Transposing Relations: from *Maybe* Functions to Hash Tables

J.N. Oliveira and C.J. Rodrigues

Techn. Report DI-PURe-04.02.02

PURE
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-04.02.02
Transposing Relations: from Maybe *Functions to Hash Tables* by J.N. Oliveira and C.J. Rodrigues

Abstract

Functional transposition is a technique for converting relations into functions aimed at developing the relational algebra *via* the algebra of functions.

This paper attempts to develop a basis for *generic transposition*. Two well-known instances of this construction are considered, one applicable to any relation and the other applicable to simple relations only.

Our illustration of the usefulness of the generic transpose takes advantage of the *free theorem* of a polymorphic function. We show how to derive laws of relational combinators as free theorems of their transposes. Finally, we relate the topic of functional transposition with the *hashing* technique for data representation.

1 Introduction

This paper is concerned with techniques for functional transposition of binary relations. By functional transposition we mean the *faithful* representation of a relation by a (total) function. But — what is the purpose of such a representation?

Functions are well-known in mathematics and computer science. The functional intuition traverses mathematics from end to end because it has a solid semantics rooted on a clear-cut mathematical structure — the category of "all" sets and set-theoretical functions. Functions have arich theory. For instance, they can be dualized (as happens e.g. with the projection/ injection functions), they can be Galois connected (as happens e.g. with inverse functions) and they can be parametrically polymorphic. In the latter case, they exhibit theorems "for free" [22] which can be inferred solely by inspection of their type (thus providing semantics to functional APIs in which function bodies are intentionally hidden from the user).

However, (total) functions are not enough. In many situations, functions are *partial* in the sense that they are undefined for some of their input data. Programmers have learned to deal with this situation by enriching the codomain of such functions with a special error mark indicating that *nothing* is output. In C/C++, for instance, this leads to functions which output *pointers* to values rather than just values. In functional languages such as Haskell [8], this leads to functions which output $Maybe$-values rather than values, where $Maybe$ is datatype $Maybe\ a = Nothing \mid Just\ a$. In a different context, *finite* partial functions are represented by sets of pairs in which no first element (in each pair) is repeated. In database theory, these data-sets are called *functional* data *dependencies* [7] while in formal modelling notations such as VDM [9] or Z [21] they are called *finite mappings*, an abstract datatype of widespread use in formal specification.

Partial functions are still not enough because one very often wants to describe *what* is required of a function rather than prescribe *how* the function should compute its result. A well-known example is *sorting*: sorting a list amounts to finding an ordered permutation of the list *independently* of the particular sorting algorithm eventually chosen to perform the task (eg. quicksort, mergesort, etc.). So one is concerned not only with *implementations* but also with *specifications*, which can be vague (eg. which square root is meant when one writes "$\sqrt{x}$"?) and non-deterministic. Again, functional programmers have learned to cope with this by structuring the codomain of such functions as *sets* or *lists* of values, a strategy which can be animated in case such sets or lists are finite (bounded non-determinism).

In general, such powerset valued functions are models of binary relations: for each such f one may define the binary relation R such that bRa means $b \in (f\ a)$ suitably typed for all a and b. Such R is unique for the given f. Conversely, any binary relation R is *uniquely* transposed into a set-valued function f. The existence and uniqueness of such a transformation leads to the identification of a *transpose* operator Λ [6] satisfying the following *universal property*:

$$f = \Lambda\ R \equiv (bRa \equiv b \in f\ a) \tag{1}$$

The power-transpose operator Λ establishes an isomorphism between relations and set-valued functions which is well-known in mathematics and is often exploited in the algebra of relations. For instance, a significant part of the contents of textbook [6] draws from such an isomorphism.

Less popular and usually not identified as a transpose is the conversion of a partial function into a $Maybe$-valued function, for which one can identify, by analogy with (1), isomorphism Γ defined by

$$f = \Gamma\ R \equiv (bRa \equiv (f\ a = Just\ b)) \tag{2}$$

where R ranges over partial functions.

Terms *total* and *partial* are avoided in relation algebra because they clash with a different meaning in the context of *partial orders* and *total orders*, which are other special cases of relations. Instead, one writes *entire* for *total* and *simple relation* is written instead of *partial function*. The word *function* is reserved for total, simple relations which find a central place in the taxonomy of binary relations depicted in Fig. 1 (the other entries in the taxonomy will be explained later on).

Paper objectives. This paper is structured around three main ideas. First, we recall that Λ is not the only operator for transposing relations. It certainly is the most general, but we will identify other such operators as we go down the hierarchy of binary relations. Such transposes have to do with the (generic) notion of *membership*. (This extends "$\in$" to collective types other than the powerset [6, 12].) In particular, one of these operators will be related with the technique of representing finite data collections by *hash-tables*, which are efficient data-structures well-known in computer science [23, 13].

Second, we want to stress on the usefulness of transposing relations by exploiting the calculation power of functions, namely *free theorems*. Such powerful reasoning devices can be applied to relations provided we represent relations as functions (by functional transposition), reason functionally and come back to relations where appropriate. In fact, several relational combinators studied in

[6] arise from the definition of the *power-transpose* $\mathcal{P}B \xleftarrow{\Lambda\ R} A$ of a relation R. ($\mathcal{P}B$ denotes the set of all subsets of B.) However, some results could have been produced as free-theorems, as we will show in the sequel.

Last but not least, we want to provide evidence of the practicality of the *pointfree* relation calculus. The fact that pointfree notation abstracts from "points" or variables makes the reasoning more compact and effective [1]. This is apparent in our final section on hash-tables, if compared with its pointwise counterpart which one of the authors did several years ago [16]: notation and reasoning are simpler and easier to follow.

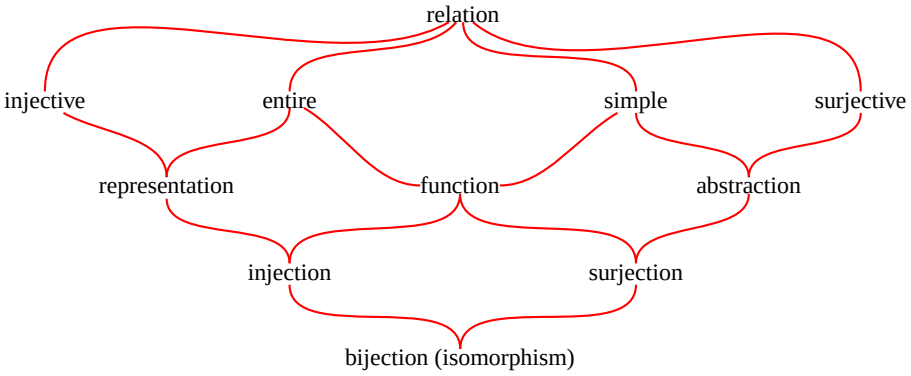

Fig. 1. Binary relation taxonomy

Related work. Equations (1) and (2) are well-known in the literature, although they have been dealt with in different contexts. While (1) is adopted as "the" standard transpose in [6], for instance, (2) is studied in [11] as an example of an *adjunction* between the categories Tot of total functions and Par of partial ones. From the literature on the related topic of *generic membership* we select [6] and the chapter of [12] devoted to the subject.

Paper structure. This paper is structured as follows. In the next section we present an overview of (pointfree) relation algebra. Section 3 presents our relational study of generic transpose. In section 4 two well-known transposes are framed in the generic view. Section 5 presents an example of reasoning based

[1] The move from the *pointwise* level (involving operators as well as variable symbols, logical connectives, quantifiers, etc.) to the *pointfree* one is compared elsewhere [18] to the *Laplace transformation*. The former is more intuitive but harder to reason about, the latter is less descriptive but more algebraic and compact. As in traditional mathematics, there is room for both in program calculation.

on the generic transpose operator and its instances. In the remainder of the paper we relate the topic of functional transposition with the *hash table* technique for data representation and draw some conclusions which lead to plans for future work.

2 Overview of the relational calculus

Relations. Let $B \xleftarrow{R} A$ denote a binary relation on datatypes A (source) and B (target). We write bRa to denote that pair (b, a) is in R. The underlying partial order on relations will be written $R \subseteq S$, meaning that S is either more defined or less deterministic than R, that is, $R \subseteq S \equiv bRa \Rightarrow bSa$ for all a, b. Equality on relations can be established by $\subseteq$-antisymmetry: $R = S \equiv R \subseteq S \wedge S \subseteq R$.

Relations can be combined by three basic operators: composition $(R \cdot S)$, converse (R°) and meet $(R \cap S)$. Meet corresponds to set-theoretical intersection and composition is defined in the usual way: $b(R \cdot S)c$ holds wherever there exists some $a \in A$ such that $bRa \wedge aSc$. 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 $B \xleftarrow{R} A$ admits 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.

Coreflexives. 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 $id_A \subseteq R$ holds and as *coreflexive* iff $R \subseteq id_A$ holds. As a rule, subscripts are dropped wherever types are implicit or easy to infer.

Coreflexive relations are fragments of the identity relation which can be used to model predicates or sets. The meaning of a *predicate* p is the coreflexive $[\![p]\!]$ such that $b[\![p]\!]a \equiv b = a \wedge (p\ a)$, that is, the relation that maps every a which satisfies p (and only such a) onto itself. The meaning of a *set* $S \subseteq A$ is $[\![\lambda a.a \in S]\!]$, that is, $b[\![S]\!]a \equiv b = a \wedge a \in S$. Wherever clear from the context, we will omit the $[\![\ ]\!]$ brackets.

Orders. Preorders are reflexive, transitive relations, where R is transitive iff $R \cdot R \subseteq R$ holds. Partial orders are anti-symmetric preorders, where R is anti-symmetric wherever $R \cap R^\circ \subseteq id$ holds, for R° the *converse* of R, that is, the relation such that $a(R^\circ)b \equiv bRa$ holds. A preorder R is an *equivalence* if it is symmetric, that is, if $R = R^\circ$. Fig. 2 depicts this taxonomy of orders, where words *partial* and *total* have the usual meaning. A total order R is a connected preorder, where R is connected iff $R \cup R^\circ = \top$ holds. $\cup$ is the join of two

relations and $\top$ is the largest relation of its type. Its dual is $\bot$, the smallest such relation.

Converse is of paramount importance in establishing a wider taxonomy of binary relations. Let us first define two derived operators, *kernel*

$$ker\ R \stackrel{\text{def}}{=} R^\circ \cdot R \tag{3}$$

and *image* (its dual)

$$img\ R \stackrel{\text{def}}{=} ker\ (R^\circ) \tag{4}$$

An alternative to (4) is to define $img\ R = R \cdot R^\circ$, since converse commutes with composition, $(R \cdot S)^\circ = S^\circ \cdot R^\circ$ and is involutive, that is, $(R^\circ)^\circ = R$. Kernel and image lead to the following terminology: a relation R is said to be *entire* (or total) iff its kernel is reflexive; or *simple* (or functional) iff its image is coreflexive. (So, simplicity is the dual of entirety.) Dually, R is *surjective* iff R° is entire, and R is *injective* iff R° is simple. This terminology is captured by the following summary table:

	Reflexive	*Coreflexive*
ker R	entire R	injective R
img R	surjective R	simple R

(5)

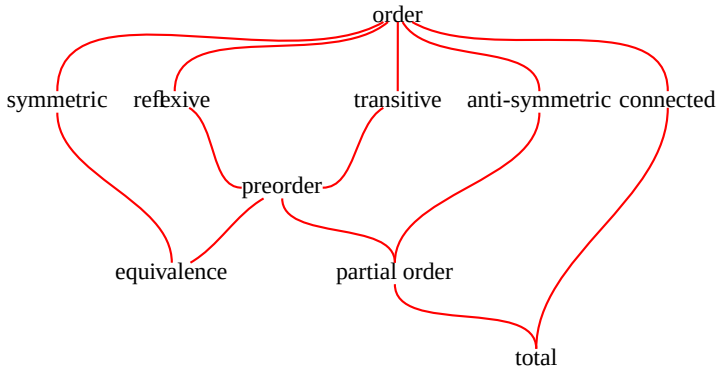

Fig. 2. Order taxonomy

Functions. A relation is a *function* iff it is both simple and entire. Functions will be denoted by lowercase letters (f, g, etc.) and are such that bfa means $b =$

$f\ a$. Function converses enjoy a number of properties of which the following is singled out because of its rôle in pointwise-pointfree conversion [3] :

$$b(f^\circ \cdot R \cdot g)a \equiv (f\ b)R(g\ a) \tag{6}$$

The overall taxonomy of binary relations is pictured in Fig. 1 where, further to the standard classification, we add *representations* and *abstractions*. These are classes of relations useful in data-refinement [15]. Because of $\subseteq$-antisymmetry, $img\, R = id$ wherever R is an *abstraction* and $ker\, R = id$ wherever R is an *representation*. This ensures that "no confusion" arises in a representation and that all abstract data are reachable ("no junk").

Isomorphisms (such as Λ and Γ above) are functions, abstractions and representations at the same time. A particular isomorphism is id, which also is the smallest equivalence on a particular data domain, that is, $b\ id\ a$ is the same as $b = a$. So id can be found in both Fig. 1 and Fig. 2.

Functions and relations. The interplay between functions and relations is a rich part of the relation calculus. This arises when one relates the arguments and results of pairs of functions f and g in, essentially, two ways:

$$f \cdot S \subseteq R \cdot g \tag{7}$$

$$f^\circ \cdot S = R \cdot g \tag{8}$$

As we shall see shortly, (7) is equivalent to $S \subseteq f^\circ \cdot R \cdot g$ which, by (6), means that f and g produce R-related outputs $f\ b$ and $g\ a$ provided their inputs are S-related (bSa). This situation is so frequent that one says that, everywhere f and g are such that (7) holds, f is $(R \leftarrow S)$*-related* to g:

$$f\,(R \leftarrow S)\,g \equiv f \cdot S \subseteq R \cdot g \quad \text{cf. diagram} \quad \begin{array}{ccc} B & \xleftarrow{S} & A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ C & \xleftarrow[R]{} & D \end{array} \tag{9}$$

For instance, for partial orders $R, S := \leq, \sqsubseteq$, fact $f\,(\leq \leftarrow \sqsubseteq)\,f$ means that f is monotone. For $R, S := \leq, id$, fact $f\,(\leq \leftarrow id)\,g$ means

$$f \dot{\leq} g \equiv f \subseteq\ \leq \cdot g \tag{10}$$

that is, f and g are such that $f\ b \leq g\ b$ for all b (the pointwise ordering $\leq$ is lifted to the functional level). In general, relation $R \leftarrow S$ will be referred to as

"Reynolds arrow combinator" (see section 5), which is extensively studied in [3].

Concerning the other way to combine relations with functions, equality (8) becomes interesting wherever R and S are preorders:

$$f^\circ \cdot \sqsubseteq = \leq \cdot g \quad \text{cf. diagram:} \quad \begin{array}{ccc} \leq & & \sqsubseteq \\ \circlearrowleft & \overset{f}{\underset{g}{\rightleftarrows}} & \circlearrowright \\ B & & C \end{array} \tag{11}$$

In this case functions f, g are always monotone and are 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 (11) via (6) we obtain

$$(f\ b) \sqsubseteq a \equiv b \leq (g\ a) \tag{12}$$

Note that (11) boils down to $f^\circ = g$ (ie. $f = g^\circ$) wherever $\leq$ and $\sqsubseteq$ are id, in which case f and g are isomorphisms, that is, f° is also a function and $f\ b = a \equiv b = f^\circ a$ holds.

For further details on the rich theory of Galois connections and examples of application see [1, 3]. Galois connections in which the two preorders are relation inclusion ($\leq, \sqsubseteq\ := \subseteq, \subseteq$) are particularly interesting because the two adjoints are relational combinators and the connection itself is their universal property. The following table lists connections which are relevant for this paper:

$(f\ X) \subseteq Y \equiv X \subseteq (g\ Y)$			
Description	f	g	**Obs.**
Converse	$(\_)^\circ$	$(\_)^\circ$	
Left-division	$(\cdot R)$	$(\ /\ R)$	
Right-division	$(R \cdot)$	$(R \setminus\ )$	
Shunting rule	$(f \cdot)$	$(f^\circ \cdot)$	NB: f is a function
"Converse" *shunting* rule	$(\cdot f^\circ)$	$(\cdot f)$	NB: f is a function
Difference	$(\_ - R)$	$(R \cup\ )$	

(13)

From the two of these called *shunting rules* one infers the very useful fact that equating functions is the same as comparing them in either way:

$$f = g \equiv\ f \subseteq g \equiv g \subseteq f \tag{14}$$

Membership. Equation (1) involves the set-theoretic membership relation $A \xleftarrow{\in} \mathcal{P}A$. Sentence $a \in x$ (meaning that "a belongs to x" or "a occurs in x") can be generalized to x other that sets. For instance, one may check whether a particular integer occurs in one ore more leafs of a binary tree, or of any other *collective* or *container* type F.

Such a generic membership relation will have type $A \xleftarrow{\in} \mathsf{F}\,A$, where F is a type *parametric on* A. Technically, the parametricity of F is captured by regarding it as a *relator* [5], a concept which extends *functors* to relations: $\mathsf{F}\,A$ describes a parametric type while $\mathsf{F}\,R$ is a relation from $\mathsf{F}\,A$ to $\mathsf{F}\,B$ provided R is a relation from A to B. Relators are monotone and commute with composition, converse and the identity:

$$\mathsf{F}\,(R \cdot S) = (\mathsf{F}\,R) \cdot (\mathsf{F}\,S) \tag{15}$$

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

$$\mathsf{F}\,id = id \tag{17}$$

Themost simple relators are the *identity* relator Id, which is such that $\mathsf{Id}\,A = A$ and $\mathsf{Id}\,R = R$, and the *constant* relator K (for a particular concrete data type K) which is such that $\mathsf{K}\,A = K$ and $\mathsf{K}\,R = id_K$.

Relators can also be multi-parametric. Two well-known examples of binary relators are product and sum,

$$R \times S = \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \tag{18}$$

$$R + S = [i_1 \cdot R, i_2 \cdot S] \tag{19}$$

where π_1, π_2 denote the projection functions of a Cartesian product, i_1, i_2 denote the injection functions of a disjoint union, and the *split/either* relational combinators are defined by

$$\langle R, S \rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \tag{20}$$

$$[R, S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \tag{21}$$

By putting together these four kinds of relator (product, sum, identity and constant) one is able to specify a large class of parametric structures — called *polynomial* — such as those implementable in Haskell. For instance, the $Maybe\,a$ datatype is an implementation of polynomial relator $\mathsf{F} = \mathsf{Id}+1$ (ie. $\mathsf{F}\,A = A+1$), where 1 denotes the *singleton* datatype, written $()$ in Haskell.

There is more than one way to generalize $A \xleftarrow{\in} \mathcal{P}A$ to relators other than the powerset. (For a thorough presentation of the subject see chapter 4 of

[12].) For the purpose of this paper it will be enough to say that $A \xleftarrow{\in_{\mathsf{F}}} \mathsf{F}\, A$, if it exists, is a *lax natural transformation* [6], that is,

$$\in_{\mathsf{F}} \cdot \mathsf{F}\, R \subseteq R \cdot \in_{\mathsf{F}} \tag{22}$$

holds. Moreover, *polynomial* relators have membership defined inductively as follows:

$$\in_{\mathsf{K}} \stackrel{\text{def}}{=} \bot \tag{23}$$

$$\in_{\mathsf{Id}} \stackrel{\text{def}}{=} id \tag{24}$$

$$\in_{\mathsf{F}\times\mathsf{G}} \stackrel{\text{def}}{=} (\in_{\mathsf{F}} \cdot \pi_1) \cup (\in_{\mathsf{G}} \cdot \pi_2) \tag{25}$$

$$\in_{\mathsf{F}+\mathsf{G}} \stackrel{\text{def}}{=} [\in_{\mathsf{F}}, \in_{\mathsf{G}}] \tag{26}$$

3 A study of generic transposition

Thanks to rule (6), it is easy to remove variables b and a from transposition rules (1) and (2), yielding

$$f = \Lambda\ R \equiv (R = \in \cdot f) \tag{27}$$

$$f = \Gamma\ R \equiv (R = i_1^\circ \cdot f) \tag{28}$$

where, in the second equivalence, R ranges over simple relations and $Just$ is replaced by injection i_1 associated with relator $\mathsf{Id} + 1$. In turn, f and R can also be abstracted from (27,28) using the same rule, whereby we end up with

$$\Lambda = (\in\cdot)^\circ$$
$$\Gamma = (i_1^\circ\cdot)^\circ$$

because "$=$" is the same as "id".

The generalization of both equations starts from the observation that, in the same way $\in$ is the membership relation associated with the powerset, i_1° is the membership relation associated with $\mathsf{Id} + 1$, as can be easily checked:

$$\begin{array}{ll} & \in_{\mathsf{Id}+1} \\ = & \quad \{\ \text{by (26)}\ \} \\ & [\in_{\mathsf{Id}}, \in_1] \\ = & \quad \{\ \text{by (24) and (23)}\ \} \\ & [id, \bot] \end{array} \tag{29}$$

$$
\begin{array}{ll}
= & \{ \text{ by (21) and properties of } \bot \} \\
& id \cdot i_1^\circ \\
= & \{ \text{ identity } \} \\
& i_1^\circ
\end{array}
$$

This suggests the definitions and results which follow.

Definition. Given a relator F with membership relation $\in_{\mathsf{F}}$, we will say that R is F-*transposable* iff the following universal property holds

$$f = \Gamma_{\mathsf{F}} R \equiv \in_{\mathsf{F}} \cdot f = R \qquad \text{cf. diagram} \qquad \begin{array}{ccc} A & \xleftarrow{R} & B \\ {\scriptstyle \in_{\mathsf{F}}}\uparrow & \swarrow_{f} & \\ \mathsf{F}\, A & & \end{array} \tag{30}$$

where function Γ_{F}, if it exists, is called the F-*transpose*.

In other words, such a generic F-transpose operator is the converse of membership pre-composition:

$$\Gamma_{\mathsf{F}} = (\in_{\mathsf{F}} \cdot)^\circ \tag{31}$$

The two instances we have seen of (30) are the power-transpose ($\mathsf{F}\, A = \mathcal{P} A$) and the $Maybe$-transpose ($\mathsf{F}\, A = A + 1$). While the former is known to be applicable to every relation [6], the latter is only applicable to simple relations, a result to be justified shortly. Prior to checking it, we review the main properties of generic transposition. These extend those presented in [6] for the power-transpose.

Properties. Cancellation and reflection

$$\in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} R = R \tag{32}$$

$$\Gamma_{\mathsf{F}} \in_{\mathsf{F}} = id \tag{33}$$

arise from (30) by substitutions $f := \Gamma_{\mathsf{F}} R$ and $f := id$, respectively. Fusion

$$\Gamma_{\mathsf{F}} (T \cdot S) = (\Gamma_{\mathsf{F}} T) \cdot S \Leftarrow (\Gamma_{\mathsf{F}} T) \cdot S \text{ is a function} \tag{34}$$

arises in the same way — this time for substitution $f := (\Gamma_{\mathsf{F}} T) \cdot S$ — as follows (assuming the side condition ensuring that $(\Gamma_{\mathsf{F}} T) \cdot S$ is a function):

$$(\Gamma_{\mathsf{F}} T) \cdot S = \Gamma_{\mathsf{F}} R \equiv \in \cdot ((\Gamma_{\mathsf{F}} T) \cdot S) = R$$

$$
\begin{array}{ll}
\equiv & \{\ \text{associativity}\ \} \\
& (\in \cdot \Gamma_{\mathsf{F}} T) \cdot S = R \\
\equiv & \{\ \text{cancellation (32)}\ \} \\
& T \cdot S = R
\end{array}
$$

The side condition of (34) requires S to be entire but not necessarily simple. In fact, it suffices that $\mathit{img}\ S \subseteq \mathit{ker}\ (\Gamma_{\mathsf{F}} T)$ since, in general, the simplicity of $f \cdot R$ equivales $\mathit{img}\ R \subseteq \mathit{ker}\ f$:

$$
\begin{array}{ll}
& \mathit{img}\ R \subseteq \mathit{ker}\ f \\
\equiv & \{\ \text{definitions}\ \} \\
& (R \cdot R^{\circ}) \subseteq f^{\circ} \cdot f \\
\equiv & \{\ id \text{ is the unit of composition}\ \} \\
& (R \cdot R^{\circ}) \subseteq f^{\circ} \cdot id \cdot f \\
\equiv & \{\ \text{shunting rules (13)}\ \} \\
& f \cdot (R \cdot R^{\circ}) \cdot f^{\circ} \subseteq id \\
\equiv & \{\ \text{composition is associative ; converse of composition}\ \} \\
& (f \cdot R) \cdot (f \cdot R)^{\circ} \subseteq id \\
\equiv & \{\ \text{definition of } \mathit{img}\ \} \\
& \mathit{img}\ (f \cdot R) \subseteq id \\
\equiv & \{\ \text{simplicity}\ \} \\
& (f \cdot R) \text{ is simple}
\end{array}
$$

In summary, the simplicity of (entire) S is a sufficient (but not necessary) condition for the fusion law (34) to hold. In particular, S can be a function, and it is under this condition that the law is presented in [6] [2].

Substitution $f := \Gamma_{\mathsf{F}} S$ and cancellation (32) lead to the *injectivity law*,

$$\Gamma_{\mathsf{F}} S = \Gamma_{\mathsf{F}} R \equiv S = R \tag{35}$$

Finally, the generic version of the *absorption property*,

$$\mathsf{F}\ R \cdot \Gamma_{\mathsf{F}} S = \Gamma_{\mathsf{F}} (R \cdot S) \Leftarrow R \cdot \in_{\mathsf{F}} \subseteq \in_{\mathsf{F}} \cdot \mathsf{F}\ R \tag{36}$$

[2] Cf. exercise 5.9 in the book. See also exercise 4.48 for a result which can be of help in reasoning about the side condition of (34).

is justified as follows:

$$
\begin{array}{ll}
& \mathsf{F}\,R \cdot \Gamma_{\mathsf{F}} S = \Gamma_{\mathsf{F}}(R \cdot S) \\
\equiv & \quad \{ \text{ universal property (30) } \} \\
& \in_{\mathsf{F}} \cdot \mathsf{F}\,R \cdot \Gamma_{\mathsf{F}} S = R \cdot S \\
\equiv & \quad \{ \text{ assume } \in_{\mathsf{F}} \cdot \mathsf{F}\,R = R \cdot \in_{\mathsf{F}}) \ \} \\
& R \cdot \in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} S = R \cdot S \\
\equiv & \quad \{ \text{ cancellation (32) } \} \\
& R \cdot S = R \cdot S
\end{array}
$$

The side condition of (36) arises from the property assumed in the second step of the proof. Together with (22), it establishes the required equality by anti-symmetry, which is equivalent to writing $\mathsf{F}\,R = \Gamma_{\mathsf{F}}(R \cdot \in)$ in such situations.

Unit and inclusion. Two concepts of set-theory can be made generic in the context above. The first one has to do with *singletons*, that is, data structures which contain a single datum. The function τ_{F} mapping every A to its singleton of type F is obtainable by transposing id, $\tau_{\mathsf{F}} = \Gamma_{\mathsf{F}} id$, and is such that (by the fusion law) $\tau_{\mathsf{F}} \cdot f = \Gamma_{\mathsf{F}} f$. Another concept relevant in the sequel is *generic inclusion*, defined by

$$\mathsf{F}\,A \xleftarrow{\in_{\mathsf{F}} \backslash \in_{\mathsf{F}}} \mathsf{F}\,A \tag{37}$$

and involving *left division* (13), the relational operator which is defined by the fact that $(R \backslash\ )$ is the upper-adjoint of $(R \cdot\ )$ for every R.

4 Instances of generic transposition

In this section we discuss the power-transpose ($\mathsf{F} = \mathcal{P}$) and the $Maybe$-transpose ($\mathsf{F} = \mathsf{Id} + 1$) as instances of the generic transpose (30). Unlike the former, the latter is not applicable to every relation. To conclude that only simple relations are $Maybe$-transposable, we first show that, for every F-transposable R, its image is at most the image of $\in_{\mathsf{F}}$:

$$\mathit{img}\ R \subseteq \mathit{img}\ \in_{\mathsf{F}} \tag{38}$$

The proof is easy to follow:

$$\mathit{img}\ R$$

$$
\begin{array}{ll}
= & \{ \text{ definition } \} \\
& R \cdot R^{\circ} \\
= & \{ \ R \text{ is } \mathsf{F}\text{-transposable ; cancellation (32) } \} \\
& (\in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} R) \cdot (\in_{\mathsf{F}} \cdot \Gamma_{\mathsf{F}} R)^{\circ} \\
= & \{ \text{ converses } \} \\
& \in_{\mathsf{F}} \cdot (\Gamma_{\mathsf{F}} R \cdot \Gamma_{\mathsf{F}} R^{\circ}) \cdot {\in_{\mathsf{F}}}^{\circ} \\
\subseteq & \{ \ \Gamma_{\mathsf{F}} R \text{ is simple ; monotonicity } \} \\
& \in_{\mathsf{F}} \cdot {\in_{\mathsf{F}}}^{\circ} \\
= & \{ \text{ definition } \} \\
& \mathit{img} \ \in_{\mathsf{F}}
\end{array}
$$

So, $\in_{\mathsf{F}}$ restricts the class of relations R which are F-*transposable*. Concerning the power-transpose, it is easy to see that $\mathit{img} \ \in_{\mathsf{F}} = \top$ since, for every a, a', there exists at least the set $\{a, a'\}$ which both a and a' belong to. Therefore, no restriction is imposed on $\mathit{img}\, R$ and transposition witnesses the well-known isomorphism $(2^A)^B \cong 2^{B \times A}$ (writing 2^A for $\mathcal{P}A$ and identifying every relation with its *graph*, a set of pairs).

By contrast, simple memberships can only be associated to the transposition of simple relations. This is what happens with $\in_{\mathsf{Id}+1} = i_1^{\circ}$ which, as the converse of an injection, is simple (5).

Conversely, all simple relations are $(\mathsf{Id}+1)$-transposable. (See a proof of this fact in appendix A.) Therefore, $(\mathsf{Id}+1)$-transposability *defines* the class of simple relations and witnesses isomorphism $(B+1)^A \cong A \rightharpoonup B$ where $A \rightharpoonup B$ denotes the set of all simple relations from A to B [3].

Another difference between the two instances of generic transposition considered so far can be found in the application of the absorption property (36). That its side condition holds for the $Maybe$-transpose is easy to show:

$$
\begin{array}{ll}
& R \cdot i_1^{\circ} \subseteq i_1^{\circ} \cdot (R + id) \\
\equiv & \{ \text{ shunting } \} \\
& i_1 \cdot R \subseteq (R + id) \cdot i_1 \\
\Leftarrow & \{ \text{ anti-symmetry } \} \\
& i_1 \cdot R = (R + id) \cdot i_1 \\
\equiv & \{ \ R + S \text{ (19) is a coproduct [6] } \}
\end{array}
$$

[3] This isomorphism is central to the data refinement calculus presented in [15].

$$i_1 \cdot R = i_1 \cdot R$$

Concerning the power-transpose, [6] define the absorption property for the *existential image* functor, $\mathsf{E}R = \Lambda(R \cdot \in)$, which coincides with the powerset relator for functions. However, E is not a relator [4]. So, the absorption property of the power-transpose can only be used where R is a function: $\mathcal{P}f \cdot \Lambda S = \Lambda(f \cdot S)$.

Finally, inclusion (37) for the power-transpose is the set-theoretic *subset ordering* [6], while its $Maybe$ instance corresponds to the expected "*flat-cpo* ordering":

$$x(\in_{\mathsf{Id}+1} \setminus \in_{\mathsf{Id}+1})y \equiv \forall a.\, x = (i_1\ a) \Rightarrow y = (i_1\ a)$$

So $Nothing$ will be included in anything and every "non-$Nothing$" x will be included only in itself [5].

5 Applications of generic transpose

The main purpose of representing relations by functions is to take advantage of the (sub)calculus of functions when applied to the transposed relations. In particular, transposition can be used to infer properties of relational combinators. Suppose that $f \oplus g$ is a functional combinator whose properties are known, for instance, $f \oplus g = [f, g]$ for which we know universal property

$$k = [f, g] \equiv \begin{cases} k \cdot i_1 = f \\ k \cdot i_2 = g \end{cases} \tag{39}$$

We may inquire about the corresponding property of another, this time *relational*, combinator $R \otimes S$ induced by transposition:

$$\Gamma_{\mathsf{F}}(R \otimes S) = (\Gamma_{\mathsf{F}} R) \oplus (\Gamma_{\mathsf{F}} S) \tag{40}$$

$$\equiv \quad \{\ (30)\ \}$$

$$R \otimes S = \in_{\mathsf{F}} \cdot ((\Gamma_{\mathsf{F}} R) \oplus (\Gamma_{\mathsf{F}} S)) \tag{41}$$

This can happen in essentially two ways, which are described next.

[4] See [12] and exercise 5.15 in [6].

[5] This is, in fact, the ordering `<=` which is derived for $Maybe$ as instance of the `Ord` class in the Haskell Prelude [8].

Proof of universality by transposition. It may happen that the universal property of functional combinator $\oplus$ is carried intact along the move from functions to relations. A good example of this is relational coproduct, whose existence is shown in [6] to stem from functional coproducts (39) by transposition [6]. One only has to instantiate (39) for $k, f, g := \Gamma_{\mathsf{F}} T, \Gamma_{\mathsf{F}} R, \Gamma_{\mathsf{F}} S$ and reason:

$$
\begin{array}{ll}
& \Gamma_{\mathsf{F}} T = [\Gamma_{\mathsf{F}} R, \Gamma_{\mathsf{F}} S] \equiv (\Gamma_{\mathsf{F}} T) \cdot i_1 = \Gamma_{\mathsf{F}} R \wedge (\Gamma_{\mathsf{F}} T) \cdot i_2 = \Gamma_{\mathsf{F}} S \\
\equiv & \quad \{ \text{ (30) and fusion (34) for } S := i_1, i_2 \} \\
& T = \ \in \cdot [\Gamma_{\mathsf{F}} R, \Gamma_{\mathsf{F}} S] \equiv \Gamma_{\mathsf{F}} (T \cdot i_1) = \Gamma_{\mathsf{F}} R \wedge \Gamma_{\mathsf{F}} (T \cdot i_2) = \Gamma_{\mathsf{F}} S \\
\equiv & \quad \{ \text{ injectivity (35) } \} \\
& T = \ \in \cdot [\Gamma_{\mathsf{F}} R, \Gamma_{\mathsf{F}} S] \equiv T \cdot i_1 = R \wedge T \cdot i_2 = S \\
\equiv & \quad \{ \text{ define } [R, S] = \in \cdot [\Gamma_{\mathsf{F}} R, \Gamma_{\mathsf{F}} S] \} \\
& T = [R, S] \equiv T \cdot i_1 = R \wedge T \cdot i_2 = S \\
\equiv & \quad \{ \text{ coproduct definition } \} \\
& [R, S] \text{ is a coproduct}
\end{array}
$$

Defined in this way, relational coproducts enjoy all properties of functional coproducts, eg. fusion, absorption etc.

This calculation, however, cannot be dualized to the generalization of the *split*-combinator $\langle f, g \rangle$ to relational $\langle R, S \rangle$. In fact, relational product is not a categorical product, which means that some properties will not hold, namely the fusion law,

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

when g, h, f are replaced by relations. According to [6], what we have is

$$\langle R, S \rangle \cdot f = \langle R \cdot f, S \cdot f \rangle \tag{43}$$

whose proof can be carried out by resorting to the explicit definition of the *split* combinator (20) and some properties of simple relations grounded on the so-called *modular law* [7].

In the following we present an alternative proof of (43) as an example of the calculation power of transposes *combined* with *Reynolds abstraction theorem* in the pointfree style [3]. The proof is more general and leads to other versions of

[6] For the same outcome *without* resorting to transposition see §2.5.2 of [12].
[7] See Exercise 5.9 in [6].

the law, depending upon which transposition is adopted, that is, which class of relations is considered.

From the diagram of $\langle f, g\rangle$

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\ & {\scriptstyle f}\nwarrow & \uparrow{\scriptstyle \langle f,g\rangle} & \nearrow{\scriptstyle g} & \\ & & C & & \end{array} \tag{44}$$

we infer the following type for *split*

$$\langle \_, \_\rangle : ((A \times B) \leftarrow C) \leftarrow ((A \leftarrow C) \times (B \leftarrow C))$$

Inspired by this diagram, we want to define the relational version of this combinator — denote it by $(\_ \otimes \_)$ for the time being — via the adaptation of (44) to transposed relations, to be denoted by $(\_ \oplus \_)$. This will be of type

$$t = (\mathsf{F}\,(A \times B) \leftarrow C) \leftarrow ((\mathsf{F}\,A \leftarrow C) \times (\mathsf{F}\,B \leftarrow C))) \tag{45}$$

Reynolds abstraction theorem. Instead of defining $(\_ \oplus \_)$ explicitly, we will reason about its properties by applying the *abstraction theorem* due to J. Reynolds [20] and advertised by P. Wadler [22] under the "*theorem for free*" heading. We follow the pointfree styled presentation of this theorem in [3], which is remarkably elegant: f be a polymorphic *function* $f : t$, whose type t can be written according to the following "grammar" of types:

$$\begin{array}{ll} t ::= t' \leftarrow t'' & \\ t ::= \mathsf{F}(t_1, \ldots, t_n) & \text{for } n\text{-ary relator } \mathsf{F} \\ t ::= v & \text{for } v \text{ a type variable (=}\textit{polymorphism}\text{ "dimension")} \end{array}$$

Let V be the set of type variables involved in type t; $\{R_v\}_{v \in V}$ be a V-indexed family of relations (f_v in case all such R_v are functions); and R_t be a relation defined inductively as follows:

$$\begin{array}{rl} R_{t:=\mathsf{F}(t_1,\ldots,t_n)} & = \mathsf{F}(R_{t_1}, \ldots, R_{t_n}) \\ R_{t:=v} & = R_v \\ R_{t:=t'\leftarrow t''} & = R_{t'} \leftarrow R_{t''} \end{array}$$

where $R_{t'} \leftarrow R_{t''}$ is defined by (9). The *free theorem of type* t reads as follows: *given any function* $f : t$ *and* V *as above,* $f\ R_t\ f$ *holds for any relational instantiation of type variables in* V. Note that this theorem is a result about t and holds for *any* polymorphic function of type t *independently* of its actual definition [8].

[8] See [3] for comprehensive evidence on the the power of this theorem when combined with Galois connections, which stems basically from the interplay between equations (7) and (8).

In the remainder of this section we deduce the *free theorem* of type t (45) and draw conclusions about the fusion and absorption properties of relational split based on such a theorem. First we calculate R_t:

$$
\begin{array}{ll}
& R_t \\
\equiv & \quad \{ \text{ induction on the structure of } t \text{ (45) } \} \\
& (\mathsf{F}\,(R_A \times R_B) \leftarrow R_C) \leftarrow ((\mathsf{F}\,R_A \leftarrow R_C) \times (\mathsf{F}\,R_B \leftarrow R_C))) \\
\equiv & \quad \{ \text{ substitution } R_A, R_B, R_C := R, S, T \text{ in order to remove subscripts } \} \\
& (\mathsf{F}\,(R \times S) \leftarrow T) \leftarrow ((\mathsf{F}\,R \leftarrow T) \times (\mathsf{F}\,S \leftarrow T)))
\end{array}
$$

Next we calculate the free theorem of $(\_ \oplus \_) : t$:

$$
\begin{array}{ll}
& (\_ \oplus \_)(R_t)(\_ \oplus \_) \\
= & \quad \{ \text{ expansion of } R_t \} \\
& (\_ \oplus \_)(\mathsf{F}\,(R \times S) \leftarrow T) \leftarrow ((\mathsf{F}\,R \leftarrow T) \times (\mathsf{F}\,S \leftarrow T)))(\_ \oplus \_) \\
= & \quad \{ \text{ meaning of Reynolds arrow combinator (9) } \} \\
& (\_ \oplus \_) \cdot ((\mathsf{F}\,R \leftarrow T) \times (\mathsf{F}\,S \leftarrow T)) \subseteq \mathsf{F}\,(R \times S) \leftarrow T) \cdot (\_ \oplus \_) \\
= & \quad \{ \textit{ shunting } (13) \} \\
& ((\mathsf{F}\,R \leftarrow T) \times (\mathsf{F}\,S \leftarrow T)) \subseteq ((\_ \oplus \_))^\circ \cdot (\mathsf{F}\,(R \times S) \leftarrow T) \cdot (\_ \oplus \_) \\
= & \quad \{ \text{ by (6) } \} \\
& (f, g)((\mathsf{F}\,R \leftarrow T) \times (\mathsf{F}\,S \leftarrow T))(h, k) \Rightarrow (f \oplus g)(\mathsf{F}\,(R \times S) \leftarrow T)(h \oplus k) \\
= & \quad \{ \text{ product relator } \} \\
& f(\mathsf{F}\,R \leftarrow T)h \wedge g(\mathsf{F}\,S \leftarrow T)k \Rightarrow (f \oplus g) \cdot T \subseteq \mathsf{F}\,(R \times S) \cdot (h \oplus k) \\
= & \quad \{ \text{ Reynolds arrow combinator (9) three times } \} \\
& f \cdot T \subseteq \mathsf{F}\,R \cdot h \wedge g \cdot T \subseteq \mathsf{F}\,S \cdot k \Rightarrow (f \oplus g) \cdot T \subseteq \mathsf{F}\,(R \times S) \cdot (h \oplus k)
\end{array}
$$

Should we replace functions f, h, g, k by transposed relations $\Gamma_{\mathsf{F}} U, \Gamma_{\mathsf{F}} V, \Gamma_{\mathsf{F}} X, \Gamma_{\mathsf{F}} Z$, respectively, we obtain

$$((\Gamma_{\mathsf{F}} U) \oplus (\Gamma_{\mathsf{F}} X)) \cdot T \subseteq \mathsf{F}\,(R \times S) \cdot ((\Gamma_{\mathsf{F}} V) \oplus (\Gamma_{\mathsf{F}} Z)) \tag{46}$$

provided conjunction

$$(\Gamma_{\mathsf{F}} U) \cdot T \subseteq \mathsf{F}\,R \cdot (\Gamma_{\mathsf{F}} V) \wedge (\Gamma_{\mathsf{F}} X) \cdot T \subseteq \mathsf{F}\,S \cdot (\Gamma_{\mathsf{F}} Z) \tag{47}$$

holds. Assuming (40), (46) can be re-written as

$$\Gamma_{\mathsf{F}}(U \otimes X) \cdot T \subseteq \mathsf{F}\,(R \times S) \cdot \Gamma_{\mathsf{F}}(V \otimes Z) \tag{48}$$

At this point we restrict T to a function t and apply the fusion law (34) without extra side conditions:

$$\Gamma_{\mathsf{F}}((U \otimes X) \cdot t) \subseteq \mathsf{F}\,(R \times S) \cdot \Gamma_{\mathsf{F}}(V \otimes Z) \tag{49}$$

For $R, S := id, id$ we will obtain —"for free" — the standard fusion law

$$(U \otimes X) \cdot t = (U \cdot t \otimes X \cdot t)$$

presented in [6] for the *split* combinator (43), ie. for $(R \otimes S) = \langle R, S \rangle$. In the reasoning, all factors involving R and S disappear and fusion takes place in both conjuncts of (47). Moreover, inclusion ($\subseteq$) becomes equality of transposed relations — thanks to (14) — and injectivity (35) is used to remove all occurrences of Γ_{F}.

In case R and S are not identities, one has different results depending on the behaviour of the chosen transposition concerning the absorption property (36).

Power transpose. In case of arbitrary relations under the power-transpose, absorption requires R and S to be functions (say r, s), whereby (49) re-writes to

$$\Gamma_{\mathsf{F}}((U \otimes X) \cdot t) \subseteq \Gamma_{\mathsf{F}}((r \times s) \cdot (V \otimes Z)) \tag{50}$$

provided $\Gamma_{\mathsf{F}}(U \cdot t) \subseteq \Gamma_{\mathsf{F}}(r \cdot V)$ and $\Gamma_{\mathsf{F}}(X \cdot t) \subseteq \Gamma_{\mathsf{F}}(s \cdot Z)$ hold. By combined use of (14) — recall that transposed relations are functions — and injectivity (35) one gets

$$(U \otimes X) \cdot t = \mathsf{F}\,(r \times s) \cdot (V \otimes Z) \tag{51}$$

provided $U \cdot t = r \cdot V$ and $X \cdot t = s \cdot Z$ hold. For $t := id$ and $(\_ \otimes \_)$ instantiated to relational split, this becomes absorption law

$$\langle r \cdot V, s \cdot Z \rangle = (r \times s) \cdot \langle V, Z \rangle \tag{52}$$

As a matter of fact, this law holds for *arbitrary* R and S, as [6] show in (admittedly) a rather tricky way. The fact that we could only arrive at a restricted version of the law is not a problem: what we have shown is that version (52) of the law in [6] is a "free" theorem which we were able to deduce in a *parametric* way.

$Maybe$ transpose. In case of *simple* relations under the $Maybe$-transpose, absorption has no side condition, and so we have

$$\Gamma_{\mathsf{F}}((U \otimes X) \cdot t) \subseteq \Gamma_{\mathsf{F}}((R \times S) \cdot (V \otimes Z)) \tag{53}$$

for (50). Under a similar reasoning, and again instantiating $t := id$ and $(\_ \otimes \_) = \langle \_, \_ \rangle$, we obtain absorption law

$$\langle R \cdot V, S \cdot Z \rangle = (R \times S) \cdot \langle V, Z \rangle \tag{54}$$

This time, our reasoning has shown that the absorption law *for simple relations* is another free theorem.

6 Other transposes

So far we have considered two instances of transposition, one applicable to *any* relation and the other restricted to *simple* relations. That *entire* relations will have their own instance of transposition is easy to guess: it will be a variant of the power-transpose imposing *non-empty* power objects (see exercise 4.45 in [6]). Dually, by (5) we will obtain a method for reasoning about *surjective* and *injective* relations.

We conclude our study of relational transposition by relating it with a data representation technique known in computer science as *hashing*. This will require further restricting the class of the transposable relations to *coreflexive* relations. On the other hand, the transpose combinator will be enriched with an extra parameter called the "hash function".

7 The Hash Transpose

Hashing. Hash tables are well known data structures [23, 13] whose purpose is to efficiently combine the advantages of both static and dynamic storage of data. Static structures such as *arrays* provide random access to data but have the disadvantage of filling too much primary storage. Dynamic, *pointer*-based structures (*eg.* search lists, search trees etc.) are more versatile with respect to storage requirements but access to data is not as immediate.

The idea of *hashing* is suggested by the informal meaning of the term itself: a large database file is "hashed" into as many "pieces" as possible, each of which is randomly accessed. Since each sub-database is smaller than the original, the time spent on accessing data is shortened by some order of magnitude. Random access is normally achieved by a so-called *hash function*, say $B \xleftarrow{h} A$, which computes, for each data item a (of type A), its *location* $h\ a$ (of type B)

in the *hash table*. Standard terminology regards as *synonyms* all data competing for the same location. A set of synonyms is called a *bucket*.

There are several ways in which data collision is handled, *eg. linear probing* [23] or *overflow handling* [13]. The former is not a totally correct representation of a data collection. The strategy of overflow handling consists in partitioning a given data collection $S \subseteq A$ into n-many, disjoint buckets, each one addressed by the relevant hash index computed by h [9].

This partition can be modelled by a function t of type $\mathcal{P}A \xleftarrow{t} B$ and the so-called "hashing effect" is the following: the membership test $a \in S$ (which requires an inspection of the whole dataset S) can be replaced by $a \in t(h\ a)$ (which only inspects the bucket addressed by location $h\ a$). That is, equivalence

$$a \in S \equiv a \in t(h\ a) \tag{55}$$

must hold for t to be regarded as a *hash table*.

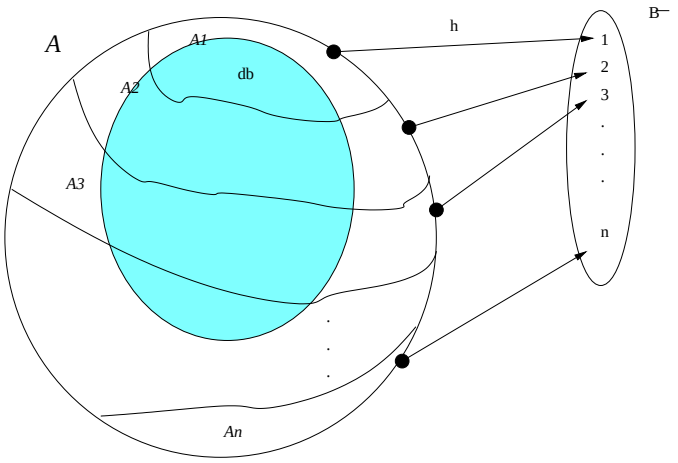

Fig. 3. The "hashing effect" (where locations are natural numbers).

Hashing as a transpose. First of all, we reason about equation (55):

$$\begin{aligned} & a \in S \equiv a \in t(h\ a) \\ = \quad & \{\ \text{introduce } b = h\ a\ \} \end{aligned}$$

[9] In fact, such buckets ("collision segments") are but the *equivalence* classes of *ker h* restricted to *db* (note that the kernel of a function is always an equivalence relation).

$$
\begin{array}{ll}
& a \in S \wedge b = h\ a \equiv a \in t\ b \\
= & \quad \{\ \text{introduce } a = a'\ \} \\
& a \in S \wedge a = a' \wedge b = h\ a' \equiv a \in t\ b \\
= & \quad \{\ \text{introduce } S \text{ as a coreflexive ; converse of hash function}\ \} \\
& aSa' \wedge a'h^\circ b \equiv a \in t\ b \\
= & \quad \{\ \text{relational composition and rule (6)}\ \} \\
& a(S \cdot h^\circ)b \equiv a(\in \cdot t)b \\
= & \quad \{\ \text{going pointfree}\ \} \\
& S \cdot h^\circ = \in \cdot t \\
= & \quad \{\ \text{power transpose}\ \} \\
& t = \Lambda(S \cdot h^\circ)
\end{array}
$$

So, for an arbitrary coreflexive relation $A \xleftarrow{S} A$, its hash-transpose (for a fixed hash function $B \xleftarrow{h} A$) is a function $\mathcal{P}A \xleftarrow{t} B$, satisfying

$$
\in \cdot t = S \cdot h^\circ \qquad
\begin{array}{ccc}
A & \xleftarrow{S} & A \\
{\scriptstyle\in}\uparrow & & \uparrow{\scriptstyle h^\circ} \\
\mathcal{P}A & \xleftarrow[t]{} & B
\end{array}
$$

By defining

$$\Theta_h\ S = \Lambda(S \cdot h^\circ) \tag{56}$$

we obtain a h-indexed family of *hash transpose* operators and associated universal properties

$$t = \Theta_h\ S \equiv \in \cdot t = S \cdot h^\circ \tag{57}$$

and thus the cancellation law

$$\in \cdot (\Theta_h\ S) = S \cdot h^\circ \tag{58}$$

etc.

In summary, the hash-transpose extends the power-transpose of coreflexive relations in the sense that $\Lambda = (\Theta_{id})$. That is, the power-transpose is the hash-transpose using id as hash function. In practice, this is an extreme case, since

some "lack of injectivity" is required of h for the hash effect to take place. Note, in passing, that the other extreme case is $h = !_A$, where $1 \xleftarrow{!} A$ denotes the unique function of its type: there is a maximum loss of injectivity and all data become synonyms!

Hashing as a Galois connection. As powerset-valued functions, hash tables are ordered by the lifting of the subset ordering $\mathcal{P}A \xleftarrow{\leq} \mathcal{P}A$ defined by $\leq = \in \setminus \in$, recall (37).

That the construction of hash tables is monotonic can be shown using the relational calculus. First we expand $\dot{\leq}$:

$$
\begin{array}{ll}
 & t \mathrel{\dot{\leq}} t' \\
\equiv & \quad \{ \text{ pointwise ordering lifted to functions (10) } \} \\
 & t \subseteq \leq \cdot t' \\
\equiv & \quad \{ \text{ definition of the subset ordering } \} \\
 & t \subseteq (\in \setminus \in) \cdot t' \\
\equiv & \quad \{ \text{ law } (R \setminus S) \cdot f = R \setminus (S \cdot f) \text{ [6], since } t' \text{ is a function } \} \\
 & t \subseteq \in \setminus (\in \cdot t') \\
\equiv & \quad \{ \text{ } (\in \cdot) \text{ is lower adjoint of } (\in \setminus) \text{ } \} \\
 & \in \cdot t \subseteq \in \cdot t' \qquad (59)
\end{array}
$$

Then we have

$$
\begin{array}{ll}
 & (\Theta_h) S \mathrel{\dot{\leq}} (\Theta_h) R \\
\equiv & \quad \{ \text{ by (59) } \} \\
 & \in \cdot (\Theta_h) S \subseteq \in \cdot (\Theta_h) R \\
\equiv & \quad \{ \text{ cancellation (58) } \} \\
 & S \cdot h^\circ \subseteq R \cdot h^\circ \\
\Leftarrow & \quad \{ \text{ } (\cdot h^\circ) \text{ is monotone, cf. lower-adjoints in (13) } \} \\
 & S \subseteq R
\end{array}
$$

So, the smallest hash-table is that associated with the empty relation $\bot$, that is $\Lambda \bot$, which is constant function $t = \underline{\emptyset}$, and the largest one is $t = \Lambda h^\circ$, the hash-transpose of id_A. In set-theoretic terms, this is A itself, the "largest" data-set of data of type A.

That the hash-transpose is not an isomorphism is intuitive: not every function t mapping B to $\mathcal{P}A$ will be a hash-table, because it may fail to place data in the correct bucket. Anyway, it is always possible to "filter" the wrongly placed synonyms from t yielding the "largest" (correct) hash table t' it contains,

$$t' = t \dot{\cap} \Lambda(h^\circ)$$

where, using *vector notation* [4], $f \dot{\cap} g$ is the lifting of $\cap$ to powerset-valued functions, $(f \dot{\cap} g)b = (f\ b) \cap (g\ b)$ for all b. In order to recover all data from such filtered t' we evaluate

$$rng\ (\in \cdot t')$$

where *rng* R (read "range of R") means $img\ R \cap id$. Altogether, we may define a function on powerset valued functions $\Xi_h t = rng\ (\in \cdot (t \dot{\cap} \Lambda(h^\circ)))$ which extracts the coreflexive relation associated with all data correctly placed in t. By reconverting $\Xi_h t$ into a hash-table again one will get a table smaller than t:

$$\Theta_h(\Xi_h t) \dot{\leq} t \tag{60}$$

(See proof in appendix B.) Another fact we can prove is "perfect" cancellation on the other side:

$$\Xi_h(\Theta_h S) = S \tag{61}$$

(See proof in appendix C.) These two cancellations, together with the monotonicity of the hash transpose Θ_h and that of Ξ_h (this is monotone because it only involves monotonic combinators) are enough, by Theorem 5.24 in [1], to establish Galois connection

$$\Theta_h R \dot{\leq} t \equiv R \subseteq rng\ (\in \cdot (t \dot{\cap} \Lambda(h^\circ)))$$

cf. diagram

$$\{S \mid S \subseteq id_A\} \underset{\Xi_h}{\overset{(\Theta_h)}{\rightleftarrows}} (\mathcal{P}A)^B$$

Being a lower adjoint, the hash-transpose will distribute over union, $\Theta_h(R \cup S) = (\Theta_h R)\ \dot{\cup}\ (\Theta_h S)$ (so hash-table construction is compositional) and enjoy other properties known of Galois connections.

Function Θ_h can be regarded as a data *representation* and Ξ_h as the corresponding *abstraction* function (Fig. 1), whereby typical "database" operations such as *insert*, *find*, and *remove* (specified on top of the powerset algebra) can be implemented by calculation [16, 19].

8 Conclusions and future work

Functional transposition is a technique for converting relations into functions aimed at developing the relational algebra indirectly *via* the algebra of functions. A functional transpose of a binary relation of a particular class is an "F-resultric" function where F is a parametric datatype with membership. This paper attempts to develop a basis for a theory of *generic transposition* under the following slogan: *generic transpose is the converse of membership pre-composition*.

Instances of *generic transpose* provide universal properties which all relations of particular *classes* of relations satisfy. Two well-known instances are considered in this paper, one applicable to any relation and the other applicable only to simple relations. In either cases, *genericity* consists of reasoning about the transposed relations without using the explicit definition of the transpose operator itself.

Our illustration of the purpose of transposition takes advantage of the *free theorem* of a polymorphic function. We show how to derive laws of relational combinators as free theorems involving their transposes. Finally, we relate the topic of functional transposition with *hashing* as a foretaste of a generic treatment of this well-known data representation technique [19].

Concerning future work, there are several directions for improving the contents of this paper. We list some of our concerns below.

Generic membership. Our use of this device, which has received some attention in recent years [6, 12], is still very superficial. Moreover, membership is being used to structure a refinement calculus of software components [14]. We would like to organize the taxonomy of relations in terms of morphisms among the membership relations of their "characteristic" transposes.

The monadic flavour. Transposed relations are "F-resultric" and so can be framed in a monadic structure if F is a monad. This is suggested in the study of the power-transpose in [6] but we haven't yet checked the genericity of the proposed constructs. This concern is related to exploiting the adjoint situations studied in [11, 10] and, in general, those involving the Kleisli category of a monad [2].

Generic hashing. Our approach to *hashing* in this paper stems from [16]. "Fractal" types [17] were later introduced as an attempt to generalize the process of hash table construction, based on characterizing datatype invariants by *sub-objects* and *pullbacks*. In the current paper we could dispense with such machinery by using *coreflexive* relations instead. The extension of this technique to other transposes based on Galois connections is currently under research [19].

Acknowledgments

The work reported in this paper has been carried out in the context of the PURE Project *(Program Understanding and Re-engineering: Calculi and Applications)* funded by FCT (the Portuguese Science and Technology Foundation) under contract `POSI/ICHS/ 44304/2002`.

References

1. Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes, December 1992.
2. J. Adámek, H. Herrlich, and G.E. Strecker. *Abstract and Concrete Categories*. John Wiley & Sons, Inc., 1990.
3. K. Backhouse and R.C. Backhouse. Safety of abstract interpretations for free,via logical relations and Galois connections. *Science of Computer Programming*, 2003. Accepted for publication.
4. R.C. Backhouse. Regular algebra applied to language problems. Available from `http://www.cs.nott.ac.uk/řcb/papers/` (Extended version of *Fusion on Languages* published in ESOP 2001. Springer LNCS 2028, pp. 107–121.).
5. R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, , and J. van der Woude. Polynomial relators. In *2nd Int. Conf. Algebraic Methodology and Software Technology (AMAST'91)*, pages 303–362. Springer LNCS, 1992.
6. R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.
7. E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, June 1970.
8. Simon Peyton Jones (ed.), John Hughes (ed.), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the programming language Haskell 98 —a non-strict, purely functional language. Technical report, February 1999.
9. J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development* . Cambridge University Press, 1st edition, 1998.
10. M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.
11. M.M. Fokkinga and L. Meertens. Adjunctions. Memoranda Informatica 94-31, University of Twente, June 1994.
12. Paul Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, University of Eindhoven, The Netherlands, 1997.
13. E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Software Engineering Series. Pitman, 1978. E. Horowitz (Ed.).
14. Sun Meng and L.S. Barbosa. On refi nement of generic state-based software components. Technical Report DI-PURe-04.1.01, Departamento de Informática, Universidade do Minho, January 2004.
15. J. N. Oliveira. *Software Reifi cation using the SETS Calculus* . In Tim Denvir, Cliff B. Jones, and Roger C. Shaw, editors, *Proc. of the BCS FACS 5th Refi nement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. ISBN 0387197524, Springer-Verlag, 8–10 January 1992. (Invited paper).

16. J. N. Oliveira. *Hash Tables —A Case Study in $\preceq$-calculation.* Technical Report DI/INESC 94-12-1, INESC Group 2361, Braga, December 1994.
17. J. N. Oliveira. *'Fractal' Types: an Attempt to Generalize Hash Table Calculation.* In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.
18. J. N. Oliveira. "Bagatelle in C arranged for VDM SoLo". *Journal of Universal Computer Science*, 7(8):754–781, 2001. Special Issue on *Formal Aspects of Software Engineering* (*Colloquium in Honor of Peter Lucas* , Institute for Software Technology, Graz University of Technology, May 18-19, 2001).
19. J. N. Oliveira. Hash tables as (generic) transposed data structures, 2004. PURe Project technical report (in preparation).
20. J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing 83*, pages 513–523, 1983.
21. J. M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989. C. A. R. Hoare.
22. P. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, Sep. 1989. ACM.
23. N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

A Proof that all simple relations are $Maybe$-transposable

We have to prove the existence of function $\Gamma_{\mathsf{Id}+1}$ which converts simple relations into $(\mathsf{Id}+1)$-resultric functions and is such that $\Gamma_{\mathsf{Id}+1} = \in^{\circ}_{\mathsf{Id}+1}$ holds.

Our guess for $\Gamma_{\mathsf{Id}+1}$ is quite "obvious": $\bot$ should be mapped to the "everywhere-$Nothing$" function $i_2 \cdot !$ (recall that $1 \xleftarrow{!} A$ is the unique function of its type), and any other simple relation R should "override" $i_2 \cdot !$ with the (non-$Nothing$) entries in $i_1 \cdot R$. Altogether, we define

$$\Gamma_{\mathsf{Id}+1} R \stackrel{\text{def}}{=} (i_2 \cdot !) \dagger (i_1 \cdot R) \tag{62}$$

where we resort to the "relation override" operator [10] defined by $R \dagger S = (R \ominus S) \cup S$, where $R \ominus S$ abbreviates $R \cdot (id - ker\, S)$. Because $R \dagger S$ preserves entirety on any argument and simplicity on both (simultaneously), $\Gamma_{\mathsf{Id}+1} R$ will be a function provided R is simple.

Next we prove the $\in^{\circ}_{\mathsf{Id}+1} = \Gamma_{\mathsf{Id}+1}$ isomorphism, that is

$$\in \cdot f = R \equiv f = \Gamma\ R$$

omitting subscripts for improved readability. The proof is adapted from [11]:

$$\begin{array}{ll} & \in \cdot f = R \\ \equiv & \quad \{\ \text{recall (29)}\ \} \\ & [id, \bot] \cdot f = R \end{array}$$

[10] This extends the *map override* operator of VDM [9].

$$
\begin{array}{ll}
\equiv & \{\ \text{injectivity (35)}\ \} \\
& \Gamma([id,\bot]\cdot f) = \Gamma\ R \\
\equiv & \{\ \text{fusion (34)}\ \} \\
& (\Gamma[id,\bot])\cdot f = \Gamma\ R \\
\equiv & \{\ \Gamma\ [R,S] = [\Gamma\ R, \Gamma\ S]\text{, cf. section 5}\ \} \\
& [\Gamma\ id, \Gamma\ \bot]\cdot f = \Gamma\ R \\
= & \{\ \text{definition (62)}\ \} \\
& [i_1, i_2\cdot !]\cdot f = \Gamma\ R \\
= & \{\ \text{uniqueness of } 1 \xleftarrow{!} 1\ = id\ \} \\
& [i_1, i_2]\cdot f = \Gamma\ R \\
= & \{\ \text{coproduct reflexion}\ \} \\
& id\cdot f = \Gamma\ R \\
= & \{\ \text{identity}\ \} \\
& f = \Gamma\ R
\end{array}
$$

B Proof of (60)

First note that

$$
\begin{array}{ll}
\Xi_h t = \mathit{rng}\ (\in\cdot(t\ \dot\cap\ \Lambda(h^\circ))) & \\
\quad = & \{\ \text{since } \in\cdot(f\ \dot\cap\ g) = (\in\cdot f)\cap(\in\cdot g)\ \} \\
\quad & \mathit{rng}\ ((\in\cdot t)\cap(\in\cdot \Lambda h^\circ)) \\
\quad = & \{\ \text{cancellation}\ \} \\
\quad & \mathit{rng}\ (\in\cdot t\cap h^\circ) \qquad (63)
\end{array}
$$

Thus (60) rewrites to

$$
\begin{array}{ll}
& (60) \\
= & \{\ \text{by (59) and (63)}\ \} \\
& \in\cdot\Theta_h(\mathit{rng}\ (\in\cdot t\cap h^\circ)) \subseteq \in\cdot t \\
\equiv & \{\ \text{definition of } \Theta_h \text{ and cancellation}\} \\
& (\mathit{rng}\ (\in\cdot t\cap h^\circ))\cdot h^\circ \subseteq \in\cdot t
\end{array}
$$

$\equiv$ { shunting }

$rng\,(\in \cdot t \cap h^\circ) \subseteq \in \cdot t \cdot h$

$\Leftarrow$ { $rng\,R \subseteq img\,R$ and $R \cap S \subseteq S$ }

$img\,(\in \cdot t \cap h^\circ) \subseteq \in \cdot t \cdot h$

$\Leftarrow$ { definition of *img* and converses }

$(\in \cdot t \cap h^\circ) \cdot (h \cap (\in \cdot t)^\circ) \subseteq \in \cdot t \cdot h$

$\Leftarrow$ { by $T \cdot (R \cap S) \subseteq (T \cdot R) \cap (T \cdot S)$ and $(R \cap S) \cdot T \subseteq (R \cdot T) \cap (S \cdot T)$ }

$\in \cdot t \cdot h \cap img\,(\in \cdot t) \cap ker\,h \cap (\in \cdot t \cdot h)^\circ \subseteq \in \cdot t \cdot h$

$\Leftarrow$ { $R \cap S \subseteq S$ }

T

C Proof of (61)

$\Xi_h(\Theta_h S) = S$

$\equiv$ { definitions and (63) }

$rng\,(\in \cdot (\Theta_h S) \cap h^\circ) = S$

$\equiv$ { definition }

$rng\,(\in \cdot (\Lambda(S \cdot h^\circ)) \cap h^\circ) = S$

$\equiv$ { cancellation }

$rng\,(S \cdot h^\circ \cap h^\circ) = S$

$\equiv$ { $S \cdot h^\circ \subseteq h^\circ$ since S is coreflexive and h is entire (see (64) below)}

$rng\,(S \cdot h^\circ) = S$

$\equiv$ { range of composition }

$rng\,(S \cdot rng\,h^\circ) = S$

$\equiv$ { h is entire }

$rng\,S = S$

$\equiv$ { S is coreflexive }

$S = S$

Auxiliar result

$$S \cdot h^\circ \subseteq h^\circ \Leftarrow \ S \text{ is coreflexive and } h \text{ is entire} \tag{64}$$

is easily proved:

$$\begin{array}{ll} & S \cdot h^\circ \subseteq h^\circ \\ \equiv & \quad \{ \text{ shunting } \} \\ & S \subseteq h^\circ \cdot h \\ \Leftarrow & \quad \{ \text{ monotonicity } \} \\ & S \subseteq id \wedge id \subseteq \mathit{ker}\, h \\ \equiv & \quad \{ \ S \text{ is coreflexive and } h \text{ is entire } \} \\ & \mathrm{T} \end{array}$$