Transforming Data by Calculation

J.N. Oliveira

Ref. [Ol08a] — 2008

J.N. Oliveira. *Transforming Data by Calculation.* In *GTTSE 2007*, volume 5235 of *LNCS*, pages 134–192, 2008.

Transforming Data by Calculation

José N. Oliveira

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

Abstract. This paper addresses the foundations of data-model transformation. A catalog of *data mappings* is presented which includes abstraction and representation relations and associated constraints. These are justified in an algebraic style via the *pointfree-transform*, a technique whereby predicates are lifted to binary relation terms (of the algebra of programming) in a two-level style encompassing both data and operations. This approach to data calculation, which also includes transformation of recursive data models into "flat" database schemes, is offered as alternative to standard database design from abstract models. The calculus is also used to establish a link between the proposed transformational style and bidirectional *lenses* developed in the context of the classical *view-update problem*.

Keywords: Theoretical foundations, mapping scenarios, transformational design, refinement by calculation.

1 Introduction

Watch yourself using a pocket calculator: every time a digit key is pressed, the corresponding digit is displayed on the LCD display once understood by the calculator, a process which includes representing it internally in binary format:

$$\begin{tikzcd}
digits \arrow[d, bend left, "input"] \\
binary \arrow[u, bend left, "display"]
\end{tikzcd}$$

This illustrates the main ingredients of one's everyday interaction with machines: the abstract objects one has in mind (eg. digits, numbers, etc) need to be *represented* inside the machine before this can perform useful calculations, eg. square root, as displayed in the diagram below.

$$\begin{tikzcd}
digits & digits \arrow[d, bend left, "input"] \\
binary \arrow[u, bend left, "display"] & binary \arrow[l, bend left, "\sqrt{}"]
\end{tikzcd}$$

However, it may happen that our calculator is faulty. For instance, sometimes the digit displayed is not the one whose key was just pressed; or *nothing* at all is displayed; or even the required operation (such as triggered by the square root key) is not properly computed. It is the designer's responsibility to ensure that the machine we are using never misbehaves and can thus be trusted.

R. Lämmel, J. Visser, and J. Saraiva (Eds.): GTTSE 2007, LNCS 5235, pp. 134–195, 2008.

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 on 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 $input/display$ 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 $represent/retrieve$ contracts 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 computation. The first two have to do with *data representation* and the third with *data processing*. Helping in preventing any of these from happening in software designs is the main aim of this paper.

We will see that most of the work has to do with *data transformation*, a technique which the average programmer is often unaware of using when writing, most often in an 'ad hoc' way, 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.

Structure of the paper. This paper is organized as follows. Section 2 presents the overall spirit of the approach and introduces a simple running example. Section 3 reviews the binary relation notation and calculus, referred to as the *pointfree (PF) transform*. Section 4 shows how to denote the meaning of data in terms of such unified notation. Section 5 expresses data impedance mismatch in the PF-style. While sections 6 to 8 illustrate the approach in the context of (database) relational modeling, recursive data modeling is addressed from section 9 onwards. Then we show how to handle cross-paradigm impedance by calculation (section 10) and how to transcribe operations from recursive to flat data models (section 11). Section 12 addresses related work. In particular, it establishes a link between data mappings and bidirectional *lenses* developed in the context of the *view-update problem* and reviews work on a library for data transformations (2LT) which is strongly related to the current paper. Finally, section 13 concludes and points out a number of research directions in the field.

Technical sketch of the paper. This text puts informal, technology dependent approaches to data transformation together with data calculation formalisms which are technology agnostic. It is useful to anticipate how such schools of thought are related along the paper, while pinpointing the key formal concepts involved.

The main motivation for data calculation is the need for *data-mappings* as introduced in section 2: one needs to ensure that data flow unharmed across the boundaries of software layers which use different technologies and/or adopt different data models. On the technical side, this is handled (in section 2) by ordering data formats by degree of abstraction and writing $A \leq B$ wherever format A is safely implemented by format B. Technically, $\leq$ is a *preorder* and $\leq$-facts are witnessed by *relations* telling how data should flow back and forth between formats A and B.

The need for handling such relations in a compositional, calculational way leads to the relational calculus and the pointfree transform. The whole of section 3 is devoted to providing a summary of the required background, whose essence lies in a number of laws which can be used to calculate with relations directly (instead of using set theory to indirectly convey the same results). The fact that all relations are binary is not a handicap: they can be thought of as arrows of the form $A \xrightarrow{R} B$ which express data flow in a natural way and can be composed with each other to express more complex data flows. Data filtering is captured by relations of a particular kind, known as *coreflexives*, which play a prominent role throughout the whole calculus.

The bridge between formal and informal data structuring becomes more apparent from section 4 onwards, where typical data structures are shown to be expressible not only in terms of abstract constructs such as Cartesian product ($A \times B$), disjoint sum ($A + B$) and equations thereof (as in the case of recursive types), but also in terms of typed finite relations, thus formalizing the way data models are recorded by entity-relationship diagrams or UML class diagrams, for instance.

Further to structure, *constraints* (also known as *invariants*) are essential to data modeling, making it possible to enforce semantic properties on data. Central to such data constraints is *membership*, a relation of type $A \xleftarrow{\in} \mathsf{T}A$ which is able to tell which data elements can be found in a particular data structure of shape T. The key ingredient at this point is the fact that set-theoretic membership can be extended to data containers other than sets.

Sections 5 and 6 are central to the whole paper: they show how to calculate complex data mappings by combining a number of $\leq$-rules which are proposed and justified using (pointfree) relation calculus. Compositionality is achieved in two ways: by transitivity, suitably typed $\leq$-rules can be chained; by monotonicity, they can be promoted from the parameters of a parametric type T to the whole type, for instance by inferring $\mathsf{T}A \leq \mathsf{T}B$ from $A \leq B$. The key of the latter result consists in regarding T as a *relator*, a concept which traverses relation calculus from beginning to end and explains, in the current paper, data representation techniques such as those involving dynamic heaps and pointer dereferencing. On the practical side, a number of $\leq$-facts are shown to be applicable to calculating database schemata from abstract models (sections 6 and 7) and reasoning about entity-relationship diagrams (section 8).

Abstract (and language-based) data models often involve recursive data which pose challenges of their own to data mapping formalization. Sections 9 to 11 show how the calculus of fixpoint solutions to relational equations (known as *hylomorphisms*) offers a basis for refining recursive data structures. This framework is set to work in section 10 where it is applied to the paper's running example, the `PTree` recursive model of pedigree trees, which is eventually mapped onto a flat, non-recursive model, after stepping through a pointer-based representation. The layout of calculations not only captures the $\leq$ relationships among source, intermediate and target data models, but also the abstraction and representation relations implicit in each step, which altogether synthesize two overall 'map forward" and "map backward" data transformations.

Section 11 addresses the *transcription level*, the third component of a *mapping scenario*. This has to do with refining operations whose input and output data formats have changed according to such big-step 'map forward" and "map backward" transformations. Technically, this can be framed into the discipline of data refinement. The examples given, which range from transcribing a query over `PTree` downto the level of its flat version (obtained in section 10) to calculating low level operations handling heaps and pointers, show once again the power of data calculation performed relationally, and in particular the usefulness of so-called *fusion*-properties.

Finally, section 12 includes a sketch of how $\leq$-diagrams can be used to capture bidirectional (asymmetric) transformations known as *lenses* and their properties.

2 Context and Motivation

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 $c\ R\ a$ to denote the fact that *datum* c *represents datum* a (assuming that a and c range over two given data types A and C, respectively) and the converse fact $a\ R^{\circ}\ c$ to denote that a *is the datum represented by* c. The use of definite article "*the*" instead of "*a*" in the previous sentence is already a symptom of the **no confusion** principle — we want c to represent *only one* datum of interest:

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

The **no loss** principle means that no data are lost in the representation process. Put in other words, it ensures that every datum of interest a is representable by some c:

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

Above we mention the converse R° of R, which is the relation such that $a(R^{\circ})c$ holds iff $c\ R\ a$ holds. Let us use this rule in re-writing (1) in terms of $F = R^{\circ}$:

$$\langle \forall\, c, a, a' \ :: \ a\ F\ c \ \wedge \ a'\ F\ c \Rightarrow a = a' \rangle \tag{3}$$

This means that F, the converse of R, can be thought of as an *abstraction relation* which is *functional* (or deterministic): two outputs a, a' for the same input c are bound to be the same.

Before going further, note the notation convention of writing the outputs of F on the left hand side and its inputs on the right hand side, as suggested by the usual way of declaring functions in ordinary mathematics, $y = f\ x$, where y ranges over outputs (cf. the vertical axis of the Cartesian plane) and x over inputs (cf. the other, horizontal axis). This convention is adopted consistently throughout this text and is extended to relations, as already seen above [1].

[1] The fact that $a\ F\ c$ is written instead of $a = F\ c$ reflects the fact that F is not a total function, in general. See more details about notation and terminology in section 3.

Expressed in terms of F, (2) becomes

$$\langle \forall\, a \;::\; \langle \exists\, c \;::\; a\, F\, c \rangle \rangle \tag{4}$$

meaning that F is *surjective*: every abstract datum a is reachable by F. In general, it is useful to let the abstraction relation F to be larger that R°, provided that it keeps properties (3,4) — being functional and surjective, respectively — and that it stays *connected* to R. This last property is written as

$$\langle \forall\, a, c \;::\; c\, R\, a \Rightarrow a\, F\, c \rangle$$

or, with less symbols, as

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

by application of the rule which expresses relational inclusion:

$$R \subseteq S \;\;\equiv\;\; \langle \forall\, b, a \;::\; b\, R\, a \Rightarrow b\, S\, a \rangle \tag{6}$$

(Read $R \subseteq S$ as "R is at most S", meaning that S is either more defined or less deterministic than R.)

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

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "F"] \end{tikzcd} \tag{7}$$

where A is the datatype of data *to be represented* and C is the chosen datatype of representations [2]. 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 referred to as the *abstraction relation* in a (R, F) pair.

Layered representation. In general, it will make sense to chain several layers of abstraction as in, for instance,

$$\begin{tikzcd} I \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & M \arrow[l, bend left, "F"] \arrow[r, bend left, "R'"] \arrow[r, phantom, "\leq"] & D \arrow[l, bend left, "F'"] \end{tikzcd} \tag{8}$$

where letters I, M and D have been judiciously chosen so as to suggest the words *interface*, *middleware* and *dataware*, respectively.

[2] Diagrams such as (7) should not be confused with commutative diagrams expressing properties of the relational calculus, as in eg. [11], since the ordering $\leq$ in the diagram is an ordering on objects and not on arrows.

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 diagram (8) above is given in figure 1, where layers I, M and D are represented by concentric circles.

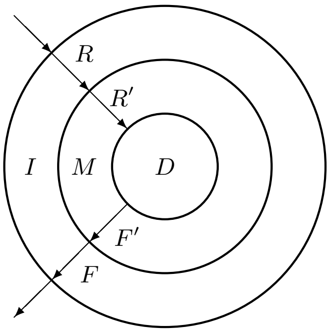
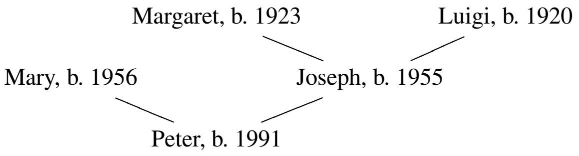


Fig. 1. Layered software architecture

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



(9)

Trees — which are the users' mental model of recursive structures — become pointer structures (figure 2a) 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 of database files (cf. figure 2b).

Modeling pedigree trees will be our main running example throughout this paper.

Mapping scenarios. Once materialized in some technology (eg. XML, C/C++/Java, SQL, etc), the layers of figure 1 stay apart from each other in different programming *paradigms* (eg. markup languages, object-orientated databases, relational databases, etc) each requiring its own skills and programming techniques.

As shown 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* [3], in the words of reference [42] where a thorough account is given of the many problems which hinder software technology in this respect. Still quoting [42]:

> *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. (...)*

[3] According to [3], the label *impedance mismatch* was coined in the early 1990's to capture (by analogy with a similar situation in electrical circuits) the technical gap between the object and relational technologies. Other kinds of impedance mismatch are addressed in [42, 67].

> *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.*

The same reference 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, diagram (7) can be seen as a succinct presentation of the two first ingredients, the former being captured by the $\leq$-ordering on data models and the latter by the (R, F) pair of relations. The third can easily be captured by putting two instances of (7) 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:

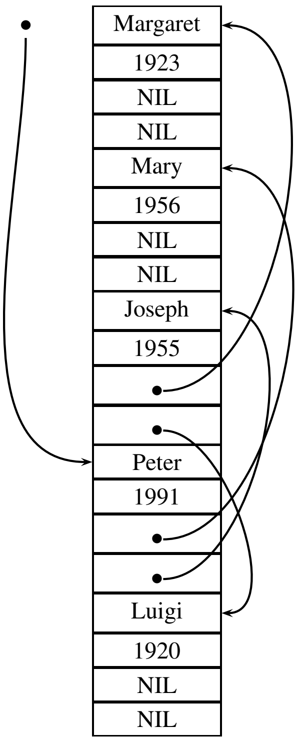
$$\begin{tikzcd}
A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] \arrow[d, "O"'] & C \arrow[l, bend left, "F"] \arrow[d, "P"] \\
B \arrow[r, bend left, "R'"] \arrow[r, phantom, "\leq"] & D \arrow[l, bend left, "F'"]
\end{tikzcd} \tag{10}$$

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 B are the same (and so are C and D), and O is regarded as a state-transforming operation of 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.

Data refinement. The theory behind diagrams such as (10) is known as *data refinement*. It is among the most studied formalisms in software design theory and is available from several textbooks — see eg. [20, 38, 49].

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 far too complex a discipline for most software practitioners, a fact which is mirrored on its prolific terminology — cf. *downward*, *upward* refinement [31], *forwards*, *backwards* refinement [31, 48, 70], *S,SP,SC*-refinement [21] and so on. Another weakness 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 paper 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"?



(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. 2. Middleware (a) and dataware (b) formats for family tree sample data (9)

Calculational techniques. 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 [46] and Floyd's work on program meaning [26]. A so-called *program transformation* school was already active in the mid 1970s, see for instance references [16, 19]. But program transformation becomes *calculational* only after the inspiring work of J. Backus in his *algebra of (functional) programs* [7] where the emphasis is put on the calculus of functional combinators rather than on the λ-notation and its variables, or *points*. This is why Backus' calculus is said to be *point-free*.

Intensive research on the (pointfree) program calculation approach in the last thirty years has led to the *algebra of programming* discipline [5, 11]. 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 [51, 52, 53] where the $\leq$-ordering and associated rules are defined. The approach, however, was not agile enough. It is only after its foundations are stated in the pointfree style [54, 56] that succinct calculations can be performed to derive data representations.

Summary. We have thus far introduced the topic of data representation framed in two contexts, one practical (data mapping scenarios) and the other theoretical (data refinement). In the remainder of the paper the reader will be provided with strategies and tools for handling mapping scenarios by calculation. This is preceded by the section which follows, which settles basic notation conventions and provides a brief overview of the binary relational calculus and the pointfree-transform, which is essential to understanding data calculations to follow. Textbook [11] is recommended as further reading.

3 Introducing the Pointfree Transform

By *pointfree transform* [60] ("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 [61], eventually led to what is known today as the *algebra of programming* [5, 11]. As suggested in [60], the PF-transform offers to the predicate calculus what the Laplace transform [41] offers to the differential/integral calculus: the possibility of changing the underlying mathematical space in a way which enables agile algebraic calculation.

Theories "refactored" via the PF-transform become more general, more structured and simpler [58, 59, 60]. 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 [5, 11] until proofs are discharged or solutions are found for the original problem statements, which are mapped back to logics if required.

Relations. Let arrow $B \xleftarrow{R} A$ 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. Type declarations $B \xleftarrow{R} A$ and $A \xrightarrow{R} B$ will mean the same.

$R \cup S$ (resp. $R \cap S$) denotes the union (resp. intersection) of two relations R and S. $\top$ is the largest relation of its type. Its dual is $\bot$, the smallest such relation (the empty one). Two other operators are central to the relational calculus: composition ($R \cdot S$) and converse (R°). The latter has already been introduced in section 2. 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 one of the kernel rules of the PF-transform:

$$b(R \cdot S)c \ \equiv \ \langle \exists\, a \ :: \ bRa \ \wedge \ aSc \rangle \tag{11}$$

Note that converse is an involution

$$(R^\circ)^\circ = R \tag{12}$$

and commutes with composition:

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

All these relational operators are $\subseteq$-monotonic, where $\subseteq$ is the inclusion partial order (6). 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.) Relational equality can be established by $\subseteq$-antisymmetry:

$$R = S \;\equiv\; R \subseteq S \,\wedge\, S \subseteq R \tag{14}$$

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. They are denoted by uppercase Greek letters (eg. Φ, Ψ) and obey a number of interesting properties, among which we single out the following, which prove very useful in calculations:

$$\Phi \cdot \Psi \;=\; \Phi \cap \Psi \;=\; \Psi \cdot \Phi \tag{15}$$

$$\Phi^\circ = \Phi \tag{16}$$

The PF-transform of a (unary) *predicate* p is the coreflexive Φ_p such that

$$b\,\Phi_p\,a \equiv (b = a) \;\wedge\; (p\,a)$$

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 $\Phi_{\lambda a . a \in S}$, that is, $b\,\Phi_S\,a$ means $(b = a) \;\wedge\; a \in S$.

Preorders are reflexive and 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, $\mathsf{ker}\,R \overset{\mathrm{def}}{=} R^\circ \cdot R$ and its dual, $\mathsf{img}\,R \overset{\mathrm{def}}{=} R \cdot R^\circ$, the *image* of R [4]. From (12, 13) one immediately draws

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

$$\mathsf{img}\,(R^\circ) = \mathsf{ker}\,R \tag{18}$$

Kernel and image lead to the following terminology:

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

(19)

[4] As explained later on, these operators are relational extensions of two concepts familiar from set theory: the image of a function f, which corresponds to the set of all y such that $\langle \exists\, x \;::\; y = f\,x \rangle$, and the kernel of f, which is the equivalence relation $b(\mathsf{ker}\,f)a \;\equiv\; f\,b \;=\; f\,a$. (See exercise 3.)

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

Recall that part of this terminology has already been mentioned in section 2. In this context, let us check formula (1) against the definitions captured by (19) as warming-up exercise in pointfree-to-pointwise conversion:

$$\begin{array}{ll}
 & \langle \forall\, c, a, a' \;::\; c\, R\, a \;\wedge\; c\, R\, a' \Rightarrow a = a' \rangle \\
\equiv & \qquad \{ \text{ rules of quantification [5] and converse } \} \\
 & \langle \forall\, a, a' \;:\; \langle \exists\, c \;::\; a\, R^\circ\, c \;\wedge\; c\, R\, a' \rangle \;:\; a = a' \rangle \\
\equiv & \qquad \{ \text{ (11) and rules of quantification } \} \\
 & \langle \forall\, a, a' \;::\; a (R^\circ \cdot R) a' \Rightarrow a = a' \rangle \\
\equiv & \qquad \{ \text{ (6) and definition of kernel } \} \\
 & \mathsf{ker}\, R \subseteq id
\end{array}$$

Exercise 1. Derive (2) from (19). □

Exercise 2. Resort to (17,18) and (19) 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) □

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, ϕ) or identifiers starting with lowercase characters. Function application will be denoted by juxtaposition, eg. $f\ a$ instead of $f(a)$. Thus bfa means the same as $b = f\ a$.

The overall taxonomy of binary relations is pictured in figure 3 where, further to the standard classes, we add *representations* and *abstractions*. As seen already, these are the relation classes involved in $\leq$-rules (7). 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 bijection 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 [11]. For instance, the PF-transform rule which follows, involving two functions (f, g) and an arbitrary relation R

$$b(f^\circ \cdot R \cdot g)a \equiv (f\ b)R(g\ a) \qquad (20)$$

plays a prominent role in the PF-transform [4]. The pointwise definition of the kernel of a function f, for example,

$$b(\mathsf{ker}\, f)a \;\;\equiv\;\; f\, b \;=\; f\, a \qquad (21)$$

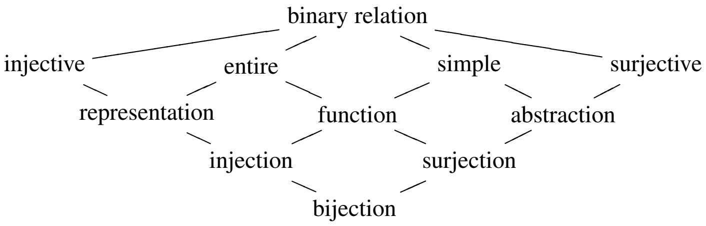


Fig. 3. Binary relation taxonomy

stems from (20), 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 data domain.)

Exercise 3. Given a function $B \xleftarrow{f} A$, calculate the pointwise version (21) of $\mathsf{ker}\, f$ and show that $\mathsf{img}\, f$ is the coreflexive associated to predicate $p\ b = \langle \exists\ a\ ::\ b = f\ a \rangle$. □

Given two preorders $\leq$ and $\sqsubseteq$, one may relate arguments and results of pairs of suitably typed functions f and g in a particular way,

$$f^\circ \cdot \sqsubseteq \;\; = \;\; \leq \cdot\, g \tag{22}$$

in which case both f, g are monotone and said to be *Galois connected*. Function f (resp. g) is referred to as the *lower* (resp. *upper*) adjoint of the connection. By introducing variables in both sides of (22) via (20), we obtain, for all a and b

$$(f\ b) \sqsubseteq a \;\; \equiv \;\; b \leq (g\ a) \tag{23}$$

Quite often, the two adjoints are *sections* of binary operators. Given a binary operator θ, its two sections $(a\theta)$ and (θb) are unary functions f and g such that, respectively:

$$f = (a\theta) \;\; \equiv \;\; f\ b = a\,\theta\,b \tag{24}$$

$$g = (\theta b) \;\; \equiv \;\; g\ a = a\,\theta\,b \tag{25}$$

Galois connections in which the two preorders are relation inclusion ($\leq, \sqsubseteq := \subseteq, \subseteq$) and whose adjoints are sections of relational combinators are particularly interesting because they express universal properties about such combinators. Table 1 lists connections which are relevant for this paper.

It is easy to recover known properties of the relation calculus from table 1. 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 in table 1, namely

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

Table 1. Sample of Galois connections in the relational calculus. The general formula given on top is a logical equivalence universally quantified on S and R. It has a left part involving lower adjoint f and a right part involving upper adjoint g.

$$(f\ R) \subseteq S \equiv R \subseteq (g\ S)$$

Description	f	g	**Obs.**
converse	$(\_)^\circ$	$(\_)^\circ$	
shunting rule	$(h\cdot)$	$(h^\circ\cdot)$	h is a function
"converse" *shunting* rule	$(\cdot h^\circ)$	$(\cdot h)$	h is a function
domain	δ	$(\top\cdot)$	left $\subseteq$ restricted to coreflexives
range	ρ	$(\cdot\top)$	left $\subseteq$ restricted to coreflexives
difference	$(\_ - R)$	$(R \cup\ )$	

These equivalences are popularly known as "shunting rules" [11]. 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 (see eg. [11]).

It should be mentioned that some rules in table 1 appear in the literature under different guises and usually not identified as GCs [5]. For a thorough presentation of the relational calculus in terms of GCs see [1, 5]. There are *many* advantages in such an approach: further to the systematic tabulation of operators (of which table 1 is just a sample), GCs have a rich algebra of properties, namely:

- Both 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, $b \leq g(f\ b)$ and $f\ (g\ a) \sqsubseteq a$, respectively known as *left-cancellation* and *right-cancellation*.

It may happen that a cancellation law holds up to equality, for instance $f\ (g\ a) = a$, in which case the connection is said to be *perfect* on the particular side [1].

Simplicity. Simple relations (that is, partial functions) will be particularly relevant in the sequel because of their ubiquity in software modeling. In particular, they will be used in this paper to model data *identity* and any kind of data structure "embodying a functional dependency" [58] such as eg. relational database tables, memory segments (both static and dynamic) and so on.

In the same way simple relations generalize functions (figure 3), *shunting* rules (26, 27) generalize to

$$S \cdot R \subseteq T \equiv (\delta\, S) \cdot R \subseteq S^\circ \cdot T \tag{29}$$

$$R \cdot S^\circ \subseteq T \equiv R \cdot \delta\, S \subseteq T \cdot S \tag{30}$$

[5] For instance, the *shunting* rule is called *cancellation law* in [70].

for S simple. These rules involve the *domain* operator (δ) whose GC, as mentioned in table 1, involves coreflexives on the lower side:

$$\delta\, R \subseteq \Phi \equiv R \subseteq \top \cdot \Phi \tag{31}$$

We will draw harpoon arrows $B \xleftharpoondown{R} A$ or $A \xrightharpoondown{R} B$ to indicate that R is simple. Later on we will need to describe simple relations at pointwise level. The notation we shall adopt for this purpose is borrowed from VDM [38], 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[6] and $dom\ S$ is the set-theoretic version of coreflexive $\delta\, S$ above, that is,

$$\delta\, S = \Phi_{dom\ S} \tag{32}$$

holds (cf. the isomorphism between sets and coreflexives). In this way, relation S itself can be written as $\{a \mapsto S\ a \mid a \in dom\ S\}$ and projection $f \cdot S \cdot g^\circ$ as

$$\{g\ a \mapsto f(S\ a) \mid a \in dom\ S\} \tag{33}$$

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:

$$M \cdot N^\circ \subseteq id \tag{34}$$

(Suggestion: resort to universal property $(R \cup S) \subseteq X \ \equiv\ R \subseteq X \wedge S \subseteq X$.) Furthermore show that (34) 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 [25]. □

Exercise 5. It will be useful to order relations with respect to how defined they are:

$$R \preceq S \equiv \delta\, R \subseteq \delta\, S \tag{35}$$

From $\top = \mathsf{ker}\,!$ draw another version of (35), $R \preceq S \ \equiv\ ! \cdot R \subseteq\ ! \cdot S$, and use it to derive

$$R \cdot f^\circ \preceq S \equiv R \preceq S \cdot f \tag{36}$$

□

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 $((R \cdot (\delta\,(S^\circ))) \cap T) \cup V$.

[6] VDM embodies a logic of partial functions (LPF) which takes this into account [38].

Summary. The material of this section is adapted from similar sections in [59, 60], which introduce the reader to the essentials of the PF-transform. While the notation adopted is standard [11], the presentation of the associated calculus is enhanced via the use of Galois connections, a strategy inspired by two (still unpublished) textbooks [1, 5]. There is a slight difference, perhaps: by regarding the underlying mathematics as that of a *transform* to be used wherever a "hard" formula [7] needs to be reasoned about, the overall flavour is more 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 [40].

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

$$\begin{array}{c|c}
\psi & [\![\psi]\!] \\
\hline
\langle \forall\, a, b \ :: \ b\,R\,a \Rightarrow b\,S\,a \rangle & R \subseteq S \\
\langle \forall\, a \ :: \ f\,a = g\,a \rangle & f \subseteq g \\
\langle \forall\, a \ :: \ a\,R\,a \rangle & id \subseteq R \\
\langle \exists\, a \ :: \ b\,R\,a \ \wedge \ a\,S\,c \rangle & b(R \cdot S)c \\
b\,R\,a \ \wedge \ b\,S\,a & b\,(R \cap S)\,a \\
b\,R\,a \vee b\,S\,a & b\,(R \cup S)\,a \\
(f\,b)\,R\,(g\,a) & b(f^\circ \cdot R \cdot g)a \\
\textsc{True} & b\,\top\,a \\
\textsc{False} & b\,\bot\,a
\end{array} \tag{37}$$

Exercise 6. Prove that relational composition preserves *all* relational classes in the taxonomy of figure 3. □

4 Data Structures

One of the main difficulties in studying data structuring is the number of disparate (inc. graphic) notations, programming languages and paradigms one has to deal with. Which should one adopt? While graphical notations such as the UML [15] are gaining adepts everyday, it is difficult to be precise 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 paper: presenting a fully formal semantics for the data structuring facilities offered by any commercial language or notation would be more than one paper in itself.

The purpose of this section is two fold: on the one hand, to show how overwhelming data structuring notations can be even in the case of simple data models such as our family tree (running) example; on the other hand, to show how to circumvent such disparity by expressing the same models in PF-notation. Particular emphasis will be put on describing Entity-relationship diagrams [30]. Later on we will go as far as capturing recursive data models by least fixpoints over polynomial types. Once again we warn the

[7] To use the words of Kreyszig [41] in his appreciation of the Laplace transform.

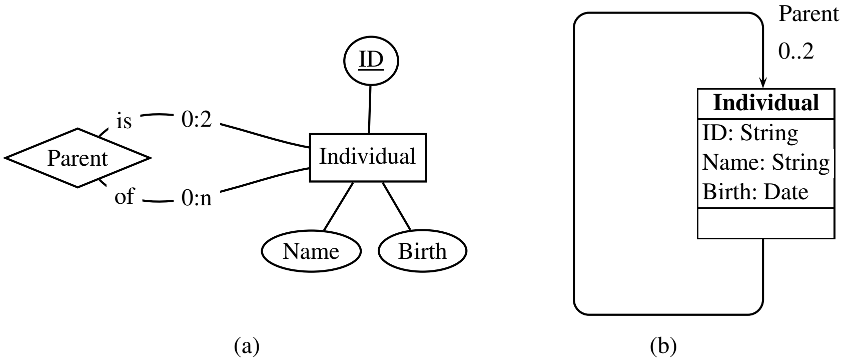


Fig. 4. ER and UML diagrams proposed for *genealogies*. Underlined identifiers denote keys.

reader that types and data modeling constructs in current programming languages are rather more complex than their obvious cousins in mathematics. For the sake of simplicity, we deliberately don't consider aspects such as non-strictness, lazy-evaluation, infinite data values [65] etc.

Back to the running example. Recall the family tree displayed in (9) and figure 2. Suppose requirements ask us 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, for instance that of figure 4(a). But many others will regard this notation too old-fashioned and will propose something like the UML diagram of figure 4(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 figure 2a — 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 plain 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 figure 2b.

What about functional programmers? By looking at pedigree tree (9) where we started from, an inductive data type can be defined, eg. in Haskell,

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

(38)

whereby (9) 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 still be encoded in XML notation eg. using DTD

```
<!-- 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 (9). However, how can one ensure that the same *print-family-tree* operation won't loop forever while retrieving data from eg. figure 2b? This would clearly happen if, by mistake, record $\begin{array}{|c|c|c|}\hline 1 & Father & 2 \\ \hline \end{array}$ in figure 2b were updated to $\begin{array}{|c|c|c|}\hline 1 & Father & 5 \\ \hline \end{array}$: $Peter$ would become a descendant of himself!

Several questions suggest themselves: 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, abstract (ie. technology free) unifying notation. But syntax alone is not enough: the ability to *reason* in such a notation is essential, otherwise different data models won't be comparable. Thus the reason why, in what follows, we choose the PF-notation as unifying framework [8].

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 subject to retrieving and analytical processing. But, what *is* a record?

Any row in the tables of figure 2b is a record, ie. *records a fact*. For instance, record $\begin{array}{|c|c|c|}\hline 5 & Peter & 1991 \\ \hline \end{array}$ tells: *Peter, whose ID number is 5, was born in 1991.* A mathematician would have written $(5, Peter, 1991)$ instead of *drawing* the tabular stuff and would have inferred $(5, Peter, 1991) \in I\!N \times String \times I\!N$ from $5 \in I\!N$, $Peter \in String$ and $1991 \in I\!N$, where, given two types A and B, their (Cartesian) product $A \times B$ is the set $\{(a, b) \mid a \in A \wedge b \in B\}$. So records can be regarded as *tuples* which inhabit *products* of types.

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 given datatypes. This syntax can be decorated with names, eg.

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

as is the case of `PTree` (38). In the C programming language, the $A \times B$ datatype is realized using "struct"'s, eg.

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

The diagram below is suggestive of what product $A \times B$ actually means, where f and g are functions, the two projections π_1, π_2 are such that

$$\pi_1(a, b) = a \ \wedge \ \pi_2(a, b) = b \tag{39}$$

[8] The "everything is a relation" motto implicit in this approach is also the message of Alloy [36], a notation and associated model-checking tool which has been successful in *alloying* a number of disparate approaches to software modeling, namely model-orientation, object-orientation, etc. Quoting [36]: *(...) "the Alloy language and its analysis are a Trojan horse: an attempt to capture the attention of software developers, who are mired in the tar pit of implementation technologies, and to bring them back to thinking deeply about underlying concepts".*

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

and function $\langle f, g\rangle$ (read: *"f split g"*) is defined by $\langle f, g\rangle c \overset{\text{def}}{=} (f\ c, g\ c)$. The diagram expresses the two cancellation properties, $\pi_1 \cdot \langle f, g\rangle = f$ and $\pi_2 \cdot \langle f, g\rangle = f$, which follow from a more general (universal) property,

$$k = \langle f, g\rangle \;\equiv\; \pi_1 \cdot k = f \,\wedge\, \pi_2 \cdot k = g \tag{40}$$

which holds for arbitrary (suitably typed) functions f, g and k. This tells that, given functions f and g, each producing inhabitants of types A and B, respectively, there is a unique function $\langle f, g\rangle$ which combines f and g so as to produce inhabitants of product type $A \times B$. Read in another way: any function k delivering results into type $A \times B$ can be uniquely decomposed into its two left and right components.

It can be easily checked that the definition of $\langle f, g\rangle$ given above PF-transforms to $\langle f, g\rangle = \pi_1^\circ \cdot f \cap \pi_2^\circ \cdot g$. (Just re-introduce variables and simplify, thanks to (39), (20), etc.) This provides a hint on how to generalize the *split* combinator to relations [9]:

$$\langle R, S\rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \tag{41}$$

To feel the meaning of the extension we introduce variables in (41) and simplify:

$$\begin{array}{cl}
& \langle R, S\rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \\
\equiv & \quad \{ \text{ introduce variables; (37) } \} \\
& (a, b)\langle R, S\rangle c \;\equiv\; (a, b)(\pi_1^\circ \cdot R)c \,\wedge\, (a, b)(\pi_2^\circ \cdot S)c \\
\equiv & \quad \{ \text{ (20) twice } \} \\
& (a, b)\langle R, S\rangle c \;\equiv\; \pi_1(a, b)\ R\ c \,\wedge\, \pi_2(a, b)\ S\ c \\
\equiv & \quad \{ \text{ projections (39) } \} \\
& (a, b)\langle R, S\rangle c \;\equiv\; a\ R\ c \,\wedge\, b\ S\ c
\end{array}$$

So, relational splits enable one to PF-transform logical formulæ involving more than two variables.

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

$$R \times S \overset{\text{def}}{=} \langle R \cdot \pi_1, S \cdot \pi_2\rangle \tag{42}$$

So we can add two more entries to table (37):

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

Finally 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 $\pi_i(a_1, \ldots, a_n) = a_i$.

[9] Read more about this construct (which is also known as a *fork algebra* [28]) in section 7 and, in particular, in exercise 27.

Exercise 7. Identify which types are involved in the following bijections:

$$flatr(a,(b,c)) \overset{\mathrm{def}}{=} (a,b,c) \tag{43}$$

$$flatl((b,c),d) \overset{\mathrm{def}}{=} (b,c,d) \tag{44}$$

□

Exercise 8. Show that the side condition of the following *split-fusion* law [10]

$$\langle R,S\rangle \cdot T = \langle R\cdot T, S\cdot T\rangle \quad \Leftarrow \quad R\cdot(\mathsf{img}\,T)\subseteq R \vee S\cdot(\mathsf{img}\,T)\subseteq S \tag{45}$$

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

Exercise 9. Write the following cancellation law with less symbols assuming that $R \preceq S$ and $S \preceq R$ (35) hold:

$$\pi_1 \cdot \langle R,S\rangle = R\cdot \delta\, S \;\wedge\; \pi_2 \cdot \langle R,S\rangle = S\cdot \delta\, R \tag{46}$$

□

Data type sums. The following is a declaration of a date 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

$$\begin{tikzcd}
Bool \arrow[r, "i_1"] \arrow[dr, "Boo"'] & Bool+String \arrow[d, "{[Boo\ ,Err]}"] & String \arrow[l, "i_2"'] \arrow[dl, "Err"] \\
& X &
\end{tikzcd}$$

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 :

$$[R\ ,S] = (R\cdot i_1^\circ)\cup(S\cdot i_2^\circ) \qquad \text{cf.} \tag{47}$$

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

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

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

[10] 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*:

$$R + S = [i_1 \cdot R \, , i_2 \cdot S] \tag{48}$$

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 programmed on purpose, eg. in C (using unions)

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

where explicit integer tags are introduced so as 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 [11]. 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 capture the abstract contents of generative grammars (expressed in extended BNF notation) once non-terminal symbols are identified with types and terminal symbols are filtered. The conversion is synthesized by the following table,

BNF NOTATION		POLYNOMIAL NOTATION
$\alpha \mid \beta$	$\mapsto$	$\alpha + \beta$
$\alpha\beta$	$\mapsto$	$\alpha \times \beta$
ϵ	$\mapsto$	1
a	$\mapsto$	1

(49)

applicable to the right hand side of BNF-productions, where α, β range over sequences of terminal or non-terminal symbols, ϵ stands for *empty* and a ranges over terminal symbols. For instance, production $X \ \rightarrow \ \epsilon \mid a \; A \; X$ (where X, A are non-terminals and a is terminal) leads to equation

$$X = 1 + A \times X \tag{50}$$

[11] 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 9.

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, one 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 [68] and the analysis of datatypes with pointers [69] lead to systems of polynomial equations, that is, to mutually recursive datatypes. For instance, the two *typedef*s above lead to $Node = A \times X$ and to $X = 1 + Node$. It is the substitution of $Node$ by $A \times X$ in the second equation which gives raise to (50). 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 be dealt with later on in section 5. So, for instance, the *PTree* datatype illustrated above in the XML and Haskell syntaxes is captured by the equation

$$PTree \cong Ind \times (PTree + 1) \times (PTree + 1) \tag{51}$$

where $Ind = Name \times Birth$ packages the information relative to name and birth year, which don't participate in the recursive machinery and are, in a sense, parameters of the model. Thus one may write $PTree \cong \mathsf{G}(Ind, PTree)$, in which G abstracts the particular pattern of recursion chosen to model family trees

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

where X refers to the parametric information and Y to the inductive part [12].

Let us now think of the operation which fetches a particular individual from a given $PTree$. From (51) one is intuitively led to an algorithm which *either* finds the individual (Ind) at the root of the tree, *or* tries and finds it in the left sub-tree ($PTree$) *or* tries and finds it in the right sub-tree ($PTree$). Why is this strategy "the natural" and obvious one? The answer to this question leads to the notion of datatype *membership* which is introduced below.

Membership. There is a close relationship between the *shape* of a data structure and the algorithms which fetch data from it. Put in other words: every instance of a given datatype is a kind of *data container* whose mathematical structure determines the particular *membership* tests upon which such algorithms are structured.

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, lists are also *container types*, the intuition

[12] Types such as $PTree$, which are structured around another datatype (cf. G) which captures its structural "shape" are often referred to as *two-level types* in the literature [66].

being that a belongs (or occurs) in list $l \in A^\star$ iff it can be found in any of its positions. In this case, membership has type $A \xleftarrow{\in} A^\star$ (note the overloading of symbol $\in$). But even product $A \times A$ has membership too: 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 [11, 32, 59]. Below we deal with polynomial type membership, which is what it required in this paper. 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:

$$\in_{\mathsf{K}} \stackrel{\rm def}{=} \bot \tag{52}$$

$$\in_{\mathsf{Id}} \stackrel{\rm def}{=} id \tag{53}$$

$$\in_{\mathsf{F}\times\mathsf{G}} \stackrel{\rm def}{=} (\in_{\mathsf{F}} \cdot \pi_1) \cup (\in_{\mathsf{G}} \cdot \pi_2) \tag{54}$$

$$\in_{\mathsf{F}+\mathsf{G}} \stackrel{\rm def}{=} [\in_{\mathsf{F}}\, , \in_{\mathsf{G}}] \tag{55}$$

$$\in_{\mathsf{F}\cdot\mathsf{G}} \stackrel{\rm def}{=} \in_{\mathsf{G}} \cdot \in_{\mathsf{F}} \tag{56}$$

Exercise 10. 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 (53) above clearly indicates that, sooner or later, equality plays its role when checking for polynomial membership. And equality of complex objects is 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 figure 2b. So, instead of storing a collection of objects of (say) type A in a set of (say) 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 figure 2b.

However, thinking in terms of *tabular relations* expressed by sets of tuples where particular attributes ensure unique identification[13], as is typical of database theory [45], is neither sufficiently general nor agile enough for reasoning purposes. References [56, 58] show that relational *simplicity* [14] is what matters in unique identification. So

[13] These attributes are known as *keys*.

[14] Recall that a relation is simple wherever its image is coreflexive (19).

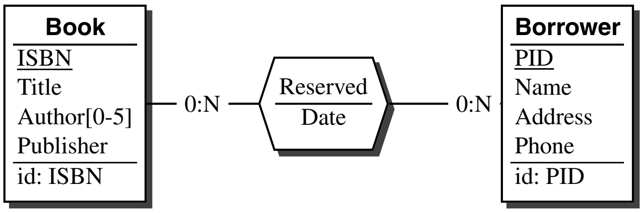


Fig. 5. Sample of GER diagram (adapted from [30]). Underlined identifiers denote keys.

it suffices to regard collections of uniquely identified objects A as simple relations of type

$$K \rightharpoonup A \tag{57}$$

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 [47].

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 [30]. Figure 5 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 T_a of attribute a by mentioning a alone:

$$Borrowers \stackrel{\mathrm{def}}{=} PID \rightharpoonup Name \times Address \times Phone$$

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:

$$Books \stackrel{\mathrm{def}}{=} ISBN \rightharpoonup Title \times (5 \rightharpoonup Author) \times Publisher \tag{58}$$

Note the use of number 5 to denote the initial segment of the natural numbers ($I\!N$) 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*. Simple relations also capture relationship formal semantics, this time involving the identities of the entities engaged. In this case:

$$Reserved \stackrel{\mathrm{def}}{=} ISBN \times PID \rightharpoonup Date$$

Altogether, the diagram specifies datatype $Db \stackrel{\mathrm{def}}{=} Books \times Borrowers \times Reserved$ inhabited by triples of simple relations.

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

$$\begin{array}{l} \phi(M,N,R) \stackrel{\mathrm{def}}{=} \\ \quad \langle \forall\, i,p,d \ :: \ d\, R\, (i,p) \Rightarrow \langle \exists\, x \ :: \ x\, M\, i \rangle \ \wedge \ \langle \exists\, y \ :: \ y\, M\, p \rangle \rangle \end{array} \qquad (59)$$

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 this, their definition in the predicate calculus is not agile enough for calculation purposes. Is there an alternative?

Space constraints preclude presenting the calculation which would show (59) *equivalent* to the following, much more concise PF-definition:

$$\phi(M,N,R) \stackrel{\mathrm{def}}{=} R \cdot {\in^\circ} \preceq M \ \wedge \ R \cdot {\in^\circ} \preceq N \qquad (60)$$

cf. diagram

$$\begin{tikzcd}
ISBN \arrow[d, "M"'] & ISBN \times PID \arrow[l, "\in=\pi_1"'] \arrow[r, "\in=\pi_2"] \arrow[d, "R"'] & PID \arrow[d, "N"] \\
\begin{array}{c} Title \times (5 \rightharpoonup \\ Author) \times \\ Publisher \end{array} & Date & \begin{array}{l} Name \times \\ Address \times \\ Phone \end{array}
\end{tikzcd}$$

To understand (60) and the diagram above, the reader must recall the definition of the $\preceq$ ordering (35) — 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:

$$ISBN \xleftarrow{\in} ISBN \times PID$$

$$\begin{array}{cl}
= & \{ \text{ polynomial decomposition, membership of product (54) } \} \\
& (\in_{\mathsf{Id}} \cdot \pi_1) \cup (\in_{\mathsf{PID}} \cdot \pi_2) \\
= & \{ \text{ (52) and (53) } \} \\
& id \cdot \pi_1 \cup \bot \cdot \pi_2 \\
= & \{ \text{ trivia } \} \\
& \pi_1
\end{array}$$

Multiplicity labels 0:N in the diagram of figure 5 indicate that there is no limit to the number of books borrowers 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 **Book** 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:

$$Reserved \stackrel{\mathrm{def}}{=} PID \rightharpoonup ISBN \times Date \tag{61}$$

Altogether, 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, $\phi(M, N, R) \stackrel{\mathrm{def}}{=} (\in \cdot R)^\circ \preceq M \ \wedge \ R \preceq N$, see diagram:

$$\begin{tikzcd}
ISBN \arrow[d, "M"'] & ISBN \times Date \arrow[l, "\in = \pi_1"'] & PID \arrow[l, harpoon, "R"'] \arrow[d, "N"] \\
Title \times (5 \rightharpoonup Author) \times Publisher & & Name \times Address \times Phone
\end{tikzcd}$$

In retrospect, note the similarity in shape between these diagrams and the corresponding Entity-Relationship diagrams. The main advantage of the former resides in their richer semantics enabling formal reasoning, as we shall see in the sequel.

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). There are so many instances of NSRI that *genericity* is the only effective way to address the topic [15]. The issue is that, whatever programming language is adopted, one faces the same (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, of 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 role of names), etc.

[15] For further insight into *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/~rm135`.

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

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

where 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 $(j = 1, n_i)$ and where T_i aggregates all types which don't participate in NSRI.

Assuming that all such F_i have membership, we can draw diagram

$$\begin{tikzcd}
N_i \arrow[r, harpoon, "S_i"] \arrow[dr, harpoon, "\in_{i,j} \cdot S_i"'] & \mathsf{F}_i(T_i, N_1, \ldots, N_j, \ldots, N_{n_i}) \arrow[d, "\in_{i,j}"] \\
& N_j
\end{tikzcd}$$

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

$$\langle \forall\, i, j \; :: \; (\in_{i,j} \cdot S_i)^{\circ} \preceq S_j \rangle \tag{62}$$

which, once the definition order $\preceq$ (35) is spelt out, converts to the pointwise:

$$\langle \forall\, n, m \; : \; n \in \mathit{dom}\; S_i \; : \; m \in_{i,j} (S_i\; n) \Rightarrow m \in \mathit{dom}\; S_j \rangle$$

Of course, (62) 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*. For instance, $\mathbb{N} \overset{H}{\rightharpoonup} \mathsf{F}\,(T, \mathbb{N})$ models a heap "of shape" F where T is some datatype of interest and addresses are natural numbers $(\mathbb{N})$. 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 9).

Summary. This section addressed data-structuring from a double viewpoint: the one of programmers wishing to build data models in their chosen programming medium and the one of the software analyst wishing to bridge between models in different notations in order to eventually control data impedance mismatch. The latter entailed the abstraction of disparate data structuring notations into a common unifying one, that of binary relations and the PF-transform. This makes it possible to study data impedance mismatch from a formal perspective.

5 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 (7). 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:

$$\left\{\begin{array}{l} -\ R \text{ is a representation } (\mathsf{ker}\, R = id) \\ -\ F \text{ is an abstraction } (\mathsf{img}\, F = id) \\ -\ R \text{ and } S \text{ are connected: } R \subseteq F^{\circ} \end{array}\right. \tag{63}$$

The higher the mismatch between A and B the more complex (R, F) are. The least impedance mismatch possible happens between a datatype and itself:

$$\begin{tikzcd} A \arrow[r, bend left, "id"] \arrow[r, phantom, "\leq"] & A \arrow[l, bend left, "id"] \end{tikzcd} \tag{64}$$

Another way to read (64) is to say that the $\leq$-ordering on data models is *reflexive*. It turns up that $\leq$ is also *transitive*,

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] & \wedge & B \arrow[r, bend left, "S"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "G"] & \Rightarrow & A \arrow[r, bend left, "S \cdot R"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "F \cdot G"] \end{tikzcd} \tag{65}$$

that is, data impedances compose. The calculation of (65) is immediate: composition respects abstractions and representations (recall exercise 6) and $(F \cdot G, S \cdot R)$ are connected:

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

Right-invertibility. A most beneficial consequence of (63) is the *right-invertibility* property

$$F \cdot R = id \tag{66}$$

which, written in predicate logic, expands to

$$\langle \forall\, a', a \ :: \ \langle \exists\, b \ :: \ a' \, F \, b \ \wedge \ b \, R \, a \rangle \ \equiv \ a' = a \rangle \tag{67}$$

The PF-calculation of (66) is not difficult:

$$F \cdot R = id$$

$$\begin{array}{ll}
\equiv & \{ \text{ equality of relations (14) } \} \\
& F \cdot R \subseteq id \ \wedge \ id \subseteq F \cdot R \\
\equiv & \{ \ \mathsf{img}\, F = id \text{ and } \mathsf{ker}\, R = id \text{ (63) } \} \\
& F \cdot R \subseteq F \cdot F^{\circ} \ \wedge \ R^{\circ} \cdot R \subseteq F \cdot R \\
\equiv & \{ \text{ converses } \} \\
& F \cdot R \subseteq F \cdot F^{\circ} \ \wedge \ R^{\circ} \cdot R \subseteq R^{\circ} \cdot F^{\circ} \\
\Leftarrow & \{ \ (F \cdot) \text{ and } (R^{\circ} \cdot) \text{ are monotone } \} \\
& R \subseteq F^{\circ} \ \wedge \ R \subseteq F^{\circ} \\
\equiv & \{ \text{ trivia } \} \\
& R \subseteq F^{\circ} \\
\equiv & \{ \ R \text{ and } F \text{ are connected (63) } \} \\
& \textsc{True}
\end{array}$$

Clearly, this *right-invertibility* property 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.

While (as we have just seen) $F \cdot R = id$ is entailed by (63), the converse entailment *does not hold*: $F \cdot R = id$ ensures R a representation and F surjective, but not 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 $\perp \subset F \subset !$.

Exercise 11. 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 (67) is entailed by the *pointwise* statement of (R, F) as a connected abstraction/ representation pair. □

Exercise 12. 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 checks the validity of

$$\begin{tikzcd} A \times B + 1 \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & (A+1) \times B \arrow[l, bend left, "f"] \end{tikzcd} \tag{68}$$

where $R \overset{\text{def}}{=} [i_1 \times id \, , \langle i_2, !^{\circ} \rangle]$ and $f = R^{\circ}$, that is, f satisfying clauses $f(i_1\ a, b) = i_1(a, b)$ and $f(i_2\ \textsc{Nil}, b) = i_2\ \textsc{Nil}$, where $\textsc{Nil}$ denotes the unique inhabitant of type 1. □

Right-invertibility happens to be *equivalent* to (63) wherever both the abstraction and the representation are *functions*, say f, r:

$$\begin{tikzcd} A \arrow[r, bend left, "r"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "f"] \end{tikzcd} \quad \equiv \quad f \cdot r = id \tag{69}$$

Let us show that $f \cdot r = id$ is equivalent to $r \subseteq f^\circ$ and entails f surjective and r injective:

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

The right invertibility property is a handy way of spotting $\leq$ rules. For instance, the following cancellation properties of product and sum hold [11]:

$$\pi_1 \cdot \langle f, g \rangle = f \ , \ \pi_2 \cdot \langle f, g \rangle = g \tag{70}$$

$$[g \ , f] \cdot i_1 = g \ , \ [g \ , f] \cdot i_2 = f \tag{71}$$

Suitable instantiations 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 (69) — the following $\leq$-rules

$$\begin{tikzcd}
A \arrow[r, bend left, "\langle id, g \rangle"] \arrow[r, phantom, "\leq"] & A \times B \arrow[l, bend left, "\pi_1"] & B \arrow[r, bend left, "\langle f, id \rangle"] \arrow[r, phantom, "\leq"] & A \times B \arrow[l, bend left, "\pi_2"]
\end{tikzcd} \tag{72}$$

$$\begin{tikzcd}
A \arrow[r, bend left, "i_1"] \arrow[r, phantom, "\leq"] & A + B \arrow[l, bend left, "{[id \ , f]}"] & B \arrow[r, bend left, "i_2"] \arrow[r, phantom, "\leq"] & A + B \arrow[l, bend left, "{[g \ , id]}"]
\end{tikzcd} \tag{73}$$

which 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 [17]. Alternatively, they can be replaced by the top relation $\top$ (meaning a *don't care* representation strategy). In the case of (73), even $\perp$ will work instead of f, g, leading, for $A = 1$, to the standard representation of datatype A by a "*pointer* to A":

$$\begin{tikzcd} A \arrow[r, bend left, "i_1"] \arrow[r, phantom, "\leq"] & A+1 \arrow[l, bend left, "i_1^\circ"] \end{tikzcd}$$

Exercise 13. Show that $[id\ , \perp] = i_1^\circ$ and that $[\perp\ , id] = i_2^\circ$. □

Isomorphic data types. As instance of (69) consider f and r such that both

$$\begin{tikzcd} A \arrow[r, bend left, "r"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "f"] \arrow[r, phantom, "\wedge"] & C \arrow[r, bend left, "f"] \arrow[r, phantom, "\leq"] & A \arrow[l, bend left, "r"] \end{tikzcd}$$

hold. This is equivalent to

$$\begin{array}{l} \quad r \subseteq f^\circ \ \wedge \ f \subseteq r^\circ \\ \equiv \qquad \{ \text{ converses ; (14) } \} \\ \quad r^\circ = f \end{array} \tag{74}$$

So r (a function) is the converse of another function f. This means that both are bijections (isomorphisms) — recall figure 3 — since

$$f \text{ is an isomorphism} \ \equiv \ f^\circ \text{ is a function} \tag{75}$$

In a diagram:

$$\begin{tikzcd} A \arrow[r, bend left, "r=f^\circ"] \arrow[r, phantom, "\cong"] & C \arrow[l, bend left, "f=r^\circ"] \end{tikzcd}$$

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, two isomorphic types A and C are "abstractly" the same. Here is a trivial example

$$\begin{tikzcd} A \times B \arrow[r, bend left, "swap"] \arrow[r, phantom, "\cong"] & B \times A \arrow[l, bend left, "swap"] \end{tikzcd} \tag{76}$$

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.

The question arises: how can one be *certain* that $swap$ is an isomorphism? A constructive, elegant way is to follow the advice of (75), which appeals to calculating the converse of $swap$,

$$\begin{array}{cl}
 & swap^\circ \\
= & \quad \{\ (41)\ \} \\
 & (\pi_1^\circ \cdot \pi_2 \cap \pi_2^\circ \cdot \pi_1)^\circ \\
= & \quad \{\ \text{converses}\ \} \\
 & \pi_2^\circ \cdot \pi_1 \cap \pi_1^\circ \cdot \pi_2 \\
= & \quad \{\ (41)\text{ again}\ \} \\
 & swap
\end{array}$$

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

Exercise 14. The calculation just above was too simple. To recognize the power of (75), prove the associative property of disjoint union,

$$\begin{tikzcd} A + (B + C) \arrow[r, bend left, "r"] \arrow[r, phantom, "\cong"] & (A + B) + C \arrow[l, bend left, "f = [id + i_1\ , i_2 \cdot i_2]"] \end{tikzcd} \tag{77}$$

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° and prove the two cancellations $f \cdot f^\circ = id$ and $f^\circ \cdot f = id$. □

Exercise 15. The following are known isomorphisms involving sums and products:

$$A \times (B \times C) \cong (A \times B) \times C \tag{78}$$
$$A \cong A \times 1 \tag{79}$$
$$A \cong 1 \times A \tag{80}$$
$$A + B \cong B + A \tag{81}$$
$$C \times (A + B) \cong C \times A + C \times B \tag{82}$$

Guess the relevant isomorphism pairs. □

Exercise 16. Show that (75) holds, for f a function (of course). □

Relation transposes. Once again let us have a look at isomorphism pair (r, f) in (74), this time to introduce variables in the equality:

$$r^\circ = f$$

$$
\begin{array}{cl}
\equiv & \{ \text{ introduce variables } \} \\
 & \langle \forall\, a, c \ :: \ c\,(r^\circ)\; a \equiv c\; f\; a \rangle \\
\equiv & \{ \text{ (20) } \} \\
 & \langle \forall\, a, c \ :: \ r\; c = a \ \equiv \ c = f\; a \rangle
\end{array}
$$

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

$$k = \Lambda R \ \equiv \ R = \in \cdot\, k \tag{83}$$

where Λ converts a binary relation into *the corresponding* set-valued function [11], of

$$k = \mathit{tot}\ S \ \equiv \ S = \underbrace{i_1^\circ \cdot k}_{\mathit{untot}\ k} \tag{84}$$

where *tot totalizes* a simple relation S into *the corresponding* "Maybe-function" [16], and of

$$k = \mathit{curry}\ f \ \equiv \ f = \underbrace{\mathit{ap} \cdot (k \times \mathit{id})}_{\mathit{uncurry}\ k} \tag{85}$$

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

These properties of Λ, tot and curry are normally referred to as *universal properties*, because of their particular pattern of universal quantification which ensures uniqueness [17]. Novice readers will find them less cryptic once further (quantified) variables are introduced on their right hand sides:

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

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

$$
\begin{tikzcd}
(\mathcal{P}B)^A \arrow[r, bend left, "(\in\cdot)"] \arrow[r, phantom, "\cong"] & A \to B \arrow[l, bend left, "\Lambda"] & (B+1)^A \arrow[r, bend left, "\mathit{untot}=(i_1^\circ\cdot)"] \arrow[r, phantom, "\cong"] & A \rightharpoonup B \arrow[l, bend left, "\mathit{tot}"]
\end{tikzcd}
$$

$$
\begin{tikzcd}
(B^A)^C \arrow[r, bend left, "\mathit{uncurry}"] \arrow[r, phantom, "\cong"] & B^{C \times A} \arrow[l, bend left, "\mathit{curry}"]
\end{tikzcd}
\tag{86}
$$

where the exponential notation Y^X describes the datatype of all functions from X to Y.

[16] See [59]. This corresponds to the view that simple relations are "possibly undefined" (ie. partial) functions. Also recall that $A \xleftarrow{i_1^\circ} A + 1$ is the membership of Maybe.

[17] Consider, for instance, the right to left implication of (85): this tells that, given f, $\mathit{curry}\ f$ is *the only* function satisfying $f = \mathit{ap} \cdot (k \times \mathit{id})$.

Exercise 17. (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 18. The following is a well-known isomorphism involving exponentials:

$$\begin{tikzcd}(B \times C)^A \arrow[r, bend left, "\langle(\pi_1\cdot),(\pi_2\cdot)\rangle"] \arrow[r, phantom, "\cong"] & B^A \times C^A \arrow[l, bend left, "\langle \_,\_ \rangle"]\end{tikzcd} \tag{87}$$

Write down the *universal property* captured by (87). □

Exercise 19. Relate function $(p2p\ p)b \overset{\text{def}}{=} \ if\ \ b\ \ then\ \ (\pi_1\ p)\ \ else\ \ (\pi_2\ p)$ (read $p2p$ as *"pair to power"*) with isomorphism

$$A \times A \ \cong\ A^2 \tag{88}$$

□

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 Λ and tot (83, 84) already establish relationships of this kind. Let us see two more which will prove useful in calculations to follow.

"Relational currying". Consider isomorphism

$$\begin{tikzcd}(C \to A)^B \arrow[r, bend left, "\overline{(\_)}^\circ"] \arrow[r, phantom, "\cong"] & B \times C \to A \arrow[l, bend left, "\overline{(\_)}"]\end{tikzcd} \tag{89}$$

and associated universal property,

$$k = \overline{R} \equiv \langle \forall\ a, b, c\ ::\ a\ (k\ b)\ c \equiv a\ R\ (b, c) \rangle \tag{90}$$

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 (83):

$$\begin{array}{ll}
 & B \times C \to A \\
\cong & \qquad \{\ \Lambda/(\in\cdot)\ \text{(83, 86)}\ \} \\
 & (\mathcal{P}A)^{B \times C} \\
\cong & \qquad \{\ curry/uncurry\ \} \\
 & ((\mathcal{P}A)^C)^B \\
\cong & \qquad \{\ \text{exponentials preserve isomorphisms}\ \} \\
 & (C \to A)^B
\end{array}$$

The fact that, for simple relations, one could have resorted above to the $Maybe$-transpose (84) instead of the power transpose (83), leads to the conclusion that relational "currying" preserves simplicity:

$$\begin{tikzcd}
(C \rightharpoonup A)^B \arrow[rr, bend left, "\overline{(\_)}^{\circ}"] \arrow[rr, phantom, "\cong"] & & B \times C \rightharpoonup A \arrow[ll, bend left, "\overline{(\_)}"]
\end{tikzcd} \qquad (91)$$

Since all relations are simple in (91), we can use notation convention (33) in the following pointwise definition of $\overline{M}$ (for M simple):

$$\overline{M}\, b = \{c \mapsto M(b', c) \mid (b', c) \in dom\ M \ \wedge\ b' = b\} \qquad (92)$$

This rule will play its role in multiple (foreign) key synthesis, see section 6.

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:

$$\begin{tikzcd}
2^A \arrow[rr, bend left, "\lambda p.\{a \in A \mid p\ a\}"] \arrow[rr, phantom, "\cong"] & & \mathcal{P}A \arrow[ll, bend left, "\lambda S.(\lambda a.a \in S)"]
\end{tikzcd} \quad \text{cf.} \quad p = (\in S) \ \equiv\ S = \{a \mid p\, a\} \qquad (93)$$

Here we see the correspondence between set comprehension and membership testing expressed by 2-valued functions, ie. predicates. By combining the *tot/untot* isomorphism (86) with (93) we obtain

$$\begin{tikzcd}
\mathcal{P}A \arrow[rr, bend left, "s2m"] \arrow[rr, phantom, "\cong"] & & A \rightharpoonup 1 \arrow[ll, bend left, "dom"]
\end{tikzcd} \qquad (94)$$

where $s2m\ S = !\cdot \Phi_S$ and dom is defined by (32). This shows that every fragment of *bang* (!) models a set [18].

Exercise 20. Show that "obvious" facts such as $S = \{a|a \in S\}$ and $p\,x \equiv x \in \{a|p\,a\}$ stem from (93). Investigate other properties of set-comprehension which can be drawn from (93). □

Relators and $\leq$-monotonicity. A lesson learned from (69) is that right-invertible functions (surjections) have a $\leq$-rule of their own. For instance, predicate $f\ n \ \stackrel{\mathrm{def}}{=}\ n \neq 0$ 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. trees of Booleans) can

[18] Relations at most *bang* (!) are referred to as *right-conditions* in [32].

be represented by a similar structure involving integers (eg. trees of integers). However, what does the word "similar" mean in this context? Typically, when building such a tree of integers, a C programmer looks at it 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 *promote* the A-into-B representation to structures of type G:

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] & \Rightarrow & \mathsf{G}\,A \arrow[r, bend left, "\mathsf{G}\,R"] \arrow[r, phantom, "\leq"] & \mathsf{G}\,B \arrow[l, bend left, "\mathsf{G}\,F"] \end{tikzcd}$$

The questions arise: does this hold for *any* parametric type G we can think of? and what do relations $\mathsf{G}\,R$ 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 to each other:

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

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

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

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

$$\mathsf{G}\,id = id \qquad (95)$$

$$\mathsf{G}\,(R \cdot S) = (\mathsf{G}\,R) \cdot (\mathsf{G}\,S) \qquad (96)$$

$$\mathsf{G}\,(R^{\circ}) = (\mathsf{G}\,R)^{\circ} \tag{97}$$

$$R \subseteq S \Rightarrow \mathsf{G}\,R \subseteq \mathsf{G}\,S \tag{98}$$

These turn up to be the properties of a *relator* [6], a concept which extends that of a *functor* to relations: a parametric datatype G is said to be a relator wherever, given a relation R from A to B, $\mathsf{G}\,R$ extends R to G-structures. In other words, it is a relation from $\mathsf{G}\,A$ to $\mathsf{G}\,B$, cf.

$$\begin{tikzcd}
A \arrow[r, dotted, no head] \arrow[d, "R"'] & \mathsf{G}\,A \arrow[d, "\mathsf{G}\,R"] \\
B \arrow[r, dotted, no head] & \mathsf{G}\,B
\end{tikzcd} \tag{99}$$

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 G in (95, 96) is that of a functor, and (97) and (98) become trivial — the former establishing that G preserves isomorphisms and the latter that G preserves equality (Leibniz).

It is easy to show that relators preserve all basic properties of relations as in figure 3. Two trivial relators are the *identity* relator Id, which is such that $\mathsf{Id}\,R = R$ and the *constant* relator 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 $R \times S$ (42) and sum $R + S$ (48).

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,

$$\begin{tikzcd}
B \arrow[r, dotted, no head] \arrow[d, "R"'] & K \rightharpoonup B \arrow[d, "K \rightharpoonup R"] \\
C \arrow[r, dotted, no head] & K \rightharpoonup C
\end{tikzcd}$$

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

$$N(K \rightharpoonup R)M \stackrel{\mathrm{def}}{=} \delta\,M = \delta\,N \;\wedge\; N \cdot M^{\circ} \subseteq R \tag{100}$$

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

$$(K \rightharpoonup f)M = f \cdot M \tag{101}$$

This can be extended to a bi-relator,

$$(g \rightharpoonup f)M = f \cdot M \cdot g^{\circ} \tag{102}$$

provided g is injective — recall (33).

Exercise 21. Show that instantiation $R := f$ in (100) leads to $N \subseteq f \cdot M$ and $f \cdot M \subseteq N$ in the body of (100), and therefore to (101). □

Exercise 22. Show that $(K \rightharpoonup \_)$ is a relator. □

Indirection and dereferencing. Indirection is a representation technique whereby data of interest stored in some data structure are replaced by references (pointers) to some global (dynamic) store — recall (57) — where the data are *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 (99) the datatype of interest (archived in some parametric container of type G, eg. binary trees of Bs). Let A be the natural numbers and S be simple. Since relators preserve simplicity, $\mathsf{G}\,S$ will be simple too, as depicted aside. The meaning of this diagram is that of declaring a generic function (say $rmap$) which, giving S simple, yields $\mathsf{G}\,S$ also simple. So $rmap$ has type

$$\begin{tikzcd} I\!\!N \arrow[r, dotted, dash] \arrow[d, "S"'] & \mathsf{G}\,I\!\!N \arrow[d, "\mathsf{G}\,S"] \\ B \arrow[r, dotted, dash] & \mathsf{G}\,B \end{tikzcd}$$

$$(I\!\!N \rightharpoonup B) \to (\mathsf{G}\,I\!\!N \rightharpoonup \mathsf{G}\,B) \tag{103}$$

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 (91) we infer that $rmap$ can be "uncurried" into a simple relation of type $((I\!\!N \rightharpoonup B) \times \mathsf{G}\,I\!\!N) \rightharpoonup \mathsf{G}\,B$ which is surjective, for finite structures. Of course we can replace $I\!\!N$ above by any data domain, say K (suggestive of *key*), with the same cardinality, that is, such that $K \ \cong\ I\!\!N$. Then

$$\begin{tikzcd} \mathsf{G}\,B \arrow[rr, bend left, "R"] \arrow[rr, phantom, "\leq"] & & (K \rightharpoonup B) \times \mathsf{G}\,K \arrow[ll, bend left, "Dref"] \end{tikzcd} \tag{104}$$

holds for abstraction relation $Dref$ such that $\overline{Dref} = rmap$, that is, such that (recalling (90))

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

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 Bs, in which fact $l'\ Dref\ (S, l)$ instantiates to $l'(S^\star)l$, itself meaning

$$\begin{array}{lll} l'(S^\star)l & \equiv & length\, l' = length\, l\ \wedge \\ & & \qquad \langle \forall\, i\ :\ 1 \leq i \leq length\, l :\ l\, i \in dom\ S\ \wedge\ (l'\ i) = S(l\ i) \rangle \end{array}$$

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

$$k \in l \Rightarrow \langle \exists\, b\ ::\ b\, S\, k \rangle \tag{105}$$

where $\in$ denotes finite list membership.

Exercise 23. Check that (105) PF-transforms to $(\in \cdot \underline{l})^\circ \preceq S$, an instance of NSRI (62) where $\underline{l}$ denotes the "everywhere l" constant function. □

Exercise 24. Define a representation function $r \subseteq \mathit{Dref}^\circ$ (104) for $\mathsf{G}\,X = X^\star$. □

Summary. This section presented the essence of this paper's approach to data calculation: a preorder ($\leq$) on data types which formalizes data impedance mismatch in terms of representation/abstraction pairs. This preorder is compatible with the data type constructors introduced in section 4 and leads to a data structuring calculus whose laws enable systematic calculation of data implementations from abstract models. This is shown in the sections which follow.

6 Calculating Database Schemes from Abstract Models

Relational schema modeling is central to the "open-ended list of mapping issues" identified in [42]. 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 9.

Relational schemes "relationally". Broadly speaking, a relational database is a n-tuple of tables, where each table is a relation involving value-level tuples. The latter 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

$$RDBT \overset{\text{def}}{=} \prod_{i=1}^{n}(\prod_{j=1}^{n_i} K_j \rightharpoonup \prod_{k=1}^{m_i} D_k) \tag{106}$$

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 *data of interest* (index k). Wherever $m_i = 0$ we have $\prod_{k=1}^{0} D_k \cong 1$, meaning — via (94) — 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 [45].

Compared to what we have seen so far, type $RDBT$ (106) is "flat": there are no sums, no exponentials, no room for a single recursive datatype. Thus the mismatch identified in [42]: how does one map structured data (eg. encoded in XML) or a text generated according to some grammar, or even 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 "flat" $RDBT$. Such rules share 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 (89) and (94) are already rules of this kind (from left to right), the latter enabling one to get rid of powersets and the other 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. [11]) that the *either* combinator $[R\ ,S]$ as defined by (47) is an isomorphism. This happens because one can always (uniquely) project a relation $(B+C) \xrightarrow{T} A$ into two components $B \xrightarrow{R} A$ and $C \xrightarrow{S} A$, such that $T = [R\ ,S]$. Thus we have

$$\begin{tikzcd}(B+C) \to A \arrow[r, phantom, "\cong"] \arrow[r, bend left, "{[\_\,,\_]^\circ}"] & (B \to A) \times (C \to A) \arrow[l, bend left, "{[\_\,,\_]}"]\end{tikzcd} \qquad (107)$$

which establishes universal property

$$T = [R\ ,S] \ \equiv\ T \cdot i_1 = R \ \wedge\ T \cdot i_2 = S \qquad (108)$$

When applied from left to right, rule (107) 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 (49). The application of rule (107) 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 (107), which is easily performed via the power-transpose [11], can alternatively be performed via the *Maybe*-transpose [59] — in the case of simple relations — meaning that relational *either* preserves simplicity:

$$\begin{tikzcd}(B+C) \rightharpoonup A \arrow[r, phantom, "\cong"] \arrow[r, bend left, "{[\_\,,\_]^\circ}"] & (B \rightharpoonup A) \times (C \rightharpoonup A) \arrow[l, bend left, "{[\_\,,\_]}"]\end{tikzcd} \qquad (109)$$

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

$$\begin{tikzcd}A \to (B+C) \arrow[r, phantom, "\cong"] \arrow[r, bend left, "\triangle_+"] & (A \to B) \times (A \to C) \arrow[l, bend left, "\overset{+}{\bowtie}"]\end{tikzcd} \qquad (110)$$

where

$$M \overset{+}{\bowtie} N \stackrel{\mathrm{def}}{=} i_1 \cdot M \cup i_2 \cdot N \qquad (111)$$

$$\triangle_+\ M \stackrel{\mathrm{def}}{=} (i_1^\circ \cdot M, i_2^\circ \cdot M) \qquad (112)$$

However, (110) does not hold as it stands for simple relations, because $\overset{+}{\bowtie}$ does not preserve simplicity: the union of two simple relations is not always simple. The weakest pre-condition for simplicity to be maintained is calculated as follows:

$$\begin{array}{ll}
 & M \overset{+}{\bowtie} N \text{ is simple} \\
\equiv & \quad \{ \text{ definition (111) } \} \\
 & (i_1 \cdot M \cup i_2 \cdot N) \text{ is simple} \\
\equiv & \quad \{ \text{ simplicity of union of simple relations (34) } \} \\
 & (i_1 \cdot M) \cdot (i_2 \cdot N)^\circ \subseteq id \\
\equiv & \quad \{ \text{ converses ; shunting (26, 27) } \} \\
 & M \cdot N^\circ \subseteq i_1^\circ \cdot i_2 \\
\equiv & \quad \{ \; i_1^\circ \cdot i_2 = \bot \text{ ; (29,30) } \} \\
 & \delta\, M \cdot \delta\, N \subseteq \bot \\
\equiv & \quad \{ \text{ coreflexives (15) } \} \\
 & \delta\, M \cap \delta\, N = \bot
\end{array} \qquad (113)$$

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

Exercise 25. Show that $\overset{+}{\bowtie} \cdot \triangle_+ = id$ holds. (NB: property $id + id = id$ can be of help in the calculation.) □

Exercise 26. Do better than in exercise 25 and show that $\overset{+}{\bowtie}$ is the converse of $\triangle_+$, of course finding inspiration in (75). Universal property (108) will soften calculations if meanwhile you show that $(M \overset{+}{\bowtie} N)^\circ \; = \; [M^\circ \, , N^\circ]$ holds. □

Getting rid of multivalued types. Recall the $Books$ type (58) defined earlier on. It deviates from $RDBT$ in the second factor of its range 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:

$$\begin{array}{ll}
 & A \rightharpoonup (D \times (B \rightharpoonup C)) \\
\cong & \quad \{ \; Maybe \text{ transpose (86) } \} \\
 & (D \times (B \rightharpoonup C) + 1)^A \\
\leq & \quad \{ \text{ (68) } \} \\
 & ((D + 1) \times (B \rightharpoonup C))^A \\
\cong & \quad \{ \text{ splitting (87) } \}
\end{array}$$

$$
\begin{array}{ll}
 & (D+1)^A \times (B \rightharpoonup C)^A \\
\cong & \quad \{ \ \mathit{Maybe} \text{ transpose (86, 89)} \ \} \\
 & (A \rightharpoonup D) \times (A \times B \rightharpoonup C)
\end{array}
$$

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

$$\begin{tikzcd}
A \rightharpoonup (D \times (B \rightharpoonup C)) \arrow[rr, bend left, "\triangle_n"] & \leq & (A \rightharpoonup D) \times (A \times B \rightharpoonup C) \arrow[ll, bend left, "\bowtie_n"]
\end{tikzcd} \tag{114}$$

where the “nested join” operator $\bowtie_n$ is defined by

$$M \bowtie_n N = \langle M, \overline{N} \rangle \tag{115}$$

— recall (91) — and $\triangle_n$ is

$$\triangle_n\, M = (\pi_1 \cdot M, usc(\pi_2 \cdot M)) \tag{116}$$

where usc (=“undo simple currying”) is defined in comprehension notation as follows,

$$usc\, M \stackrel{\mathrm{def}}{=} \{(a,b) \mapsto (M\ a)b \mid a \in dom\ M, b \in dom(Ma)\} \tag{117}$$

since M is simple. (Details about the calculation of this abstraction / representation pair can be found in [63].)

Example. Let us see the application of $\leq$-rule (114) to the $Books$ data model (58). We document each step by pointing out the involved abstraction/representation pair:

$$
\begin{array}{rl}
\mathit{Books} = & \mathit{ISBN} \rightharpoonup (\mathit{Title} \times (5 \rightharpoonup \mathit{Author}) \times \mathit{Publisher}) \\
\cong_1 & \quad \{ \ r_1 = id \rightharpoonup \langle\langle \pi_1, \pi_3 \rangle, \pi_2 \rangle \ , f_1 = id \rightharpoonup \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1 \rangle \ \} \\
 & \mathit{ISBN} \rightharpoonup (\mathit{Title} \times \mathit{Publisher}) \times (5 \rightharpoonup \mathit{Author}) \\
\leq_2 & \quad \{ \ r_2 = \triangle_n \ , f_2 = \bowtie_n, \text{cf. (114)} \ \} \\
 & (\mathit{ISBN} \rightharpoonup \mathit{Title} \times \mathit{Publisher}) \times (\mathit{ISBN} \times 5 \rightharpoonup \mathit{Author}) \\
= & \mathit{Books}_2
\end{array}
$$

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 [42], 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 (65), 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:

$$\begin{array}{l} r\,M = \triangle_n((id \rightharpoonup \langle\langle \pi_1, \pi_3 \rangle, \pi_2 \rangle)M) \\ = \qquad \{ \text{ (102) } \} \\ \quad \triangle_n(\langle\langle \pi_1, \pi_3 \rangle, \pi_2 \rangle \cdot M) \\ = \qquad \{ \text{ (116) } \} \\ \quad (\pi_1 \cdot \langle\langle \pi_1, \pi_3 \rangle, \pi_2 \rangle \cdot M, usc(\pi_2 \cdot \langle\langle \pi_1, \pi_3 \rangle, \pi_2 \rangle \cdot M)) \\ = \qquad \{ \text{ exercise 8 ; projections } \} \\ \quad (\langle \pi_1, \pi_3 \rangle \cdot M, usc(\pi_2 \cdot M)) \end{array}$$

Thanks to (33), 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 (117).

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

$$\begin{array}{l} f(M, N) = (id \rightharpoonup \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1 \rangle)(M \bowtie_n N) \\ = \qquad \{ \text{ (102) and (115) } \} \\ \quad \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1 \rangle \cdot \langle M, \overline{N} \rangle \\ = \qquad \{ \text{ exercise 8 ; projections } \} \\ \quad \langle \pi_1 \cdot \pi_1 \cdot \langle M, \overline{N} \rangle, \pi_2 \cdot \langle M, \overline{N} \rangle, \pi_2 \cdot \pi_1 \cdot \langle M, \overline{N} \rangle \rangle \\ = \qquad \{ \text{ exercise 9; } \overline{N} \text{ is a function } \} \\ \quad \langle \pi_1 \cdot M, \overline{N} \cdot \delta\, M, \pi_2 \cdot M \rangle \\ = \qquad \{ \text{ (92) } \} \\ \quad \{isbn \mapsto (\pi_1(M\ isbn), N', \pi_2(M\ isbn)) \mid isbn \in dom\ M\} \end{array}$$

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 $ISBN$ don't participate 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.

Summary. This section described the application of the calculus introduced in section 5 to the transformation of abstract data models targeted at relational database implementations. It also showed how more elaborate laws can be derived from simpler ones and how to synthesize composite "forward" and "backward" data mappings using the underlying relational calculus. We proceed to showing how to take further advantage of relational reasoning in synthesizing data type invariants entailed by the representation process.

7 Concrete Invariants

The fact that R and F are connected (63) in every $\leq$-rule (7) forces the range of R to be at most the domain of F, $\rho\, R \subseteq \delta\, F$. This means that the representation space (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 target 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 domain and range

$$\delta\, R = \mathsf{ker}\, R \cap id \tag{118}$$

$$\rho\, R = \mathsf{img}\, R \cap id \tag{119}$$

$$\rho\, (R \cdot S) = \rho\, (R \cdot \rho\, S) \tag{120}$$

$$\delta\, (R \cdot S) = \delta\, (\delta\, R \cdot S) \tag{121}$$

help in inferring *concrete invariants*, in particular those induced by $\leq$-chaining (65).

Concrete invariant calculation, which is in general nontrivial, is softened wherever $\leq$-rules are expressed by GCs [19]. In this case, the range of the representation (concrete invariant) can be computed as coreflexive $r \cdot f \cap id$, that is, predicate [20]

$$\phi\, x \stackrel{\mathrm{def}}{=} r(f\ x) = x \tag{122}$$

As illustration of this process, consider law

$$\begin{tikzcd}
A \rightarrow B \times C \arrow[r, bend left, "\langle(\pi_1 \cdot),(\pi_2 \cdot)\rangle"] \arrow[r, phantom, "\leq"] & (A \rightarrow B) \times (A \rightarrow C) \arrow[l, bend left, "\langle\_,\_\rangle"]
\end{tikzcd} \tag{123}$$

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

$$X \subseteq \langle R, S\rangle \ \equiv\ \pi_1 \cdot X \subseteq R \ \wedge\ \pi_2 \cdot X \subseteq S \tag{124}$$

Calculation of the concrete invariant induced by (123) follows:

$$\begin{array}{ll}
& \phi(R,S) \\
\equiv & \quad \{\ (122, 123)\ \} \\
& (R,S) = (\pi_1 \cdot \langle R,S\rangle, \pi_2 \cdot \langle R,S\rangle) \\
\equiv & \quad \{\ (46)\ \} \\
& R = R \cdot \delta\, S \ \wedge\ S = S \cdot \delta\, R \\
\equiv & \quad \{\ \delta\, X \subseteq \Phi \equiv X \subseteq X \cdot \Phi\ \} \\
& \delta\, R \subseteq \delta\, S \ \wedge\ \delta\, S \subseteq \delta\, R \\
\equiv & \quad \{\ (14)\ \} \\
& \delta\, R = \delta\, S
\end{array}$$

In other words: if equally defined R and S are joined and then decomposed again, this will be a lossless decomposition [58].

Similarly, the following concrete invariant can be shown to hold for rule (114) [21]:

$$\phi(M,N) \stackrel{\mathrm{def}}{=} N \cdot \in^{\circ} \preceq M \tag{125}$$

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 [22]).

[19] Of course, these have to be *perfect* (64) on the source (abstract) side.

[20] See Theorem 5.20 in [1].

[21] See [63] for details.

[22] See the *Unity of opposites* theorem of [5].

Exercise 27. Infer (124) from (41) and universal property

$$X \subseteq (R \cap S) \equiv (X \subseteq R) \wedge (X \subseteq S) \tag{126}$$

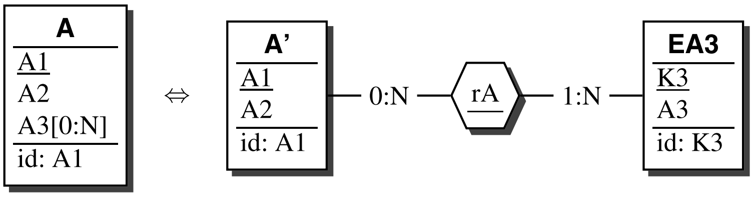
Moreover, show that (40) instantiates (124). □

Exercise 28. Show that (113) is the concrete invariant induced by rule (110), from left-to-right, in case all relations are simple. □

Concrete invariants play an important role in data refinement. For instance, Morgan [49] takes them into account in building *functional abstractions* of the form $af \cdot \Phi_{dti}$ where (entire) abstraction function af is explicitly constrained by concrete invariant dti. In the section which follows we show how such invariants help in calculating model transformations. The reader is also referred to [8] for a PF-theory of invariants in general.

8 Calculating Model Transformations

References [30] and [43] postulate a number of model transformation rules (concerning GERs in the first case and UML class diagrams in the second) which we are in position to calculate. We illustrate this process with rule 12.2 of [30], the rule which converts a (multivalued) attribute into an entity type:



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 (104) — whereafter we proceed as before:

$$\begin{array}{ll}
 & A_1 \rightharpoonup A_2 \times \mathcal{P}A_3 \\
\leq_1 & \quad \{ \text{(104)} \} \\
 & (K_3 \rightharpoonup A_3) \times (A_1 \rightharpoonup A_2 \times \mathcal{P}K_3) \\
\cong_2 & \quad \{ \text{(94)} \} \\
 & (K_3 \rightharpoonup A_3) \times (A_1 \rightharpoonup A_2 \times (K_3 \rightharpoonup 1)) \\
\leq_3 & \quad \{ \text{(114)} \} \\
 & (K_3 \rightharpoonup A_3) \times ((A_1 \rightharpoonup A_2) \times (A_1 \times K_3 \rightharpoonup 1)) \\
\cong_4 & \quad \{ \text{introduce ternary product} \} \\
 & \underbrace{(A_1 \rightharpoonup A_2)}_{A'} \times \underbrace{(A_1 \times K_3 \rightharpoonup 1)}_{rA} \times \underbrace{(K_3 \rightharpoonup A_3)}_{EA3}
\end{array}$$

The overall concrete invariant is

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

— recall eg. (125) — which can be further transformed into:

$$\begin{array}{ll}
\phi(M, R, N) = R \cdot \in^\circ \preceq M \ \wedge \ R \cdot \in^\circ \preceq N \\
\equiv \qquad \{ \text{ (54, 53) } \} \\
\quad R \cdot \pi_1^\circ \preceq M \ \wedge \ R \cdot \pi_2^\circ \preceq N \\
\equiv \qquad \{ \text{ (36) } \} \\
\quad R \preceq M \cdot \pi_1 \ \wedge \ R \preceq N \cdot \pi_2
\end{array}$$

In words, this means that relationship R (rA in the diagram) must 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.

9 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 recursive data structure representation in terms of non-recursive ones. In a sense, we want to show how to "get away with recursion" [56] in data modeling. It is a standard result (and every a programmer's experience) that *recursive types using products and sums can be implemented using pointers* [69]. Our challenge is to generalize this result and present it in a 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 G is a relator. The standard notation for such a solution is $\mu\mathsf{G}$. (This always exists when G is *regular* [11], a class which embodies all polynomial 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` (38) involves constructor `Node` and selectors `name`, `birth`, `mother` and `father`, which cannot be found in equation (51). 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.

The general case is depicted in the diagram aside, 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` (38) interpreted as solution to equation (51) is

$$\begin{tikzcd} \mu\mathsf{G} \arrow[r, bend left, "out"] \arrow[r, phantom, "\cong"] & \mathsf{G}\,\mu\mathsf{G} \arrow[l, bend left, "in"] \end{tikzcd}$$

$$in((n, b), m, f) \stackrel{\mathrm{def}}{=} Node\ n\ b\ m\ f \tag{127}$$

Programs handling $\mu\mathsf{G}$ can be of essentially two kinds: either they read (parse, inspect) $\mu\mathsf{G}$-structures (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 exhibiting their recursive (inductive) nature:

$$\begin{tikzcd} \mu\mathsf{G} \arrow[r, "out"] \arrow[d, "fold\ R"'] & \mathsf{G}\,\mu\mathsf{G} \arrow[d, "\mathsf{G}(fold\ R)"] \\ A & \mathsf{G}\,A \arrow[l, "R"] \end{tikzcd} \qquad \begin{tikzcd} \mu\mathsf{G} & \mathsf{G}\,\mu\mathsf{G} \arrow[l, "in"'] \\ A \arrow[u, "unfold\ R"] \arrow[r, "R"'] & \mathsf{G}\,A \arrow[u, "\mathsf{G}(unfold\ R)"'] \end{tikzcd}$$

Both *fold* and *unfold* are instances of a more general, binary combinator known as *hylomorphism* [11], which is normally expressed using the bracketed notation $[\![\,\_,\_\,]\!]$ of (129) below to save parentheses:

$$\begin{aligned} unfold\ R &= [\![\, in, R \,]\!] \\ fold\ S &= [\![\, R, out \,]\!] \end{aligned} \tag{128}$$

As fixed points (129), hylomorphisms enjoy a number of so-called *fusion* properties, two of which are listed below for their relevance in calculations to follow [23]:

$$\begin{tikzcd} C & \mathsf{G}\,C \arrow[l, "T"'] \\ B \arrow[u, "V"] & \mathsf{G}\,B \arrow[l, "S"'] \arrow[u, "\mathsf{G}\,V"'] \\ K \arrow[u, "{[\![\, S,H \,]\!]}"] \arrow[r, "H"] & \mathsf{G}\,K \arrow[u, "{\mathsf{G}\,[\![\, S,H \,]\!]}"'] \\ A \arrow[u, "R"] \arrow[r, "U"'] & \mathsf{G}\,A \arrow[u, "\mathsf{G}\,R"'] \end{tikzcd}$$

$$[\![\, S, H \,]\!] = \langle \mu\, X\ ::\ S \cdot (\mathsf{G}\,X) \cdot H \rangle \tag{129}$$

$$V \cdot [\![\, S, H \,]\!] \subseteq [\![\, T, H \,]\!] \Leftarrow V \cdot S \subseteq T \cdot (\mathsf{G}\,V) \tag{130}$$

$$[\![\, S, H \,]\!] \cdot R = [\![\, S, U \,]\!] \Leftarrow H \cdot R = (\mathsf{G}\,R) \cdot U \tag{131}$$

[23] These and other properties of hylomorphisms arise from the powerful *μ-fusion* theorem [5] once the relational operators involved are identified as lower adjoints in GCs, recall table 1.

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*

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

which, thanks to (89), "uncurries" into $((A \rightharpoonup \mathsf{G}\,A) \times A) \rightharpoonup \mu\mathsf{G}$.

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,

$$\begin{tikzcd} \mu\mathsf{G} \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & \underbrace{(K \rightharpoonup \mathsf{G}\,K)}_{\text{“heap”}} \times K \arrow[l, bend left, "Unf"] \end{tikzcd} \qquad (132)$$

where K can be regarded as a data type of *"heap addresses"*, or *"pointers"*, and $K \rightharpoonup \mathsf{G}\,K$ a datatype of G-structured *heaps* [24]. 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 (90). 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 (62) — and to be referentially acyclic. This second requirement can also be expressed via the membership relation associated with G: relation $K \xleftarrow{\in_{\mathsf{G}} \cdot H} K$ on references must be well-founded [23].

Jourdan [39] developed a pointwise proof of the surjectiveness of Unf (132) for K isomorphic to the natural numbers and G polynomial (see more about this in section 13). The representation relation R, which should be chosen among the entire sub-relations of Unf°, is an injective *fold* (since converses of unfolds are folds [11]). Appendix A illustrates a strategy for encoding such folds, in the case of G polynomial and K the natural numbers.

"De-recursivation" law (132) generalizes, in the generic PF-style, the main result of [69] and bears some resemblance (at least in spirit) with "defunctionalization" [35], 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 useful device for cross-paradigm transformations. For instance, [56] shows how to use it in calculating a universal SQL representation for XML data.

The sections which follow will illustrate this potential, while stressing on genericity [37]. Operations of the *algebra of heaps* such as eg. *defragment* (cf. *garbage-collection*) will be stated generically and be shown to be correct with respect to the abstraction relation.

[24] Technically, this view corresponds to regarding heaps as (finite) relational G-coalgebras.

10 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` (38) 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 notation:

$$
\begin{array}{ll}
 & PTree \\
\cong_1 & \quad \{\ r_1 = \mathit{out}\ ,\ f_1 = \mathit{in}, \text{for } \mathsf{G}\,K \overset{\text{def}}{=} \mathit{Ind} \times (K+1) \times (K+1) \text{ — cf. (51, 127)}\ \} \\
 & \mu\mathsf{G} \\
\leq_2 & \quad \{\ R_2 = \mathit{Unf}^{\circ}, F_2 = \mathit{Unf} \text{ — cf. (132)}\ \} \\
 & (K \rightharpoonup \mathit{Ind} \times (K+1) \times (K+1)) \times K \\
\cong_3 & \quad \{\ r_3 = (\mathit{id} \rightharpoonup \mathit{flatr}^{\circ}) \times \mathit{id}\ ,\ f_3 = (\mathit{id} \rightharpoonup \mathit{flatr}) \times \mathit{id} \text{ — cf. (43)}\ \} \\
 & (K \rightharpoonup \mathit{Ind} \times ((K+1) \times (K+1))) \times K \\
\cong_4 & \quad \{\ r_4 = (\mathit{id} \rightharpoonup \mathit{id} \times \mathit{p2p}) \times \mathit{id}\ ,\ f_4 = (\mathit{id} \rightharpoonup \mathit{id} \times \mathit{p2p}^{\circ}) \times \mathit{id} \text{ — cf. (88)}\ \} \\
 & (K \rightharpoonup \mathit{Ind} \times (K+1)^2) \times K \\
\cong_5 & \quad \{\ r_5 = (\mathit{id} \rightharpoonup \mathit{id} \times \mathit{tot}^{\circ}) \times \mathit{id}\ ,\ f_5 = (\mathit{id} \rightharpoonup \mathit{id} \times \mathit{tot}) \times \mathit{id} \text{ — cf. (84)}\ \} \\
 & (K \rightharpoonup \mathit{Ind} \times (2 \rightharpoonup K)) \times K \\
\leq_6 & \quad \{\ r_6 = \triangle_n\ ,\ f_6 = \bowtie_n \text{ — cf. (114)}\ \} \\
 & ((K \rightharpoonup \mathit{Ind}) \times (K \times 2 \rightharpoonup K)) \times K \\
\cong_7 & \quad \{\ r_7 = \mathit{flatl}\ ,\ f_7 = \mathit{flatl}^{\circ} \text{ — cf. (44)}\ \} \\
 & (K \rightharpoonup \mathit{Ind}) \times (K \times 2 \rightharpoonup K) \times K \\
=_8 & \quad \{\ \text{since } \mathit{Ind} = \mathit{Name} \times \mathit{Birth} \text{ (51)}\ \} \\
 & (K \rightharpoonup \mathit{Name} \times \mathit{Birth}) \times (K \times 2 \rightharpoonup K) \times K
\end{array}
\tag{133}
$$

In summary:

- Step 2 moves from the functional (inductive) to the pointer-based representation. In our example, this corresponds to mapping inductive tree (9) to the heap of figure 2a.
- Step 5 starts the move from pointer-based to relational-based representation. Isomorphism (84) between *Maybe*-functions and simple relations (which is the main theme of [59]) provides the relevant data-link between the two paradigms: pointers "become" primary/foreign keys.
- Steps 7 and 8 deliver an RDBT structure (illustrated in figure 2b) 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 (133) 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 script 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 29. 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 π_1. □

11 On the Transcription Level

Our final calculations have to do with what the authors of [42] identify as the *transcription level*, the third ingredient of a *mapping scenario*. This has to do with diagram (10): once two pairs of data maps ("map forward" and "map backward") F, R and F', R' have been calculated so as to represent two source 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 (the latter *wrapped* within the relevant abstraction and representation relations) cannot be distinguished:

$$O = F' \cdot P \cdot R \tag{134}$$

Equality is, however, much too strong a requirement. In fact, there is no disadvantage in letting the target side of (134) be more defined than the source operation O, provided both are simple [25]:

$$O \subseteq F' \cdot P \cdot R \tag{135}$$

Judicious use of (29, 30) will render (135) equivalent to

$$O \cdot F \subseteq F' \cdot P \tag{136}$$

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. $F' = id$ in (136), this 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 inductive. Does this entail inductive

[25] Staying within this class of operations is still quite general: it encompasses all deterministic, possibly partial computations. Within this class, inclusion coincides with the standard definition of *operation refinement* [60].

reasoning? Let us see. Focusing on this step alone, we want to solve equation $name \cdot Unf \subseteq Hname$ for unknown $Hname$ — a query of type $((K \rightharpoonup \mathsf{G}\,K) \times K) \rightharpoonup Name$.

Simple relation currying (91) 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 as such, $name \cdot \llbracket\, in, H \,\rrbracket \subseteq \llbracket\, T, H \,\rrbracket$, so that T becomes the unknown. Then we calculate:

$$
\begin{array}{ll}
 & name \cdot \llbracket\, in, H \,\rrbracket \subseteq \llbracket\, T, H \,\rrbracket \\
\Leftarrow & \quad \{ \text{ fusion (130) } \} \\
 & name \cdot in \subseteq T \cdot \mathsf{G}(name) \\
\equiv & \quad \{ \ name \cdot Node = \pi_1 \cdot \pi_1 \text{ (127) ; expansion of } \mathsf{G}(name) \ \} \\
 & \pi_1 \cdot \pi_1 \subseteq T \cdot (id \times (name + id) \times (name + id)) \\
\Leftarrow & \quad \{ \ \pi_1 \cdot (f \times g) = f \cdot \pi_1 \ \} \\
 & T = \pi_1 \cdot \pi_1
\end{array}
$$

Thus

$$
\begin{array}{rl}
\overline{Hname}\ H = & \llbracket\, \pi_1 \cdot \pi_1, H \,\rrbracket \\
= & \quad \{ \text{ (129) } \} \\
 & \langle \mu\, X \ :: \ \pi_1 \cdot \pi_1 \cdot (id \times (X + id) \times (X + id)) \cdot H \rangle \\
= & \quad \{ \ \pi_1 \cdot (f \times g) = f \cdot \pi_1 \ \} \\
 & \langle \mu\, X \ :: \ \pi_1 \cdot \pi_1 \cdot H \rangle \\
= & \quad \{ \text{ trivia } \} \\
 & \pi_1 \cdot \pi_1 \cdot H
\end{array}
$$

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: should pointer k be successfully dereferenced in H, selection of the Ind field will take place, wherefrom the name field is finally selected (recall that $Ind = Name \times Birth$).

The exercise of mapping $Hname$ down to the SQL level (133) 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 N are the two relational tables which originated from H after step 2. $Rname$ can be encoded into SQL as something like

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

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

The main ingredient of the exercise just completed is the use of fusion property (130). 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 heap 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 (33), defragmentation is easy to model as a projection:

$$\begin{array}{l} \mathit{defragment} \ : \ (K \longrightarrow K) \longrightarrow (K \rightharpoonup \mathsf{G}\,K) \longrightarrow (K \rightharpoonup \mathsf{G}\,K) \\ \mathit{defragment}\ f\ H \stackrel{\mathrm{def}}{=} (\mathsf{G}\,f) \cdot H \cdot f^{\circ} \end{array} \tag{137}$$

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

$$t\ \mathit{Unf}\ (\mathit{defragment}\ f\ H, f\ k) \ \equiv \ t\ \mathit{Unf}\ (H, k) \tag{138}$$

Let us check (138):

$$\begin{array}{cl} & t\ \mathit{Unf}(\mathit{defragment}\ f\ H, f\ k) \ \equiv \ t\ \mathit{Unf}(H,k) \\ \equiv & \quad \{\ \text{(132) ; (128)}\ \} \\ & t\ [\![\ \mathit{in}, \mathit{defragment}\ fH\ ]\!]\ (f\ k) \ \equiv \ t\ [\![\ \mathit{in}, H\ ]\!]\ k \\ \equiv & \quad \{\ \text{go pointfree (20); definition (137)}\ \} \\ & [\![\ \mathit{in}, (\mathsf{G}\,f) \cdot H \cdot f^{\circ}\ ]\!] \cdot f = [\![\ \mathit{in}, H\ ]\!] \\ \Leftarrow & \quad \{\ \text{fusion property (131)}\ \} \\ & (\mathsf{G}\,f) \cdot H \cdot f^{\circ} \cdot f = (\mathsf{G}\,f) \cdot H \\ \Leftarrow & \quad \{\ \text{Leibniz}\ \} \\ & H \cdot f^{\circ} \cdot f = H \\ \equiv & \quad \{\ \text{since } H \subseteq H \cdot f^{\circ} \cdot f \text{ always holds}\ \} \\ & H \cdot f^{\circ} \cdot f \subseteq H \end{array}$$

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

$$k \in \mathit{dom}\ H \ \wedge \ f\ k = f\ k' \ \Rightarrow \ k' \in \ \mathit{dom}\ H \ \wedge \ H\ k = H\ k'$$

for all heap addresses k, k') is sufficient for $\mathit{defragment}$ to preserve the information stored in the heap *and* its simplicity [26]. Of course, any injective f will qualify for safe defragmentation, for *every* heap.

[26] In fact, $H \cdot f^{\circ} \cdot f \subseteq H$ ensures $H \cdot f^{\circ}$ simple, via (30) and monotonicity.

Some comments are in order. First of all, and unlike what is common in data refinement involving recursive data structures (see eg. [24] 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.)

Secondly, the $defragment$ operation we've just reasoned about is a so-called *representation changer* [34]. These operations (which include garbage collection, etc) are important because they add to efficiency without disturbing the service delivered to the client. In the *mapping scenario* terminology of [42], 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 [42].

The ubiquity of *simplicity* in data modeling, as shown throughout this paper, invites one to pay special attention to the CRUD of this kind of relation. Reference [57] identifies some "design patterns" for simple relations. The one dealt with in this paper is the *identity pattern*. For this pattern, a succinct specification of the four CRUD operations on simple M is as follows:

- $Create(N)$: $M \mapsto N \dagger M$, where (simple) argument N embodies the new entries to add to M. The use of the override operator † [38, 59] 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 Φ get updated by f; all the other remain unchanged.
- $Delete(\Phi)$: $M \mapsto M \cdot (id - \Phi)$, where $R - S$ means relational difference (cf. table 1). All entries whose keys are selected by Φ are removed.

Space constraints preclude going further on this topic in this paper. The interested reader will find in reference [57] the application of the PF-transform in speeding-up reasoning about CRUD preservation of datatype invariants on simple relations, as a particular case of the general theory [8]. Similar gains are expected from the same approach applied to CRUD transcription.

Exercise 30. Investigate the transcription of selector function `mother` (38) to the heap-and-pointer level, that is, solve $mother \cdot Unf \subseteq P$ for P. You should obtain a simple relation which, should it succeed in dereferencing the input pointer, it will follow on to the second position in the heap-cell so as to unfold (if this is the case) and show the tree accessible from that point. The so-called *hylo-computation rule* — $[\![R, S]\!] = R \cdot (\mathsf{F}\,[\![R, S]\!]) \cdot S$ — is what matters this time. □

Summary. The transcription level is the third component of a mapping scenario whereby abstract operations are "mapped forward" to the target level and give room to concrete implementations (running code). In the approach put forward in this paper, this is performed by solving an equation (134) where the unknown is the concrete implementation P one is aiming at. This section gave an example of how to carry out this task in presence of recursive data structures represented by heaps and pointers. The topic of CRUD operation transcription was also (briefly) addressed.

12 Related Work

This section addresses two areas of research which are intimately related to the data transformation discipline put forward in the current paper. One is *bidirectional programming* used to synchronize heterogeneous data formats [13]. The other is the design of term rewriting systems for type-safe data transformation [17].

Lenses. The proximity is obvious between abstraction/representation pairs implicit in $\leq$-rules and bidirectional transformations known as *lenses* and developed in the context of the classical *view-update problem* [13, 14, 27, 33]. Each lens connects a concrete data type 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.)

A lens is said to be *well-behaved* if two conditions hold,

$$get(put(v, s)) = v \qquad \text{and} \qquad put(get\ s, s) = s$$

known as *acceptability* and *stability*, respectively. For total lenses, these are easily PF-transformed into

$$put \cdot \pi_1^\circ \subseteq get^\circ \tag{139}$$

$$\langle get, id \rangle \subseteq put^\circ \tag{140}$$

which can be immediately recognized as stating the connectivity requirements of $\leq$-diagrams

$$\begin{tikzcd} & A \times C \arrow[dr, "put"] & & & & & \\ A \arrow[ur, "\pi_1^\circ"] & \leq & C \arrow[ll, bend left, "get"] & \text{and} & C \arrow[rr, bend left, "\langle get{,}id \rangle"] & \leq & A \times C \arrow[ll, bend left, "put"] \end{tikzcd} \tag{141}$$

respectively.

Proving that these diagrams hold in fact is easy to check in the PF-calculus: stability (140) enforces put surjective (of course $\langle get, id \rangle$ is injective even in case get is not). Acceptability (139) enforces get surjective since it is larger than the converse of entire $put \cdot \pi_1^\circ$ (recall rules of thumb of exercise 2). Conversely, being at most the converse of a function, $put \cdot \pi_1^\circ$ is injective, meaning that

$$\begin{aligned} & \pi_1 \cdot put^\circ \cdot put \cdot \pi_1^\circ \subseteq id \\ \equiv \quad & \{ \text{ shunting (26, 27) and adding variables } \} \\ & put(a, c) = put(a', c') \Rightarrow a = a' \end{aligned}$$

holds. This fact is known in the literature as the *semi-injectivity* of put [27].

Exercise 31. A (total, well-behaved) lens is said to be *oblivious* [27] if put is of the form $f \cdot \pi_1$, for some f. Use the PF-calculus to show that in this case get and f are bijections, that is, A and C in (141) are isomorphic [27]. Suggestion: show that $get = f^\circ$ and recall (75). □

Put side by side, the two $\leq$-diagrams displayed in (141) express the bidirectional nature of lenses in a neat way [28]. They also suggest that lenses could somehow be "programmed by calculation" in the same manner as the structural transformations investigated in the main body of this paper. See section 13 for future research directions in this respect.

2LT — a library for two-level data transformation. The 2LT package of the U.Minho Haskell libraries [10, 17, 18] applies the theory presented in the current paper to data refinement via (typed) strategic term re-writing using GADTs. The refinement process is modeled by a type-changing rewrite system, each rewrite step of which animates a $\leq$-rule of the calculus: it takes the form $A \mapsto (C, to, from)$ where C, the target type, is packaged with the conversion functions (to and $from$) between the old (A) and new type (C). By repeatedly applying such rewrite steps, complex conversion functions (data mappings) are calculated incrementally while a new type is being derived. (So, 2LT representation mappings are restricted to functions.)

Data mappings obtained after type-rewriting can be subject to subsequent simplification using laws of PF program calculation. Such simplifications include migration of queries on the source data type to queries on a target data type by fusion with the relevant data mappings (a particular case of transcription, as we have seen). Further to PF functional simplification, 2LT implements rewrite techniques for transformation of structure-shy functions (XPath expressions and strategic functions), see eg. [18].

In practice, 2LT can be used to scale-up the data transformation/mapping techniques presented in this paper to real-size case-studies, mainly by mechanizing repetitive tasks and discharging housekeeping duties. More information can be gathered from the project's website: `http://code.google.com/p/2lt`.

13 Conclusions and Future Work

This paper presented a mathematical approach to 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 relation pointfree notation to express *both* algorithms

[27] This is Lemma 3.9 in [27], restricted to functions.

[28] Note however that, in general, lenses are not entire [27].

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 *uniformly* PF-transform both programs *and data* are already apparent at practical level, see eg. the work reported in [50].

Thanks to the PF-transform, opportunities for creativity steps are easier to spot and carry out with less symbol trading. 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 [29]. 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. The items listed below are proposed as prompt topics for research.

Lenses. The pointwise treatment of lenses as partial functions in [27] is *cpo*-based, entailing the need for continuity arguments. In this paper we have seen that partial functions are *simple* relations easily accommodated in the binary relation calculus. At first sight, generalizing put and get of section 12 from functions to simple relations doesn't seem to be particularly hard, even in the presence of recursion, thanks to the PF hylomorphism calculus (recall section 9).

How much the data mapping formalism presented in the current paper can offer to the theory of bidirectional programming is the subject of on-going research.

Heaps and pointers at target. We believe that Jourdan's long, inductive pointwise argument [39] for $\leq$-law (132) can be supplanted by succinct pointfree calculation if results developed meanwhile by Gibbons [29] are taken into account. Moreover, the same law should be put in parallel with other related work on calculating with pointers (read eg. [12] and follow the references).

Separation logic. Law (132) has a clear connection to shared-mutable data representation and thus with *separation logic* [62]. There is work on a PF-relational model for this logic [64] which is believed to be useful in better studying and further generalizing law (132) and to extend the overall approach to in-place data-structure updating.

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 7 and detailed in [63] are the subject of current research taking into account the PF-calculus of invariants of [8]. Moreover, $\leq$-rules should be able to take invariants into account (a topic suggested but little developed in [55]).

Mapping scenarios for the UML. Following the exercise of section 8, a calculational theory of UML mapping scenarios could be developed starting from eg. K. Lano's catalogue [43]. This should also take the *Calculating with Concepts* [22] semantics for UML class diagrams into account. For preliminary work on this subject see eg. [9].

[29] The $\leq$-rules of the calculus are used in practical classes and lab assignments in the derivation of database schemas from abstract models, including the synthesis of data mappings. The proofs of such rules (as given in the current paper) are addressed in the theory classes.

PF-transform. Last but not least, we think that further research on the PF-transform should go along with applying it in practice. In particular, going further and formalizing the analogy with the Laplace transform (which so far has only been hinted at) would be a fascinating piece of research in mathematics and computer science in itself, and one which would *put the vast storehouse in order*, to use the words of Lawvere and Schanuel [44]. In these times of widespread pre-scientific software technology, putting the PF-transform under the same umbrella as other mathematical transforms would contribute to better framing the software sciences within engineering mathematics as a whole.

Acknowledgments

The author wishes to thank his colleagues at Minho University and his (current and former) students for the warm reception to his (ever evolving) ideas on data calculation. Special thanks go to L.S. Barbosa, to C.J. Rodrigues, to J.C. Ramalho and to the 2LT team *core*: Alcino Cunha, Joost Visser, Tiago Alves and Hugo Pacheco. Jeremy Gibbons comments on the proceedings version of this paper are gratefully acknowledged.

The author is also indebted to the anonymous referees for detailed and helpful comments which improved the paper's presentation and technical contents.

Part of this research was carried out in the context of the PURE Project *(Program Understanding and Re-engineering: Calculi and Applications)* funded by FCT contract `POSI/ICHS/44304/2002`.

References

1. Aarts, C., Backhouse, R.C., Hoogendijk, P., Voermans, E., van der Woude, J.: A relational theory of datatypes (December 1992), `http://www.cs.nott.ac.uk/~rcb`
2. Alves, T.L., Silva, P.F., Visser, J., Oliveira, J.N.: Strategic term rewriting and its application to a VDM-SL to SQL conversion. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 399–414. Springer, Heidelberg (2005)
3. Ambler, S.W.: The object-relational impedance mismatch (February15, 2006), `http://www.agiledata.org/essays/impedanceMismatch.html`
4. Backhouse, K., Backhouse, R.C.: Safety of abstract interpretations for free, via logical relations and Galois connections. SCP 15(1–2), 153–196 (2004)
5. Backhouse, R.C.: Mathematics of Program Construction, pages 608. Univ. of Nottingham (2004); Draft of book in preparation
6. Backhouse, R.C., de Bruin, P., Hoogendijk, P., Malcolm, G., Voermans, T.S., van der Woude, J.: Polynomial relators. In: AMAST 1991, pp. 303–362. Springer, Heidelberg (1992)
7. Backus, J.: Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. CACM 21(8), 613–639 (1978)
8. Barbosa, L.S., Oliveira, J.N., Silva, A.M.: Calculating invariants as coreflexive bisimulations. LNCS, vol. 5140, pp. 83–99. Springer, Heidelberg (2008)
9. Berdaguer, P.: Algebraic representation of UML class-diagrams, May, Dept. Informatics, U.Minho. Technical note (2007)
10. Berdaguer, P., Cunha, A., Pacheco, H., Visser, J.: Coupled Schema Transformation and Data Conversion For XML and SQL. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 290–304. Springer, Heidelberg (2006)

11. Bird, R., de Moor, O.: Algebra of Programming. C.A.R. Hoare, series editor, Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1997)
12. Bird, R.S.: Unfolding pointer algorithms. J. Funct. Program. 11(3), 347–358 (2001)
13. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: ACM SIGPLAN–SIGACT POPL Symposium, pp. 407–419 (January 2008)
14. Bohannon, A., Vaughan, J.A., Pierce, B.C.: Relational lenses: A language for updateable views. In: Principles of Database Systems (PODS) (2006)
15. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley Longman, Amsterdam (1999)
16. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. JACM 24(1), 44–67 (1977)
17. Cunha, A., Oliveira, J.N., Visser, J.: Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 284–289. Springer, Heidelberg (2006)
18. Cunha, A., Visser, J.: Transformation of structure-shy programs: applied to XPath queries and strategic functions. In: PEPM 2007, pp. 11–20. ACM, New York (2007)
19. Darlington, J.: A synthesis of several sorting algorithms. Acta Informatica 11, 1–30 (1978)
20. de Roever, W.-P., Engelhardt, K., Coenen, J., Buth, K.-H., Gardiner, P., Lakhnech, Y., Stomp, F.: Data Refinement Model-Oriented Proof methods and their Comparison. Cambridge University Press, Cambridge (1999)
21. Deutsch, M., Henson, M., Reeves, S.: Modular reasoning in Z: scrutinising monotonicity and refinement (to appear, 2006)
22. Dijkman, R.M., Pires, L.F., Joosten, S.: Calculating with concepts: a technique for the development of business process support. In: pUML. LNI, vol. 7, pp. 87–98. GI (2001)
23. Doornbos, H., Backhouse, R., van der Woude, J.: A calculational approach to mathematical induction. Theoretical Computer Science 179(1–2), 103–135 (1997)
24. Fielding, E.: The specification of abstract mappings and their implementation as B^{+}-trees. Technical Report PRG-18, Oxford University (September 1980)
25. Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques for Software Development, 1st edn. Cambridge University Press, Cambridge (1998)
26. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Proc. Symposia in Applied Mathematics Mathematical Aspects of Computer Science, vol. 19, pp. 19–32. American Mathematical Society (1967)
27. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst 29(3), 17 (2007)
28. Frias, M.F.: Fork algebras in algebra, logic and computer science. Logic and Computer Science. World Scientific Publishing Co, Singapore (2002)
29. Gibbons, J.: When is a function a fold or an unfold?, Working document 833 FAV-12 available from the website of IFIP WG 2.1, 57th meeting, New York City, USA (2003)
30. Hainaut, J.-L.: The transformational approach to database engineering. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 95–143. Springer, Heidelberg (2006)
31. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196 (1986)
32. Hoogendijk, P.: A Generic Theory of Data Types. PhD thesis, University of Eindhoven, The Netherlands (1997)
33. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Proc. ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 178–189. ACM Press, New York (2004)

34. Hutton, G., Meijer, E.: Back to basics: Deriving representation changers functionally. Journal of Functional Programming (1993) (Functional Pearl)
35. Hutton, G., Wright, J.: Compiling exceptions correctly. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 211–227. Springer, Heidelberg (2004)
36. Jackson, D.: Software abstractions: logic, language, and analysis. The MIT Press, Cambridge Mass (2006)
37. Jeuring, J., Jansson, P.: Polytypic programming. In: Advanced Functional Programming. Springer, Heidelberg (1996)
38. Jones, C.B.: Systematic Software Development Using VDM, 1st edn. Series in Computer Science. Prentice-Hall Int., Englewood Cliffs (1986)
39. Jourdan, I.S.: Reificação de tipos abstractos de dados: Uma abordagem matemática. Master's thesis, University of Coimbra (1992) (in Portuguese)
40. Kahl, W.: Refinement and development of programs from relational specifications. ENTCS 4, 1–4 (2003)
41. Kreyszig, E.: Advanced Engineering Mathematics, 6th edn. J. Wiley & Sons, Chichester (1988)
42. Lämmel, R., Meijer, E.: Mappings make data processing go round. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 169–218. Springer, Heidelberg (2006)
43. Lano, K.: Catalogue of model transformations, `http://www.dcs.kcl.ac.uk/staff/kcl/`
44. Lawvere, B., Schanuel, S.: Conceptual Mathematics: a First Introduction to Categories. Cambridge University Press, Cambridge (1997)
45. Maier, D.: The Theory of Relational Databases. Computer Science Press (1983)
46. McCarthy, J.: Towards a mathematical science of computation. In: Popplewell, C.M. (ed.) Proc. IFIP 62, pp. 21–28. North-Holland Pub.Company, Amsterdam (1963)
47. McLarty, C.: Elementary Categories, Elementary Toposes, 1st edn. Oxford Logic Guides, vol. 21. Calendron Press, Oxford (1995)
48. Meng, S., Barbosa, L.S.: On refinement of generic state-based software components. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 506–520. Springer, Heidelberg (2004) (Best student co-authored paper award)
49. Morgan, C.: Programming from Specification. C.A.R. Hoare, series (ed.), Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1990)
50. Necco, C., Oliveira, J.N., Visser, J.: Extended static checking by strategic rewriting of pointfree relational expressions. Technical Report FAST:07.01, CCTC Research Centre, University of Minho (2007)
51. Oliveira, J.N.: Refinamento transformacional de especificaşões (terminais). In: Proc. of XII Jornadas Luso-Espanholas de Matemática, vol. II, pp. 412–417 (May 1987)
52. Oliveira, J.N.: A Reification Calculus for Model-Oriented Software Specification. Formal Aspects of Computing 2(1), 1–23 (1990)
53. Oliveira, J.N.: Invited paper: Software Reification using the SETS Calculus. In: Denvir, T., Jones, C.B., Shaw, R.C. (eds.) Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK, pp. 140–171. Springer, Heidelberg (1992)
54. Oliveira, J.N.: Data processing by calculation. In: 6th Estonian Winter School in Computer Science, Palmse, Estonia, March 4-9, 2001. Lecture notes, pages 108 (2001)
55. Oliveira, J.N.: Constrained datatypes, invariants and business rules: a relational approach, PUReCafé, DI-UM, 2004.5.20 [talk], $\mathrm{PUR}_{\mathrm{E}}$ Project (POSI/CHS/44304/2002) (2004)
56. Oliveira, J.N.: Calculate databases with simplicity, Presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK (September 2004) (Slides available from the author's website)
57. Oliveira, J.N.: Reinvigorating pen-and-paper proofs in VDM: the pointfree approach. In: The Third OVERTURE Workshop, Newcastle, UK, 27-28 November (2006)

58. Oliveira, J.N.: Pointfree foundations for (generic) lossless decomposition (submitted, 2007)
59. Oliveira, J.N., Rodrigues, C.J.: Transposing relations: from Maybe functions to hash tables. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 334–356. Springer, Heidelberg (2004)
60. Oliveira, J.N., Rodrigues, C.J.: Pointfree factorization of operation refinement. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085. pp. 236–251. Springer, Heidelberg (2006)
61. Pratt, V.: Origins of the calculus of binary relations. In: Proc. of the 7th Annual IEEE Symp. on Logic in Computer Science, pp. 248–254. IEEE Computer Society Press, Los Alamitos (1992)
62. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74 (2002)
63. Rodrigues, C.J.: Software Refinement by Calculation. PhD thesis, Departamento de Informática, Universidade do Minho (submitted, 2007)
64. Wang, S., Barbosa, L.S., Oliveira, J.N.: A relational model for confined separation logic. In: TASE 2008, The 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering, June 17 - 19. LNCS. Springer, Heidelberg (2008)
65. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program 7(3), 231–264 (1997)
66. Sheard, T., Pasalic, E.: Two-level types and parameterized modules. Journal of Functional Programming 14(5), 547–587 (2004)
67. Thomas, D.: The impedance imperative tuples + objects + infosets =too much stuff! Journal of Object Technology 2(5) (September/ October 5, 2003)
68. Visser, J.: Generic Traversal over Typed Source Code Representations. Ph. D. dissertation, University of Amsterdam, Amsterdam, The Netherlands (2003)
69. Wagner, E.G.: All recursive types defined using products and sums can be implemented using pointers. In: Bergman, C., Maddux, R.D., Pigozzi, D. (eds.) Algebraic Logic and Universal Algebra in Computer Science. LNCS, vol. 425. Springer, Heidelberg (1990)
70. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Inc., Upper Saddle River (1996)

A PTree Example in Haskell

This annex presents the exercise, in Haskell, of representing inductive type `PTree` (38) 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
```

It is convenient to regard this datatype as a bifunctor [30]:

```
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')
```

[30] Note the sugaring of pairing in terms of the infix combinator `x |-> y = (x,y)`, as suggested by (33). 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`.

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
```

which generate the kth even and odd numbers. Functorial renaming of heap addresses via these functions (whose ranges are disjoint) ensure that the heaps one is joining (via list concatenation) are *separate* [62, 64]. This representation technique is reminiscent of that of storing "binary heaps" (which are not quite the same as in this paper) as arrays without pointers [31]. It can be generalized to any polynomial type of degree n by building n-functions $f_i\ k \overset{\text{def}}{=} nk + i$, for $0 \leq i < n$.

Finally, the abstraction relation is encoded as a partial function in Haskell as follows:

```
f (Heap h k) = let Just (a,x,y) = lookup k h
               in  Node (fst a)(snd a)
                        (fmap (f . Heap h) x)
                        (fmap (f . Heap h) y)
```

[31] See eg. entry Binary_heap in the Wikipedia.