References


1. Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes, December 1992. Available from `www.cs.nott.ac.uk/~rcb/papers`.
2. T.L. Alves, P.F. Silva, J. Visser, and J.N. Oliveira. Strategic term rewriting and its application to a *VDM-SL* to SQL conversion. In FM 2005*:* 13th International Symposium on Formal Methods, volume 3582 of *Lecture Notes in Computer Science*, pages 399–414. Springer-Verlag, 2005. University of Newcastle upon Tyne, United Kingdom - July 18-22.
3. K. Backhouse and R.C. Backhouse. Safety of abstract interpretations for free, via logical relations and Galois connections. *Science of Computer Programming*, 15(1–2):153–196, 2004.
4. R.C. Backhouse. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.
5. R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In *2nd Int. Conf. Algebraic Methodology and Software Technology (AMAST'91)*, pages 303–362. Springer LNCS, 1992.
6. 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.
7. R. Barker. *CASE*METHOD — Entity Relationship Modelling*. Addison-Wesley Publishing Company, Great Britain, 1992.
8. P. Berdaguer. Algebraic representation of UML class-diagrams, May 2007. Dept. Informatics of the U.Minho, Technical note.
9. Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled Schema Transformation and Data Conversion For XML and SQL. In *PADL 2007*, volume 4354 of *LNCS*, pages 290–304. Springer-Verlag, 2007.
10. R. Bird and O. de Moor. Algebra of Programming. Series in Computer Science. Prentice-Hall International, 1997. C.A. R. Hoare, series editor.
11. Richard S. Bird. Unfolding pointer algorithms. *J. Funct. Program.*, 11(3):347–358, 2001.
12. Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006.
13. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, Inc., 1999. ISBN 0-201-57168-4.
14. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.
15. Alcino Cunha, J.N. Oliveira, and Joost Visser. Type-safe two-level data transformation. In FM'06 , volume 4085 of *LNCS*, pages 284–289. Springer-Verlag, Aug. 2006.
16. Alcino Cunha and Joost Visser. Transformation of structure-shy programs, applied to XPath queries and strategic functions, 2007. In Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2007 (to appear).
17. J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.
18. W.-P. de Roever, K. Engelhardt with the assistance of J. Coenen, K.-H. Buth, P. Gardiner, Y. Lakhnech, and F. Stomp. *Data Refinement Model-Oriented Proof methods and their Comparison*. Cambridge University Press, 1999. ISBN 0521641705.

19. M. Deutsch, M. Henson, and S. Reeves. Modular reasoning in Z: scrutinising monotonicity and refinement, 2006. (To appear).
20. E. Fielding. The specification of abstract mappings and their implementation as $B^+$-trees. Technical Report PRG-18, Oxford University, September 1980.
21. J. Fitzgerald and P.G. Larsen. Modelling Systems: Practical Tools and Techniques for Software Development . Cambridge University Press, 1st edition, 1998.
22. R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19, pages 19–32. American Mathematical Society, 1967. Proc. Symposia in Applied Mathematics.
23. Jeremy Gibbons. When is a function a fold or an unfold?, 2003. Working document 833 FAV-12 available from the website of IFIP Working Group 2.1 57th meeting, New York City, USA.
24. Jean-Luc Hainaut. The transformational approach to database engineering. In Lämmel et al. [36], pages 95–143.
25. Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP'86*, volume 213 of *LNCS*, pages 187–196, 1986.
26. Paul Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, University of Eindhoven, The Netherlands, 1997.
27. Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proc. ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189. ACM Press, 2004.
28. Graham Hutton and Erik Meijer. Back to basics: Deriving representation changers functionally. *Journal of Functional Programming*, 1993. (Functional Pearl).
29. Graham Hutton and Joel Wright. Compiling exceptions correctly. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *LNCS*, pages 211–227. Springer, 2004.
30. J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in Lecture Notes in Computer Science. Springer, 1996.
31. C.B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall International, 1990. 1st edition (1986), 2nd edition (1990). PDF available from `http://www.vdmportal.org/twiki/bin/view/Main/Jonesbook`.
32. I.S. Jourdan. Reificação de tipos abstractos de dados: Uma abordagem matemática. Master's thesis, University of Coimbra, 1992. (In Portuguese).
33. Wolfram Kahl. Refinement and development of programs from relational specifications. *ENTCS*, 44(3):4.1–4.43, 2003.
34. E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, Inc., 6th edition, 1988.
35. Ralf Lämmel and Erik Meijer. Mappings make data processing go 'round. In Lämmel et al. [36], pages 169–218.
36. Ralf Lämmel, João Saraiva, and Joost Visser, editors. *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, volume 4143 of *Lecture Notes in Computer Science*. Springer, 2006.
37. K. Lano. Catalogue of model transformations. No date. Available from `http://www.dcs.kcl.ac.uk/staff/kcl/`.
38. D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN 0-914894-42-0.
39. 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.

40. C. McLarty. *Elementary Categories, Elementary Toposes*. Oxford Logic Guides nr.21. Calendron Press, Oxford, 1st edition, 1995.
41. Sun Meng and L.S. Barbosa. On refinement of generic state-based software components. In C. Rettray, S. Maharaj, and C. Shankland, editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling, August 2004. Springer Lect. Notes Comp. Sci. (3116). Best student co-authored paper award.
42. C. Morgan. *Programming from Specification*. Series in Computer Science. Prentice-Hall International, 1990. C.A. R. Hoare, series editor.
43. Claudia Necco, J.N. Oliveira, and Joost Visser. Extended static checking by strategic rewriting of pointfree relational expressions, 2007. DIUM Technical Report.
44. J.N. Oliveira. Refinamento transformacional de especificações (terminais). In *Actas das XII* Jornadas Luso-Espanholas de Matemática, volume II, pages 412–417, Maio 1987.
45. J.N. Oliveira. *A Reification Calculus for Model-Oriented Software Specification* . Formal Aspects of Computing, 2(1):1–23, April 1990.
46. J.N. Oliveira. *Software Reification using the SETS Calculus* . In Tim Denvir, Cliff B. Jones, and Roger C. Shaw, editors, *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. ISBN 0387197524, Springer-Verlag, 8–10 January 1992. (Invited paper).
47. J.N. Oliveira. *'Fractal' Types: an Attempt to Generalize Hash Table Calculation*. In *Workshop on Generic Programming (*WGP'98*), Marstrand, Sweden*, June 1998.
48. J.N. Oliveira. Data processing by calculation, 2001. 108 pages. Lecture Notes of course lectured at the *6th Estonian Winter School in Computer Science*, 4-9 March 2001, Palmse, Estonia.
49. J.N. Oliveira. Constrained datatypes, invariants and business rules: a relational approach, 2004. PUReCafé, DI-UM, 2004.5.20 [talk], PURE PROJECT (POSI/CHS/44304/2002).
50. J.N. Oliveira. *Calculate databases with 'simplicity'*, September 2004. Presentation at the *IFIP WG 2.1 #59 Meeting*, Nottingham, UK. (Slides available from the author's website.).
51. J.N. Oliveira. *Reinvigorating pen-and-paper proofs in VDM: the pointfree approach* , 2006. Presentation at the *Third* OVERTURE *Workshop: Newcastle, UK, 27-28 November 2006* . Available from the author's website.
52. J.N. Oliveira. Pointfree foundations for lossless decomposition, 2007. Draft of paper in preparation.
53. J.N. Oliveira and C.J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In MPC'04 *: Seventh International Conference on Mathematics of Program Construction, 12-14 July, 2004, Stirling, Scotland, UK (Organized in conjunction with AMAST'04)*, volume 3125 of *Lecture Notes in Computer Science*, pages 334–356. Springer, 2004.
54. J.N. Oliveira and C.J. Rodrigues. Pointfree factorization of operation refinement. In FM'06 , volume 4085 of *LNCS*, pages 236–251. Springer-Verlag, 2006.
55. M.S. Paterson and C.E. Hewitt. Comparative schematology. In *Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–127, August 1970.
56. V. Pratt. Origins of the calculus of binary relations. In *Proc. of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 248–254, Santa Cruz, CA, 1992. IEEE Computer Soc.
57. J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002. Invited Paper, LICS'02.
58. C.J. Rodrigues. *Software Refinement by Calculation*. PhD thesis, Departamento de Informática, Universidade do Minho, 2007. (Submitted.).
59. Wang Shuling, L.S. Barbosa, and J.N. Oliveira. A pointfree relational model for confined separation logic, June 2007. Technical report, DI/UM.
60. Joost Visser. *Generic Traversal over Typed Source Code Representations*. Ph. D. dissertation, University of Amsterdam, Amsterdam, The Netherlands, 2003.

61. Eric G. Wagner. All recursive types defined using products and sums can be implemented using pointers. In Clifford Bergman, Roger D. Maddux, and Don Pigozzi, editors, *Algebraic Logic and Universal Algebra in Computer Science*, volume 425 of *LNCS*. Springer, 1990.
62. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

## A PTree example in Haskell

This annex presents the exercise, in Haskell, of representing inductive type `PTree` (45) by pointers and heaps. For simplicity, the datatype of `PTree`-shaped heaps is modeled by finite lists of pairs, together with a pointer telling where to start from:

```
data Heap a k = Heap [(k,(a,Maybe k, Maybe k))] k
```

Next, we convert this into a bifunctor [20]:

```
instance BiFunctor Heap
   where bmap g f
         (Heap h k') =
              Heap [ (f k) |-> (g a, fmap f p, fmap f p')
                           | (k,(a,p,p')) <- h ]
                   (f k')
```

The chosen (functional) representation is a *fold* over `PTree`,

```
r (Node n b m f) = let x = fmap r m
                       y = fmap r f
                   in merge (n,b) x y
```

where `merge` is the interesting function:

```
merge a (Just x) (Just y) =
        Heap ([ 1 |-> (a, Just k1, Just k2) ] ++ h1 ++ h2) 1
                  where (Heap h1 k1) = bmap id even_ x
                        (Heap h2 k2) = bmap id odd_ y
merge a Nothing Nothing  =
        Heap ([ 1 |-> (a, Nothing, Nothing) ]) 1
merge a Nothing (Just x) =
        Heap ([ 1 |-> (a, Nothing, Just k2) ] ++ h2) 1
                  where (Heap h2 k2) = bmap id odd_ x
merge a (Just x) Nothing =
        Heap ([ 1 |-> (a, Just k1, Nothing) ] ++ h1) 1
                  where (Heap h1 k1) = bmap id even_ x
```

Note the use of two functions

```
even_ k = 2*k
odd_  k = 2*k+1
```

[20] Note the sugaring of pairing in terms of the infix combinator `x |-> y = (x,y)`, as suggested by (36). Class `BiFunctor` is the binary extension to standard class `Functor` offering `bmap :: (a -> b) -> (c -> d) -> (f a c -> f b d)`, the binary counterpart of `fmap`.

which generate the $k$th even and odd numbers. Functorial renaming of heap addresses via these functions (whose ranges are disjoint) ensure that the heaps we are joining (via list concatenation) are *separate* [57, 59] [21].

This representation strategy can be generalized to any polynomial type of degree $n$ by building $n$-functions $f_i\ k = nk + i$, for $0 \leq k < n$.

[21] This representation technique is reminiscent of that of storing "binary heaps" (which are not quite the same as in this tutorial) as arrays without pointers, see eg. entry `Binary_heap` in the Wikipedia.