When do datatypes commute?

Paul Hoogendijk and Roland Backhouse

Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513,
5600 MB Eindhoven,
The Netherlands.

March 6, 1997

Abstract

Polytypic programs are programs that are parameterised by type constructors (like List), unlike *polymorphic* programs which are parameterised by types (like Int). In this paper we formulate precisely the polytypic programming problem of "commuting" two datatypes. The precise formulation involves a novel notion of higher order polymorphism. We demonstrate via a number of examples the relevance and interest of the problem, and we show that all "regular datatypes" (the sort of datatypes that one can define in a functional programming language) do indeed commute according to our specification. The framework we use is the theory of allegories, a combination of category theory with the point-free relation calculus.

1 Polytypism

The ability to abstract is vital to success in computer programming. At the macro level of requirements engineering the successful designer is the one able to abstract from the particular wishes of a few clients a general purpose product that can capture a large market [31]. At the micro level of programming the ability to write so-called "generic" code capturing commonly occurring patterns is vital to reusability and thus to programmer productivity.

One of the most significant contributions to generic programming has been the notion of parametric polymorphism — first introduced by Strachey [32] and later incorporated in the language ML by Milner [25, 26]. The use of parametric polymorphism eliminates the compulsion in languages like Pascal to provide irrelevant type information. For example, it is irrelevant to the computation of the length of a list whether the elements of the list are integers, characters, or whatever. In Pascal this information must be supplied, thus enforcing the programmer to rewrite essentially the same code each time a length function is required for a new element type.

In this paper we consider a problem that entails a higher level of parametricity than can normally be expressed by polymorphism. The problem is roughly stated in the title of the paper — "when do two datatypes commute?" — and an illustrative instance of two commuting dataypes is provided by the fact that a list of trees all of the same shape can always be transformed without loss of information to a tree of lists all of the same length. The paper has three goals. First, we want to show that the problem is relevant and interesting. Second, we want to formulate the problem precisely and concisely. Third, we want to use this problem as a primer to the theory of higher order polymorphism that we have developed. It is not the purpose of this paper to provide a technical justification for all results claimed in the paper. A complete technical justification is given by Hoogendijk [14], refining earlier work of Backhouse, Doornbos and Hoogendijk [3].

Our commuting datatypes problem is an instance of what has recently been dubbed "polytypic programming" [16, 17, 23]. "Polytypic" programs distinguish themselves from polymorphic programs in that the parameter is a datatype like "list" or "tree" —a function from types to types— rather than a type like "integer", "list of integer" or "tree of string".

The emergence of polytypism as a viable research field has occurred gradually over a number of years. A landmark was the formulation by Malcolm [20, 21, 22] of a theorem expressing when two computations could be fused into one computation. Malcolm's fusion theorem was polytypic in that it was parameterised by a datatype and so could be instantiated in a variety of ways. Malcolm exploited the —polytypic— notion of a "catamorphism" and introduced the "banana bracket" notation which was popularised and extended to the —polytypic— notions of "anamorphisms" and "hylomorphisms" by Fokkinga, Meijer and Paterson [24]. (Malcolm referred to "promotion" rather than "fusion", that being the terminology used by Bird [6] at the time in his theory of lists.) Since then the theme of polytypism has been explored in a variety of ways. Several authors [4, 16, 23] have explored polytypic generalisations of existing programming problems, Doornbos [9, 8, 10] has developed a polytypic theory of program termination and the recently published book by Bird and De Moor [5] contains a wealth of material in which parameterisation by a datatype plays a central role.

Functional programmers have a well developed intuitive understanding of what it means for a function to be polymorphic. Being able to experiment with the notion by writing and executing polymorphic programs is clearly enormously beneficial to understanding. Nevertheless, an unequivocal formal semantics of "parametric polymorphism" is still an active area of research [11]. The situation with polytypism is much worse: the term is vague and probably understood in different ways by different authors. Moreover, experimental implementations of polytypism in functional programming languages are only just beginning to get off the ground. The emphasis at this point in time is in showing the ubiquity of polytypism; a drawback is the ad hoc nature of some developments. To give one simple example: the "size" function for a datatype is often cited as a polytypic generalisation of the length of a list. But what is the appropriate notion of "size" for a tree — the number of nodes or, perhaps, the depth of the tree? Without a theoretical understanding of the notion of polytypism it is difficult to provide convincing arguments for one or the other choice.

This paper contributes to the theoretical foundations of polytypism, albeit tentatively. We draw inspiration from Reynolds' [29] and Plotkin's [28] seminal accounts of the semantics of parametric polymorphism. Roughly speaking, Reynolds and Plotkin showed that any parametrically polymorphic function satisfies a certain (di)naturality property that is derivable from the type of the function via so-called "logical relations". We turn this around and define the notion of commuting datatypes by requiring that a certain higher-order naturality property be satisfied. The framework we use for formalising such properties is the theory of allegories [12], a combination of category theory with the point-free relation calculus.

In the interests of greater understanding we approach the central topic of the paper slowly and deliberately. First we need to agree on what a datatype is. For this purpose we briefly summarise Hoogendijk and De Moor's [15] arguments. The next step is to present several illustrations of "commuting datatypes". One of these is a concrete example, concerning the transposition of matrices represented as lists of list, which we learnt from D.J. Lillie. A second is more abstract: we argue that Moggi's [27] notion of "strong" functor is an instance of the phenomenon "commuting datatypes". Armed with these examples we are able to proceed to a precise formalisation of the notion.

2 Allegories and Datatypes

A brief summary of this section is that our notion of a "datatype" is a "relator with membership" [15] and an appropriate framework for developing a theory of datatypes is the theory of allegories [12].

2.1 Parametric Polymorphism

To motivate these choices let us begin by giving a brief summary of Reynolds' [29] account of parametric polymorphism. (This summary is partially borrowed from [11] with some notational changes.) Suppose we have a polymorphic function f of type $\mathsf{T}\alpha$ for all types α. That is, for each type A there is an instance $\mathsf{f_A}$ of type TA. Then parametricity of the polymorphism means that for any relation R of type $\mathsf{A} \leftarrow \mathsf{B}$ there is a relation TR of type $\mathsf{TA} \leftarrow \mathsf{TB}$ such that $(\mathsf{f_A}\,,\,\mathsf{f_B}) \in \mathsf{TR}$.

In order to make the notion of parametricity completely precise, we have to be able to extend each type constructor T in our chosen programming language to a function $\mathsf{R} \mapsto \mathsf{TR}$ from relations to relations. Reynolds did so for function spaces and product. For product he extended the (binary) type constructor $\times$ to relations by defining $\mathsf{R}\times\mathsf{S}$ for arbitrary relations R of type $A \leftarrow B$ and S of type $C \leftarrow D$ to be the relation of type $A \times C \leftarrow B \times D$ satisfying

$$((u,v)\,,\,(x,y)) \in R \times S \;\equiv\; (u,x) \in R \wedge (v,y) \in S \quad .$$

For function spaces, Reynolds extended the $\leftarrow$ operator to relations as follows. For all relations R of type $A \leftarrow B$ and S of type $C \leftarrow D$ the relation $R \leftarrow S$ is the relation of type $(A \leftarrow C) \leftarrow (B \leftarrow D)$ satisfying

$$(f\,,\,g) \in R \leftarrow S \equiv \forall (x,y{::}\ (fx\,,\,gy) \in R \Leftarrow (x,y) \in S) \quad .$$

Note that we equate a function f of type $A \leftarrow B$ with the relation f of type $A \leftarrow B$ satisfying

$$a = fb \equiv (a,b) \in f \quad .$$

This means that the relational composition $f \circ g$ of two functions is the same as their functional composition. That is, $a = f(gc) \equiv (a,c) \in f \circ g$. It also means that the definitions of $f \times g$ and $f \leftarrow g$, for functions f and g, coincide with the usual categorical definitions of the product and hom functors (respectively) for the category Set.

An example of Reynolds' parametricity property is given by function application. The type of function application is $\alpha \leftarrow (\alpha \leftarrow \beta) \times \beta$. The type constructor T is thus the function mapping types A and B to $A \leftarrow (A \leftarrow B) \times B$. The extension of T to relations maps relations R and S to the relation $R \leftarrow (R \leftarrow S) \times S$. Now suppose @ is any parametrically polymorphic function with the same type as function application. Then Reynolds' claim is that @ satisfies

$$(@_{A,C}\,,\,@_{B,D}) \in R \leftarrow (R \leftarrow S) \times S$$

for all relations R and S of types $A \leftarrow B$ and $C \leftarrow D$, respectively. Unfolding the definitions, this is the property that, for all functions f and g, and all c and d,

$$(f@c\,,\,g@d) \in R \Leftarrow \forall (x,y{::}\ (fx\,,\,gy) \in R \Leftarrow (x,y) \in S) \wedge (c,d) \in S \quad .$$

The fact that function application itself satisfies this property is in fact the basis of Reynolds' inductive proof of the parametricity property (for a particular language of typed lambda expressions). But the statement of the theorem is stronger because function application is *uniquely* defined by its parametricity property. To see this, instantiate R to the singleton set $\{(fc\,,\,fc)\}$ and S to the singleton set $\{(c,c)\}$. Then, assuming @ satisfies the parametricity property, $(f@c\,,\,f@c) \in R$. That is, $f@c = fc$. Similarly, the identity function is the unique function f satisfying the parametricity property $(f_A\,,\,f_B) \in R \leftarrow R$ for all types A and B and all relations R of type $A \leftarrow B$ —the parametricity property corresponding to the polymorphic type, $\alpha \leftarrow \alpha$ for all α, of the identity function—, and the projection function *fst* is the unique function f satisfying the parametricity property $(f_{A,B}\,,\,f_{C,D}) \in R \leftarrow R \times S$ for all types A, B, C and D and all relations R and S of types $A \leftarrow B$ and $C \leftarrow D$, respectively —the parametricity property corresponding to the polymorphic type, $\alpha \leftarrow \alpha \times \beta$ for all α and β, of the *fst* function.

The import of all this is that certain functions can be *specified* by a parametricity property. That is, certain parametricity properties have unique solutions. Most parametricity properties do not have unique solutions however. For example, both the identity function on lists and the reverse function satisfy the parametricity property of function f, for all $\mathsf{R}:\mathsf{A}\leftarrow\mathsf{B}$,

$$(\mathsf{f_A}\,,\ \mathsf{f_B}) \in \mathsf{List R} \leftarrow \mathsf{List R} \quad .$$

Here $\mathsf{List R}$ is the relation holding between two lists whenever the lists have the same length and corresponding elements of the two lists are related by R.

2.2 Allegories and Relators

2.2.1 Allegories

As we remarked earlier, a precise formalisation of Reynolds' parametricity property requires extending each type constructor T to a mapping $\mathsf{R} \mapsto \mathsf{TR}$ from relations to relations. The type requirement on this extension is that if $\mathsf{R}:\mathsf{A}\leftarrow\mathsf{B}$ then $\mathsf{TR} : \mathsf{TA} \leftarrow \mathsf{TB}$. This type requirement has of course exactly the same form as the type requirement on a functor and it has been known for a long time that datatypes are indeed functors. But just being a functor is probably much too weak a requirement to capture the notion of a datatype. Moreover, it seems to be difficult or clumsy to express non-deterministic properties in a strict categorical setting. An appropriate step to take, therefore, is to allegory theory [12] and the requirement that datatypes be "relators".

An *allegory* is a category with additional structure, the additional structure capturing the most essential characteristics of relations. Being a category means, of course, that for every object A there is an identity arrow $\mathsf{id_A}$, and that every pair of arrows $\mathsf{R} : \mathsf{A} \leftarrow \mathsf{B}$ and $\mathsf{S} : \mathsf{B} \leftarrow \mathsf{C}$, with matching source and target[1], can be composed: $\mathsf{R} \circ \mathsf{S} : \mathsf{A} \leftarrow \mathsf{C}$. Composition is associative and has id as a unit.

The additional axioms are as follows. First of all, arrows of the same type are ordered by the *partial order* $\subseteq$ and composition is monotonic with respect to this order. That is,

$$\mathsf{S_1} \circ \mathsf{T_1} \subseteq \mathsf{S_2} \circ \mathsf{T_2} \quad \Leftarrow \quad \mathsf{S_1} \subseteq \mathsf{S_2} \ \wedge \ \mathsf{T_1} \subseteq \mathsf{T_2} \quad .$$

Secondly, for every pair of arrows $\mathsf{R}\,,\mathsf{S} : \mathsf{A} \leftarrow \mathsf{B}$, their *intersection* (*meet*) $\mathsf{R} \cap \mathsf{S}$ exists and is defined by the following universal property, for each $\mathsf{X} : \mathsf{A} \leftarrow \mathsf{B}$,

$$\mathsf{X} \subseteq \mathsf{R} \wedge \mathsf{X} \subseteq \mathsf{S} \quad \equiv \quad \mathsf{X} \subseteq \mathsf{R} \cap \mathsf{S} \quad .$$

Finally, for each arrow $\mathsf{R} : \mathsf{A} \leftarrow \mathsf{B}$ its *converse* $\mathsf{R}^{\cup} : \mathsf{B} \leftarrow \mathsf{A}$ exists. The converse operator is defined by the requirements that it is its own Galois adjoint, that is,

$$\mathsf{R}^{\cup} \subseteq \mathsf{S} \quad \equiv \quad \mathsf{R} \subseteq \mathsf{S}^{\cup} \quad ,$$

[1] Note that we refer to the "source" and "target" of an arrow in a category in order to avoid confusion with the domain and range of a relation introduced later.

and is contravariant with respect to composition,

$$(\mathsf{R}\circ\mathsf{S})^{\cup} \;=\; \mathsf{S}^{\cup}\circ\mathsf{R}^{\cup} \quad .$$

All three operators of an allegory are connected by the *modular law*, also known as Dedekind's law [30]:

$$\mathsf{R}\circ\mathsf{S} \cap \mathsf{T} \subseteq (\mathsf{R} \cap \mathsf{T}\circ\mathsf{S}^{\cup})\circ\mathsf{S} \quad .$$

The standard example of an allegory is Rel, the allegory with sets as objects and relations as arrows. With this allegory in mind, we refer henceforth to the arrows of an allegory as "relations".

2.2.2 Relators

Now that we have the definition of an allegory we can give the definition of a relator.

Definition 1 (Relator) A *relator* is a monotonic functor that commutes with converse. That is, let $\mathcal{A}$ and $\mathcal{B}$ be allegories. Then the mapping $\mathsf{F} : \mathcal{A} \leftarrow \mathcal{B}$ is a relator iff,

$$\mathsf{FR} \circ \mathsf{FS} = \mathsf{F}(\mathsf{R}\circ\mathsf{S}) \quad \text{for each } \mathsf{R} : \mathsf{A} \leftarrow \mathsf{B} \text{ and } \mathsf{S} : \mathsf{B} \leftarrow \mathsf{C}, \tag{2}$$

$$\mathsf{Fid}_{\mathsf{A}} = \mathsf{id}_{\mathsf{FA}} \quad \text{for each object } \mathsf{A}, \tag{3}$$

$$\mathsf{FR} \subseteq \mathsf{FS} \;\Leftarrow\; \mathsf{R} \subseteq \mathsf{S} \quad \text{for each } \mathsf{R} : \mathsf{A} \leftarrow \mathsf{B} \text{ and } \mathsf{S} : \mathsf{A} \leftarrow \mathsf{B}, \tag{4}$$

$$(\mathsf{FR})^{\cup} = \mathsf{F}(\mathsf{R}^{\cup}) \quad \text{for each } \mathsf{R} : \mathsf{A} \leftarrow \mathsf{B}. \tag{5}$$

□

Two examples of relators have already been given. List is a unary relator, and product is a binary relator. List is an example of an inductively-defined datatype; in [1] it was observed that all inductively-defined datatypes are relators.

A design requirement which lead to the above definition of a relator [1, 2] is that a relator should extend the notion of a functor but in such a way that it coincides with the latter notion when restricted to functions. Formally, relation $\mathsf{R} : \mathsf{A} \leftarrow \mathsf{B}$ is *total* iff

$$\mathsf{id}_{\mathsf{B}} \subseteq \mathsf{R}^{\cup}\circ\mathsf{R} \quad ,$$

and relation R is single-valued or *simple* iff

$$\mathsf{R}\circ\mathsf{R}^{\cup} \subseteq \mathsf{id}_{\mathsf{A}} \quad .$$

A *function* is a relation that is both total and simple. It is easy to verify that total and simple relations are closed under composition. Hence, functions are closed under composition too. In other words, the functions form a sub-category. For an allegory $\mathcal{A}$, we denote the sub-category of functions by $Map(\mathcal{A})$. In particular, $Map(\mathsf{Rel})$ is the category having sets as objects and functions as arrows. Now the desired property of relators is that relator $\mathsf{F} : \mathcal{A} \leftarrow \mathcal{B}$ is a functor of type $Map(\mathcal{A}) \leftarrow Map(\mathcal{B})$. It is easily shown that our definition of relator guarantees this property.

2.2.3 Division and Tabulation

The allegory Rel has more structure than we have captured so far with our axioms. For instance, in Rel we can take arbitrary unions (joins) of relations. There are also two "division" operators, and Rel is "tabulated". In full, Rel is a unitary, tabulated, locally complete, division allegory. For full discussion of these concepts see [12] or [5]. Here we briefly summarise the relevant definitions.

We say that an allegory is *locally complete* if for each set $\mathcal{S}$ of relations of type $A \leftarrow B$, the union $\cup\mathcal{S} : A \leftarrow B$ exists and, furthermore, intersection and composition distribute over arbitrary unions. The defining property of union is that, for all $X : A \leftarrow B$,

$$\cup\mathcal{S} \subseteq X \equiv \forall \langle S \in \mathcal{S} :: S \subseteq X \rangle \ .$$

We use the notation $\perp\!\!\!\perp_{A,B}$ for the smallest relation of type $A \leftarrow B$ and $\top\!\!\!\top_{A,B}$ for the largest relation of the same type.

The existence of a largest relation for each pair of objects A and B is guaranteed by the existence of a "unit" object, denoted by 1. We say that object 1 is a *unit* if id_1 is the largest relation of its type and for every object A there exists a total relation $!_A : 1 \leftarrow A$. If an allegory has a unit then it is said to be *unitary*.

The most crucial consequence of the distributivity of composition over union is the existence of two so-called *division* operators "\" and "/". Specifically, we have the following three Galois-connections. For all $R : A \leftarrow B$, $S : B \leftarrow C$ and $T : A \leftarrow C$,

$$R \circ S \subseteq T \quad \equiv \quad S \subseteq R \backslash T \ ,$$

$$R \circ S \subseteq T \quad \equiv \quad R \subseteq T/S \ ,$$

$$S \subseteq R \backslash T \quad \equiv \quad R \subseteq T/S \ ,$$

(where, of course, the third is just a combination of the first two).

Note that $R\backslash T : B \leftarrow C$ and $T/S : A \leftarrow B$. The interpretation of the factors is

$$(b,c) \in R\backslash T \quad \equiv \quad \forall \langle a : \ (a,b) \in R : \ (a,c) \in T \rangle \ ,$$

$$(a,b) \in T/S \quad \equiv \quad \forall \langle c : \ (b,c) \in S : \ (a,c) \in T \rangle \ .$$

The final characteristic of Rel is that it is "tabular". That is, each relation is a set of ordered pairs. Formally, we say that an object C and a pair of functions $f : A \leftarrow C$ and $g : B \leftarrow C$ is a *tabulation* of relation $R : A \leftarrow B$ if

$$R = f \circ g^{\cup} \ \wedge \ f^{\cup} \circ f \cap g^{\cup} \circ g = \mathsf{id}_C \ .$$

An allegory is said to be *tabular* if every relation has a tabulation.

Allegory Rel is tabular. Given relation $R : A \leftarrow B$, define C to be the subset of the cartesian product $A \times B$ containing the pairs of elements for which $(x,y) \in R$. Then the pair of projection functions $\mathsf{outl} : A \leftarrow C$ and $\mathsf{outr} : B \leftarrow C$ is a tabulation of R.

If allegory $\mathcal{B}$ is tabular, a functor is monotonic iff it commutes with converse [5]. So, if we define a relator on a tabular allegory, one has to prove either requirement (4) or (5). For this reason Bird and De Moor [5] define a relator to be a monotonic functor; they also attribute the definition to Kawahara [18] and Carboni, Kelly and Wood [7].

2.2.4 Domains

In addition to the source and target of a relation it is useful to know their domain and range. The *domain* of a relation $R : A \leftarrow B$ is that subset $R{>}$ of id_B defined by the Galois connection:

$$R \subseteq \top\!\!\top_{A,B} \circ X \;\equiv\; R{>} \subseteq X \quad \text{for each } X \subseteq \mathsf{id}_B. \tag{6}$$

The *range* of $R : A \leftarrow B$, which we denote by $R{<}$, is the domain of $R^{\cup}$.

The interpretation of the domain of a relation is the set of all y such that $(x,y) \in R$ for some x. We use the names "domain" and "range" because we usually interpret relations as transforming "input" y on the right to "output" x on the left. The domain and range operators play an important role in a relational theory of datatypes.

2.2.5 Pointwise Closed Classes of Relators

We have already mentioned a few examples of relators. Of these, only product is primitive; the others are composite. In general, our concern is with establishing that certain *classes* of relators are commuting. That is, every pair of relators in the class commutes with each other. A requirement is that a class be sufficiently rich in the sense that it is closed under a number of composition operators. The composition operators that we consider indispensable are functional composition and tupling.

Little needs to be said about functional composition at this moment. It is easy to verify that the functional composition of two relators $F : \mathcal{A} \leftarrow \mathcal{B}$ and $G : \mathcal{B} \leftarrow \mathcal{C}$, which we denote by FG, is a relator. There is also an identity relator for each allegory $\mathcal{A}$, which we denote by $\mathtt{Id}$ leaving the specific allegory to be inferred from the context. The relators thus form a category —a fact that we need to bear in mind later— .

Tupling permits the definition of relators that are multiple-valued. So far, all our examples of relators have been single-valued. Modern functional programming languages provide a syntax whereby relators (or, more precisely, the corresponding functors) can de defined as datatypes. Often datatypes are single-valued, but in general they are not. Mutually-recursive datatypes are commonly occurring programmer-defined datatypes that are not single-valued. But composite-valued relators also occur in the definition of single-valued relators. For example, the (single-valued) relator F defined by $FR = R \times R$ is the composition of the relator $\times$ after the (double-valued) doubling relator. More complicated examples like the binary relator $\otimes$ that maps the pair (R, S) to $R+S\times S$ involve projection as well as repetition (doubling), product and coproduct. The programmer is not usually aware of this because the use of multiple-valued relators is camouflaged by the use of variables. For our purposes, however, we need a variable-free mechanism for composing relators. This is achieved by making the arity of a relator explicit and introducing mechanisms for tupling and projection.

We consider a collection of allegories created by closing some base allegory $\mathcal{C}$ under the formation of finite cartesian products. (The cartesian product of two allegories, defined in the usual pointwise fashion, is clearly an allegory. Moreover, properties such as unitary, locally complete etc. are preserved in the process.) An allegory in the collection is thus $\mathcal{C}^k$ where k, the *arity* of the allegory is either a natural number or $l*m$ where l is an arity and m is a number. Note that we identify $1*k$ and $k*1$ with k.

The *arity* of a relator F is $k\leftarrow l$ if the target of F is $\mathcal{C}^k$ and its source is $\mathcal{C}^l$. We write $F : k\leftarrow l$ rather than the strictly correct $F : \mathcal{C}^k\leftarrow\mathcal{C}^l$. A relator with arity $1\leftarrow 1$ is called an *endorelator* and a relator with arity $1\leftarrow k$ for some k is called *single-valued*.

Given a number k and a number of relators F_i ($0\leq i<k$) all of the same arity $l\leftarrow m$, the relators can be tupled in the obvious way to form a relator of arity $l*k\leftarrow m$. We denote the tupled relator by $\Delta(i\colon 0\leq i<k\colon F_i)$. (Note that this defines Δ as a mapping from the range $(i\colon\colon 0\leq i<k)$ to the relators.) Some variations on this notation are used. First, we often use F_k to abbreviate the mapping $(i\colon 0\leq i<k\colon F_i)$ in a tuple expression. That is, we abbreviate $\Delta(i\colon 0\leq i<k\colon F_i)$ to ΔF_k. Second, we sometimes use Δ as an infix operator —reduced slightly in size to avoid ambiguity— ; thus, $F{\vartriangle}G$ is the relator that maps relation R to the pair of relations $(FR\,,\,GR)$. Thirdly, when all the relators are equal to one and the same relator F we write simply ΔF ; this is the relator that given relation R makes k copies of FR to create a vector of length k. Finally, there are times when we need to make the implicit parameter k explicit. In such cases we add it as a subscript to Δ. In particular, we most often write $\Delta_k F$ in order to indicate clearly the amount of duplication of F.

Complementary to tupling is projection. For each number k and for each i, $0\leq i<k$, we can define the relator Proj_i that maps a k-tuple of relations $R_0, \ldots, R_{k-1}$ to R_i . (Note that, following the convention introduced above, Proj_k denotes the function mapping i in the range $0\leq i<k$ to Proj_i.) In the case that k is 2 we use the special notation Outl and Outr for the two projections. Note that the identity relator is a special case of a projection (the case $k=1$).

Using tupling and projection we can define several other operations. The operation $\_^k$ can, of course, be extended to a functor. If F has arity $l\leftarrow m$ then

$$F^k \triangleq \Delta(i\colon 0\leq i<k\colon F\mathsf{Proj}_i)$$

has arity $l*k\leftarrow m*k$. Another relator transposes $l*k$ into $k*l$. We denote this relator by τ —irrespective of the dimensions l and k, relying on the context to determine what its dimensions are— . The definition of $\tau : k*l\leftarrow l*k$ is $\Delta_k\Delta_l(\mathsf{Proj}_l\mathsf{Proj}_k)$; it is the unique mapping such that for all matrices of single-valued relators $\mathsf{F}_{i,j}$, where $0 \le i < k$ and $0 \le j < l$, one has

$$\tau(\Delta_k \Delta_l(\mathsf{F}_{k,l})) = \Delta_l \Delta_k(\mathsf{F}_{k,l}) \quad .$$

By composing $\_^k$ and τ we get a functor dual to $\_^k$; specifically, we define ${}^k\mathsf{F}$ by ${}^k\mathsf{F} = \tau\mathsf{F}^k\tau$. Thus, for $\mathsf{F} : l \leftarrow m$ we have $\mathsf{F}^k : l{*}k \leftarrow m{*}k$ and ${}^k\mathsf{F} : k{*}l \leftarrow k{*}m$.

Projection and tupling are connected by the law

$$\mathsf{H} = \Delta\mathsf{F}_k \equiv \forall(i{:}\ 0 \le i < k{:}\ \mathsf{Proj}_i\mathsf{H} = \mathsf{F}_i) \quad , \tag{7}$$

for all H and F. We also need to bear this in mind when defining the notion of a commuting class of relators.

We conclude this subsection with the definition of a "pointwise closed" class of relators.

Definition 8 (Pointwise Closed) A collection of relators is said to be *pointwise closed* with *base allegory* $\mathcal{C}$ if each relator in the collection has type $\mathcal{C}^k \leftarrow \mathcal{C}^l$ for some arities k and l, and the collection includes all projections and is closed under functional composition and tupling.
□

We have chosen the name "pointwise closed" to suggest the idea that the classes of relators we are interested in are those that are obtained by pointwise definitions starting from some primitive collection of relators[2]. For example, the binary relator that maps the pair (R, S) to $\mathsf{R}{+}\mathsf{S}{\times}\mathsf{S}$ would be expressed as $+(\mathtt{Outl} \vartriangle (\times(\Delta_2\mathtt{Outr})))$ in the notation introduced above. The primitive relators in this example are coproduct and product which we now introduce.

2.2.6 Regular Relators

The "regular relators" are those relators constructed from three primitive (classes of) relators by pointwise closure and induction.

For each object A in an allegory there is a relator K_A defined by $\mathsf{K}_\mathsf{A}\mathsf{R} = \mathsf{id}_\mathsf{A}$. Such relators are called *constant* relators.

A *coproduct* of two objects consists of an object and two injection relations. The object is denoted by $\mathsf{A}{+}\mathsf{B}$ and the two relations by $\mathsf{inl}_{\mathsf{A},\mathsf{B}} : \mathsf{A}{+}\mathsf{B} \leftarrow \mathsf{A}$ and $\mathsf{inr}_{\mathsf{A},\mathsf{B}} : \mathsf{A}{+}\mathsf{B} \leftarrow \mathsf{B}$. For the injection relations we require that

$$\mathsf{inl}^{\cup}_{\mathsf{A},\mathsf{B}} \circ \mathsf{inl}_{\mathsf{A},\mathsf{B}} = \mathsf{id}_\mathsf{A} \quad \text{and} \quad \mathsf{inr}^{\cup}_{\mathsf{A},\mathsf{B}} \circ \mathsf{inr}_{\mathsf{A},\mathsf{B}} = \mathsf{id}_\mathsf{B} \quad , \tag{9}$$

$$\mathsf{inl}^{\cup}_{\mathsf{A},\mathsf{B}} \circ \mathsf{inr}_{\mathsf{A},\mathsf{B}} = \perp\!\!\perp_{\mathsf{A},\mathsf{B}} \quad , \tag{10}$$

and

$$\mathsf{inl}_{\mathsf{A},\mathsf{B}} \circ \mathsf{inl}^{\cup}_{\mathsf{A},\mathsf{B}} \cup \mathsf{inr}_{\mathsf{A},\mathsf{B}} \circ \mathsf{inr}^{\cup}_{\mathsf{A},\mathsf{B}} \ = \ \mathsf{id}_{\mathsf{A}+\mathsf{B}} \quad . \tag{11}$$

[2]If there is a standard term in the literature that we could use instead of "pointwise closed" then we would be happy to do so. We do not know of such a term.

Having the functions inl and inr, we can define the *junc* operator: for all $\mathsf{R} : \mathsf{C} \leftarrow \mathsf{A}$ and $\mathsf{S} : \mathsf{C} \leftarrow \mathsf{B}$,

$$\mathsf{R} \triangledown \mathsf{S} \;\triangleq\; \mathsf{R} \circ \mathsf{inl}^{\cup}_{\mathsf{A,B}} \,\cup\, \mathsf{S} \circ \mathsf{inr}^{\cup}_{\mathsf{A,B}} \quad , \tag{12}$$

and the *coproduct relator*: for all $\mathsf{R} : \mathsf{C} \leftarrow \mathsf{A}$ and $\mathsf{S} : \mathsf{D} \leftarrow \mathsf{B}$

$$\mathsf{R}{+}\mathsf{S} \;\triangleq\; (\mathsf{inl}_{\mathsf{C,D}} \circ \mathsf{R}) \triangledown (\mathsf{inr}_{\mathsf{C,D}} \circ \mathsf{S}) \quad .$$

A *product* of two objects consists of an object and two projection arrows. The object is denoted by $\mathsf{A}{\times}\mathsf{B}$ and the two arrows by $\mathsf{outl}_{\mathsf{A,B}} : \mathsf{A} \leftarrow \mathsf{A}{\times}\mathsf{B}$ and $\mathsf{outr}_{\mathsf{A,B}} : \mathsf{B} \leftarrow \mathsf{A}{\times}\mathsf{B}$. For the arrows we require them to be functions and that

$$\mathsf{outl}_{\mathsf{A,B}} \circ \mathsf{outr}^{\cup}_{\mathsf{A,B}} = \top\!\!\top_{\mathsf{A,B}} \quad , \tag{13}$$

and

$$\mathsf{outl}^{\cup}_{\mathsf{A,B}} \circ \mathsf{outl}_{\mathsf{A,B}} \,\cap\, \mathsf{outr}^{\cup}_{\mathsf{A,B}} \circ \mathsf{outr}_{\mathsf{A,B}} \;=\; \mathsf{id}_{\mathsf{A}\times\mathsf{B}} \quad . \tag{14}$$

Having the projection functions outl and outr, we can define the *split* operator on relations: for all $\mathsf{R} : \mathsf{A} \leftarrow \mathsf{C}$ and $\mathsf{S} : \mathsf{B} \leftarrow \mathsf{C}$

$$\mathsf{R} \vartriangle \mathsf{S} \;\triangleq\; \mathsf{outl}^{\cup}_{\mathsf{A,B}} \circ \mathsf{R} \,\cap\, \mathsf{outr}^{\cup}_{\mathsf{A,B}} \circ \mathsf{S} \quad , \tag{15}$$

and the *product relator*: for all for $\mathsf{R} : \mathsf{C} \leftarrow \mathsf{A}$ and $\mathsf{S} : \mathsf{D} \leftarrow \mathsf{B}$,

$$\mathsf{R}{\times}\mathsf{S} \;\triangleq\; (\mathsf{R} \circ \mathsf{outl}_{\mathsf{A,B}}) \vartriangle (\mathsf{S} \circ \mathsf{outr}_{\mathsf{A,B}}) \quad .$$

(The similarity between the symbol "Δ" used to denote tupling of relators and the split operator "$\vartriangle$" is, of course, not coincidental.)

Tree relators are defined as follows. Suppose that relation $\mathsf{in} : \mathsf{A} \leftarrow \mathsf{FA}$ is an initial F-algebra. That is to say, suppose that for each relation $\mathsf{R} : \mathsf{B} \leftarrow \mathsf{FB}$ (thus each "F-algebra") there exists a unique F−homomorphism to R from in. We denote this unique homomorphism by $(\![\mathsf{F}\,;\mathsf{R}]\!)$. Formally, $(\![\mathsf{F}\,;\mathsf{R}]\!)$ and in are characterized by the universal property that, for each relation $\mathsf{X} : \mathsf{B} \leftarrow \mathsf{A}$ and each relation $\mathsf{R} : \mathsf{B} \leftarrow \mathsf{FB}$,

$$\mathsf{X} = (\![\mathsf{F}\,;\mathsf{R}]\!) \;\;\equiv\;\; \mathsf{X} {\circ} \mathsf{in} = \mathsf{R} \circ \mathsf{FX} \quad . \tag{16}$$

Now, let $\otimes$ be a binary relator and assume that, for each A, $\mathsf{in}_{\mathsf{A}} : \mathsf{TA} \leftarrow \mathsf{A} \otimes \mathsf{TA}$ is an initial algebra of $(\mathsf{A}\otimes)$[3]. Then the mapping T defined by, for all $\mathsf{R} : \mathsf{A} \leftarrow \mathsf{B}$,

$$\mathsf{TR} = (\![\mathsf{A}\otimes\,;\, \mathsf{in}_{\mathsf{B}} \circ \mathsf{R}{\otimes}\mathsf{id}_{\mathsf{TB}}]\!)$$

is a relator, *the tree relator induced by* $\otimes$.

(Characterization (16) can be weakened without loss of generality so that the universal quantifications over *relations* X and R are restricted to universal quantifications over *functions* X and R. This, in essence, is what Bird and De Moor [5] refer to as the Eilenberg-Wright lemma.)

[3] Here and elsewhere we use the section notation $(\mathsf{A}\otimes)$ for the relator $\otimes(\mathsf{K}_{\mathsf{A}} \vartriangle \mathrm{Id})$.

2.3 Natural Transformations

Reynolds' characterisation of parametric polymorphism predicts that certain polymorphic functions are natural transformations. To see this it helps to re-express the pointwise definition of the $\leftarrow$ operator in the following point-free form:

$$(\mathsf{f},\mathsf{g}) \in \mathsf{R} \leftarrow \mathsf{S} \equiv \mathsf{f} \circ \mathsf{S} \subseteq \mathsf{R} \circ \mathsf{g} \quad .$$

Now consider, for example, the reverse function on lists, denoted here by rev. This has polymorphic type $\mathsf{List}A \leftarrow \mathsf{List}A$ for all A and so, according to Reynolds' prediction:

$$(\mathsf{rev},\mathsf{rev}) \in \mathsf{List}\mathsf{R} \leftarrow \mathsf{List}\mathsf{R}$$

for all relations R. That is,

$$\mathsf{rev} \circ \mathsf{List}\mathsf{R} \subseteq \mathsf{List}\mathsf{R} \circ \mathsf{rev}$$

for all relations R. Similarly the function that makes a pair out of a single value, here denoted by fork, has type $A \times A \leftarrow A$ for all A, and so is predicted to satisfy the property:

$$\mathsf{fork} \circ \mathsf{R} \subseteq \mathsf{R} \times \mathsf{R} \circ \mathsf{fork}$$

for all relations R.

The above properties of rev and fork are not natural transformation properties because they assert an inclusion and not an equality; they are sometimes called "lax" natural transformation properties. It so happens that the inclusion in the case of rev can be strengthened to an equality but this is certainly not the case for fork. Nevertheless, in the functional programmer's world being a lax natural transformation between two relators is equivalent to being a natural transformation between two functors as we shall now explain.

Since relators are by definition functors, the standard definition of a natural transformation between relators makes sense. That is to say, we define a collection of relations α indexed by objects (equivalently, a mapping α of objects to relations) to be *a natural transformation of type* $\mathsf{F} \leftarrow \mathsf{G}$, for relators F and G iff

$$\mathsf{F}\mathsf{R} \circ \alpha_{\mathsf{B}} = \alpha_{\mathsf{A}} \circ \mathsf{G}\mathsf{R} \quad \text{for each } \mathsf{R} : \mathsf{A} \leftarrow \mathsf{B}.$$

However, as illustrated by fork above, many collections of relations are not natural with equality but with an inclusion. That is why we define two other types of natural transformation denoted by $\mathsf{F} \hookleftarrow \mathsf{G}$ and $\mathsf{F} \hookrightarrow \mathsf{G}$, respectively. We define:

$$\alpha : \mathsf{F} \hookleftarrow \mathsf{G} \triangleq (\mathsf{F}\mathsf{R} \circ \alpha_{\mathsf{B}} \supseteq \alpha_{\mathsf{A}} \circ \mathsf{G}\mathsf{R} \quad \text{for each } \mathsf{R} : \mathsf{A} \leftarrow \mathsf{B})$$

and

$$\alpha : \mathsf{F} \hookrightarrow \mathsf{G} \triangleq (\mathsf{F}\mathsf{R} \circ \alpha_{\mathsf{B}} \subseteq \alpha_{\mathsf{A}} \circ \mathsf{G}\mathsf{R} \quad \text{for each } \mathsf{R} : \mathsf{A} \leftarrow \mathsf{B}) \quad .$$

A relationship between naturality in the allegorical sense and in the categorical sense is given by two lemmas. Recall that relators respect functions, i.e. relators are functors on the sub-category *Map*. The first lemma states that an allegorical natural transformation is a categorical natural transformation:

$$(\mathsf{Ff} \circ \alpha_{\mathsf{B}} = \alpha_{\mathsf{A}} \circ \mathsf{Gf} \quad \text{for each function } \mathsf{f} : \mathsf{A} \leftarrow \mathsf{B}) \;\Leftarrow\; \alpha : \mathsf{F} \hookleftarrow \mathsf{G} \quad .$$

The second lemma states the converse; the lemma is valid under the assumption that the source allegory of the relators F and G is tabular:

$$\alpha : \mathsf{F} \hookleftarrow \mathsf{G} \;\Leftarrow\; (\mathsf{Ff} \circ \alpha_{\mathsf{B}} = \alpha_{\mathsf{A}} \circ \mathsf{Gf} \quad \text{for each function } \mathsf{f} : \mathsf{A} \leftarrow \mathsf{B}) \quad .$$

In the case that all elements of the collection α are *functions* we thus have:

$$\alpha : \mathsf{F} \hookleftarrow \mathsf{G} \text{ in } \mathcal{A} \;\equiv\; \alpha : \mathsf{F} \leftarrow \mathsf{G} \text{ in } \mathit{Map}(\mathcal{A})$$

where by "in X" we mean that all quantifications in the definition of the type of natural transformation range over the objects and arrows of X.

Since natural transformations of type $\mathsf{F} \hookleftarrow \mathsf{G}$ are the more common ones and, as argued above, agree with the categorical notion of natural transformation in the case that they are functions, we say that α is a *natural transformation* if $\alpha : \mathsf{F} \hookleftarrow \mathsf{G}$ and we say that α is a *proper* natural transformation if $\alpha : \mathsf{F} \leftarrow \mathsf{G}$. (As mentioned earlier, other authors use the term "lax natural transformation" instead of our natural transformation.)

The natural transformations studied in the computing science literature are predominantly (collections of) functions. In contrast, the natural transformations discussed in this paper are almost all non-functional either because they are partial or because they are non-deterministic (or both).

The notion of arity is of course applicable to all functions defined on product allegories; in particular natural transformations have an arity. A natural transformation of arity $\mathsf{k} \leftarrow \mathsf{l}$ maps an l-tuple of objects to a k-tuple of relations. The governing rule is: if α is a natural transformation to F from G (of whatever type — proper or not) then the arities of F and G and α must be identical. Moreover, the composition $\alpha \circ \beta$ of two natural transformations (defined by $(\alpha \circ \beta)_{\mathsf{A}} = \alpha_{\mathsf{A}} \circ \beta_{\mathsf{A}}$) is only valid if α and β have the same arity (since the composition is componentwise composition in the product allegory).

2.4 Membership and Fans

Since our goal is to use naturality properties to specify relations it is useful to be able to interpret what it means to be "natural". All interpretations of naturality that we know of assume either implicitly or explicitly that a datatype is a way of structuring information and, thus, that one can always talk about the information stored in an instance of the datatype. A natural transformation is then interpreted as a transformation of one type of structure to another type of structure that rearranges the stored information in some way but does no actual computations on the stored information. Doing no computations on the stored information guarantees that the transformation is independent of the stored information and thus also of the representation used when storing the information.

Hoogendijk and De Moor have made this precise [15]. Their argument, briefly summarised here, is based on the thesis that a datatype is a relator with a membership relation.

Suppose F is a relator. For the moment we assume that F is an endorelator. (Thus the source of F is not a product of allegories.) The interpretation of FR is a relation between F-structures of the same shape such that corresponding values stored in the two structures are related by R. For example, ListR is a relation between two lists of the same length —the shape of a list is its length— such that the ith element of the one list is related by R to the ith element of the other. Suppose A is an object and suppose $X \subseteq \mathsf{id}_A$. So X is a partial identity relation; in effect X selects a subset of A, those values standing in the relation X to themselves. By the same token, FX is the partial identity relation that selects all F-structures in which all the stored values are members of the subset selected by X. This informal reasoning is the basis of the definition of a membership relation for the datatype F.

The precise specification of membership for F is a collection of relations mem (indexed by objects of the source allegory of F) such that $\mathsf{mem}_A : A \leftarrow \mathsf{F}A$ and such that FX is the largest subset Y of $\mathsf{id}_{\mathsf{F}A}$ whose “members” are elements of the set X. Formally, mem is required to satisfy the property:

$$\forall(X,Y\colon X \subseteq \mathsf{id}_A \wedge Y \subseteq \mathsf{id}_{\mathsf{F}A}\colon\ \mathsf{F}X \supseteq Y \ \equiv\ (\mathsf{mem}_A \circ Y)_< \subseteq X) \tag{17}$$

Note that (17) is a Galois connection. A consequence is that a necessary condition for relator F to have membership is that it preserve arbitrary intersections of partial identities. In [15] an example due to P.J. Freyd is presented of a relator that does not have this property. Thus, if one agrees that having membership is an essential attribute of a datatype, the conclusion is that not all relators are datatypes.

Property (17) doesn't make sense in the case that F is not an endorelator but the problem is easily rectified. The general case that we have to consider is a relator of arity $k \leftarrow l$ for some numbers k and l. We consider first the case that k is 1; for $k > 1$ the essential idea is to split the relator into l component relators each of arity $1 \leftarrow k$. For illustrative purposes we assume for the moment that $l = 2$.

The interpretation of a binary relator $\otimes$ as a datatype-former is that a structure of type $A_0 \otimes A_1$, for objects A_0 and A_1, contains data at two places: the left and right argument. In other words, the membership relation for $\otimes$ has two components, $\mathsf{mem}_0 : A_0 \leftarrow A_0 \otimes A_1$ and $\mathsf{mem}_1 : A_1 \leftarrow A_0 \otimes A_1$, one for each argument. Just as in the endo case, for all $\otimes$-structures being elements of the set $X_0 \otimes X_1$, for partial identities X_0 and X_1, the component for the left argument should return all and only elements of X_0, the component for the right argument all and only elements of X_1. Formally, we demand that, for all partial identities $X_0 \subseteq \mathsf{id}_{A_0}$, $X_1 \subseteq \mathsf{id}_{A_1}$ and $Y \subseteq \mathsf{id}_{A_0 \otimes A_1}$,

$$X_0 \otimes X_1 \supseteq Y \quad \equiv \quad (\mathsf{mem}_0 \circ Y)_< \subseteq X_0 \ \wedge\ (\mathsf{mem}_1 \circ Y)_< \subseteq X_1 \tag{18}$$

The rhs of (18) can be rewritten as

$$((\mathsf{mem}_0\,,\mathsf{mem}_1) \circ \Delta_2 Y)_< \ \subseteq\ (X_0\,,X_1)$$

where Δ_2 denotes the doubling functor: $\Delta_2 Y = (Y, Y)$. Now, writing $\mathsf{mem} = (\mathsf{mem}_0, \mathsf{mem}_1)$, $A = (A_0, A_1)$ and $X = (X_0, X_1)$, equation (18) becomes, for all partial identities $X \subseteq \mathsf{id}_A$ and $Y \subseteq \mathsf{id}_{(\otimes)A}$,

$$(\otimes)X \supseteq Y \;\equiv\; (\mathsf{mem} \circ \Delta_2 Y)_< \subseteq X \ .$$

The above equation for a membership relation for a binary relator suggests the equation for an arbitrary single-valued relator F of arity $1 \leftarrow l$. Specifically, we demand that the membership relation mem for F be a collection of relations such that, for all vectors of objects A (i.e. objects of arity l)

$$\mathsf{mem}_A \ :\ A \leftarrow \Delta_l F A$$

and such that, for all partial identities $X \subseteq \mathsf{id}_A$ and $Y \subseteq \mathsf{id}_{FA}$,

$$FX \supseteq Y \;\equiv\; (\mathsf{mem}_A \circ \Delta_l Y)_< \subseteq X \ . \tag{19}$$

In fact, in [15] neither (19) nor (17) is used as the defining property of membership. Instead the following definition is used, and it is shown that (19) is a consequence thereof. (Actually [15] only considers the case $l = 1$, a detail we will ignore here.)

A collection of arrows mem of arity $k * l \leftarrow l$ is a *membership* relation of relator $F : k \leftarrow l$, if for each vector of objects A

$$\mathsf{mem}_A \ :\ ((\Delta_k)^l)A \leftarrow \Delta_l F A$$

and for each object B, vector of objects A and each $R \ :\ A \leftarrow \Delta_l B$,

$$FR \circ \cap(\mathsf{mem} \setminus ((\Delta_k)^l)\mathsf{id})_{\Delta_l B} = \cap(\mathsf{mem}_A \setminus ((\Delta_k)^l)R) \ . \tag{20}$$

Properties (20) and (19) are equivalent under the assumption of extensionality as shown by Hoogendijk [14]. Note that $\cap S$ denotes the intersection of the l elements of the vector of relations S. Division in a product allegory is of course componentwise division in the base allegories.

Property (20) gives a great deal of insight into the nature of natural transformations. First, the property is easily generalised to:

$$FR \circ \cap(\mathsf{mem} \setminus ((\Delta_k)^l)S)_{\Delta_l C} = \cap(\mathsf{mem}_A \setminus ((\Delta_k)^l)(R \circ S)) \tag{21}$$

for all $R : A \leftarrow B$ and $S \ :\ B \leftarrow \Delta_l C$. Next we require that the membership of a tuple of relators is the tuple of their memberships:

$$\mathsf{mem}.F = \tau\Delta_k(\mathsf{mem}.\mathsf{Proj}_k F) \tag{22}$$

Then, it is straightforward to show that the membership, mem, of relator $F : k \leftarrow l$ is a natural transformation. Indeed

$$\mathsf{mem} \ :\ (\Delta_k)^l \hookleftarrow \Delta_l F \ ,$$

and also

$$\cap(\mathsf{mem} \setminus ((\Delta_k)^l)\mathsf{id})\Delta_l \;:\; \mathsf{F}\Delta_l \hookleftarrow \Delta_k \quad .$$

(For endorelator F these properties simplify to $\mathsf{mem} : \mathrm{Id} \hookleftarrow \mathsf{F}$ and $\mathsf{mem}\backslash\mathsf{id} : \mathsf{F} \hookleftarrow \mathrm{Id}$.) Having established these two properties, the —highly significant— observation that mem and $\cap(\mathsf{mem} \setminus ((\Delta_k)^l)\mathsf{id})\Delta_l$ are the *largest* natural transformations of their types can be made. Finally, and most significantly, suppose F and G are relators with memberships $\mathsf{mem.F}$ and $\mathsf{mem.G}$ respectively. Then the largest natural transformation of type $\mathsf{F} \hookleftarrow \mathsf{G}$ is $\cap(\mathsf{mem.F}\backslash\mathsf{mem.G})$. (We refer the reader to [15] for proofs of all these properties in the case of endorelators, and to [14] in the general case. The key element in the proof is the *identification axiom* which states that the identity function is the largest natural transformation of type $\mathrm{Id} \hookleftarrow \mathrm{Id}$. The identification axiom plays the same role in our theory as the property stated in the introduction that the identity function is the only polymorphic function of type $\mathrm{Id} \leftarrow \mathrm{Id}$ does in Reynolds'.)

The insight that these properties give is that natural transformations between datatypes can only rearrange values; computation on the stored values or invention of new values is prohibited. To see this let us consider each of the properties in turn. A natural transformation of type $\mathrm{Id} \hookleftarrow \mathsf{F}$ constructs values of type A given a structure of type $\mathsf{F}A$. The fact that the membership relation for F is the largest natural transformation of type $\mathrm{Id} \hookleftarrow \mathsf{F}$ says that all values created by such a natural transformation must be members of the structure $\mathsf{F}A$. Similarly, a natural transformation α of type $\mathsf{F} \hookleftarrow \mathsf{G}$ constructs values of type $\mathsf{F}A$ given a structure of type $\mathsf{G}A$. The fact that $\cap(\mathsf{mem.F}\backslash\mathsf{mem.G})$ is the largest natural transformation of type $\mathsf{F} \hookleftarrow \mathsf{G}$ means that $\alpha \subseteq (\mathsf{mem.F}\backslash\mathsf{mem.G})_i$ for each component i of the vector $\mathsf{mem.F}\backslash\mathsf{mem.G}$. According to the interpretation of the division operator, this means that every member of the F-structure created by α is a member of the input G-structure. A proper natural transformation $\alpha : \mathsf{F} \leftarrow \mathsf{G}$ has types $\mathsf{F} \hookleftarrow \mathsf{G}$ and $\mathsf{F} \hookrightarrow \mathsf{G}$. Consequently, a proper natural transformation copies values without loss or duplication.

The natural transformation $\cap(\mathsf{mem} \setminus ((\Delta_k)^l)\mathsf{id})\Delta_l$, the largest natural transformation of type $\mathsf{F}\Delta_l \hookleftarrow \Delta_k$, is called the *canonical fan* of F. It transforms an arbitrary value into an F-structure by non-deterministically creating an F-structure and then copying the given value at all places in the structure. It plays a crucial role in the sequel. (The name "fan" is chosen to suggest the hand-held device that was used in olden times by dignified ladies to cool themselves down.) Rules for computing the canonical fan for all regular relators are as follows. (These are used later in the construction of "zips".)

$$\begin{aligned}
\mathsf{fan.Proj} &= \mathsf{id} && (23)\\
\mathsf{fan.}\Delta\mathsf{F}_k &= \Delta(\mathsf{fan.F}_k) && (24)\\
\mathsf{fan.FG} &= \mathsf{F}(\mathsf{fan.G}) \circ \mathsf{fan.F} && (25)\\
\mathsf{fan.K}_A &= \top\!\!\top_{A,_} && (26)\\
\mathsf{fan.}+ &= (\mathsf{id} \triangledown \mathsf{id})^{\cup} && (27)
\end{aligned}$$

$$\mathsf{fan}.\times \;=\; \mathsf{id} \vartriangle \mathsf{id} \tag{28}$$

$$\mathsf{fan}.\mathsf{T} \;=\; (\![\mathsf{id}\otimes \,;\, (\mathsf{fan}.\otimes)^{\cup}]\!)^{\cup} \tag{29}$$

(where T is the tree relator induced by $\otimes$).

3 Commuting Datatypes: Examples

In this section we want to argue that the notion that two datatypes "commute" is a common occurrence.

The best known example of a commutativity property is the fact that two lists of the same length can be mapped into single list of pairs whereby

$$([a_1, a_2, \ldots], [b_1, b_2, \ldots]) \mapsto [(a_1, b_1), (a_2, b_2), \ldots]$$

The function that performs this operation is known as the "zip" function to functional programmers. Zip commutes a pair of lists into a list of pairs.

Other specific examples of commutativity properties are easy to invent. For instance, it is not difficult to imagine generalising zip to a function that commutes m lists each of length n into n lists each of length m. Indeed, this latter function is also well known under the name *matrix transposition*. Another example is the function that commutes a tree of lists all of the same length into a list of trees all of the same shape. There is also a function that "broadcasts" a value to all elements of a list —thus

$$(a, [b_1, b_2, \ldots]) \mapsto [(a, b_1), (a, b_2), \ldots]$$

— . That is, the datatype an element of type A paired with (a list of elements of type B) is "commuted" to a list of (element of type A paired with an element of type B). More precisely, for each A, the family of broadcasts indexed by B is a natural transformation of type $\mathsf{List}(A\times) \hookleftarrow (A\times)\mathsf{List}$; the two datatypes being "commuted" are thus $(A\times)$ and List. This list broadcast is itself an instance of a subfamily of the operations that we discuss later. In general, a *broadcast* operation copies a given value to all locations in a given data structure.

A final example of a generalised zip would be the (polymorphic) operation that maps values of type $(A{+}B)\times(C{+}D)$ to values of type $(A{\times}C){+}(B{\times}D)$, i.e. commutes a product of disjoint sums to a disjoint sum of products. A necessary restriction is that the elements of the input pair of values have the same "shape", i.e. both be in the left component of the disjoint sum or both be in the right component.

In general then, a *zip* operation transforms F-structures of G-structures to G-structures of F-structures. Typically, "zips" are partial since they are only well-defined on structures of the same shape. As we shall see, they may also be non-deterministic; that is, a "zip" is a relation that need not be simple. Finally, the arity of the two datatypes, F and G, need not be the same; for example, the classical zip function maps pairs of lists to lists of pairs, and pairing has arity $1{\leftarrow}2$ whereas list formation has arity $1{\leftarrow}1$.

3.1 Structure Multiplication

A good example of the beauty of the "zip" generalisation is afforded by what we shall call "structure multiplication". (This example we owe to D.J. Lillie [private communication, December 1994].) A simple, concrete example of structure multiplication is the following. Given two lists $[a_1, a_2, \ldots]$ and $[b_1, b_2, \ldots]$ form a matrix in which the (i, j)th element is the pair (a_i, b_j). We call this "structure multiplication" because the input type is the product $\mathsf{List}A \times \mathsf{List}B$ for some types A and B.

Given certain basic functions, this task may be completed in one of two ways. The first way has two steps. First, the list of a's is broadcast over the list of b's to form the list

$$[([a_1, a_2, \ldots], b_1), ([a_1, a_2, \ldots], b_2), \ldots]$$

Then each b is broadcast over the list of a's. The second way is identical but for an interchange of "a" and "b".

Both methods return a list of lists, but the results are not identical. The connection between the two results is that one is the transpose of the other. The two methods and the connection between them are summarised in the following diagram.

$$\begin{tikzcd}
 & \mathsf{List}A \times \mathsf{List}B \arrow[dl] \arrow[dr] & \\
\mathsf{List}(\mathsf{List}A \times B) \arrow[d] & & \mathsf{List}(A \times \mathsf{List}B) \arrow[d] \\
\mathsf{ListList}(A \times B) \arrow[rr, leftrightarrow] & & \mathsf{ListList}(A \times B)
\end{tikzcd}$$

The point we want to make is that there is an obvious generalisation of this procedure: replace $\mathsf{List}A$ by $\mathsf{F}A$ and $\mathsf{List}B$ by $\mathsf{G}B$ for some arbitrary relators F and G. Doing so leads to the realisation that every step involves a "zip" operation (i.e. commuting the order of a pair of datatypes). This is made explicit in the diagram below.

$$\begin{tikzcd}
 & \mathsf{FA} \times \mathsf{GB} \arrow[dl, "(\mathsf{zip}.(\mathsf{FA}\times).\mathsf{G})_{\mathsf{B}}"'] \arrow[dr, "(\mathsf{zip}.(\times\mathsf{GB}).\mathsf{F})_{\mathsf{A}}"] & \\
\mathsf{G}(\mathsf{FA} \times \mathsf{B}) \arrow[d, "\mathsf{G}(\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_{\mathsf{A}}"'] & & \mathsf{F}(\mathsf{A} \times \mathsf{GB}) \arrow[d, "\mathsf{F}(\mathsf{zip}.(\mathsf{A}\times).\mathsf{G})_{\mathsf{B}}"] \arrow[dll, "(\mathsf{zip}.\mathsf{F}(\mathsf{A}\times).\mathsf{G})_{\mathsf{B}}" description] \\
\mathsf{GF}(\mathsf{A} \times \mathsf{B}) & & \mathsf{FG}(\mathsf{A} \times \mathsf{B}) \arrow[ll, "(\mathsf{zip}.\mathsf{F}.\mathsf{G})_{\mathsf{A}\times\mathsf{B}}"]
\end{tikzcd}$$

In order to make evident which datatypes are being "commuted" at each step, each arrow has been labelled by an expression involving a "zip" term. A "zip" takes the form $\mathsf{zip.F.G}$ for some datatypes F and G. In the absence of a formal specification (to be given later) one should interpret $\mathsf{zip.F.G}$ as a family of relations indexed by types such that $(\mathsf{zip.F.G})_{\mathsf{A}} : \mathsf{GFA} \leftarrow \mathsf{FGA}$.

An additional edge has been added to the diagram to show the usefulness of generalising the notion of commutativity beyond just broadcasting; this additional inner edge shows how the commutativity of the diagram can be decomposed into smaller parts[4]. Specifically, in order to show that the whole diagram commutes (in the standard categorical sense of commuting diagram) it suffices to show that two smaller diagrams commute. Specfically, the following two equalities must be established:

$$(\mathsf{zip.F}(\mathsf{A}\times).\mathsf{G})_{\mathsf{B}} \;=\; (\mathsf{zip.F.G})_{\mathsf{A}\times\mathsf{B}} \circ \mathsf{F}(\mathsf{zip}.(\mathsf{A}\times).\mathsf{G})_{\mathsf{B}} \tag{30}$$

and

$$(\mathsf{zip.F}(\mathsf{A}\times).\mathsf{G})_{\mathsf{B}} \circ (\mathsf{zip}.(\times\mathsf{GB}).\mathsf{F})_{\mathsf{A}} \;=\; \mathsf{G}(\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_{\mathsf{A}} \circ (\mathsf{zip}.(\mathsf{FA}\times).\mathsf{G})_{\mathsf{B}} \tag{31}$$

We shall in fact design our definition of "commuting datatypes" in such a way that these two equations are satisfied (almost) by definition. In other words, our notion of "commuting datatypes" is such that the commutativity of the above diagram is automatically guaranteed.

3.2 Strength

Several scientists have argued that the notion of functor is too general to capture the notion of a datatype as understood by programmers. Moggi [27] argues that the notion of "strength" is fundamental to computation, "strength" being defined as follows.

[4] The additional edge together with the removal of the right-pointing edge in the bottom line seem to make the diagram asymmetric. But, of course, there are symmetric edges. Corresponding to the added diagonal edge there is an edge connecting $\mathsf{G}(\mathsf{FA} \times \mathsf{B})$ and $\mathsf{FG}(\mathsf{A} \times \mathsf{B})$ but only one of these edges is needed in the argument that follows.

Definition 32 (Strength) A natural transformation $\mathsf{str}_{A,B} : F(A \times B) \hookleftarrow FA \times B$ is said to be a *strength* of relator F iff $\mathsf{str}_{A\,,\,B}$ is a function that behaves coherently with respect to product in the following sense. First, the diagram

$$\begin{tikzcd}
F(A \times 1) \arrow[dr, "\mathsf{Frid}_A"'] & & FA \times 1 \arrow[ll, "\mathsf{str}_{A,1}"'] \arrow[dl, "\mathsf{ridF}_A"] \\
& FA &
\end{tikzcd}$$

(where $\mathsf{rid}_A : A \leftarrow A \times 1$ is the obvious natural isomorphism) commutes. Second, the diagram

$$\begin{tikzcd}
FA \times (B \times C) \arrow[dd, "\mathsf{str}_{A\,,\,B\times C}"'] & (FA \times B) \times C \arrow[l, "\mathsf{ass}_{FA,B,C}"'] \arrow[d, "\mathsf{str}_{A,B} \times \mathsf{id}_C"] \\
& F(A \times B) \times C \arrow[d, "\mathsf{str}_{A\times B\,,\,C}"] \\
F(A \times (B \times C)) & F((A \times B) \times C) \arrow[l, "\mathsf{Fass}_{A,B,C}"]
\end{tikzcd}$$

(where $\mathsf{ass}_{A,B,C} : A \times (B \times C) \leftarrow (A \times B) \times C$ is the obvious natural isomorphism) commutes as well. A relator that has at least one strength is said to be *strong*.
$\square$

The idea behind "strength" is very simple. A relator F is "strong" if, for each pair of types A and B, it is possible to broadcast a given value of type B to every element in an F-structure of A's. The broadcasting operation is what Moggi calls the "strength" of the relator.

The type of the "strength" $\mathsf{str}_{A,B}$ of relator F is the same as the type of $(\mathsf{zip}.(\times A).F)_B$, namely $F(A \times B) \leftarrow FA \times B$. We shall argue that, if F and the family of relators $(\times A)$ are included in a class of commuting relators, then any relation satisfying the requirements of $(\mathsf{zip}.(\times A).F)_B$ also satisfies the definition of $\mathsf{str}_{A,B}$.

Let us begin with an informal scrutiny of the definition of strength. In the introduction to this section we remarked that a broadcast operation (a "strength") is an example of a zip. Specifically, a broadcast operation is a zip of the form $(\mathsf{zip}.(\times A).F)_B$. Paying due attention to the fact that the relator F is a parameter of the definition, we observe that all the natural transformations involved in the definition of strength are special cases of a broadcast operation and thus of zips.

In the first diagram there are two occurrences of the canonical isomorphism rid. In general, we recognise a projection of type $A \leftarrow A \times B$ as a broadcast where the parameter F is instantiated to K_A, the relator that is constantly A when applied to objects and is the identity on A when applied to arrows. Thus rid_A is $(\mathsf{zip}.(\times 1).K_A)_B$ for some arbitrary B. In words, rid_A commutes the relators $(\times 1)$ and K_A. Redrawing the first diagram above, using that all the arrows are broadcasts and thus zips, we get the following diagram[5].

$$\begin{tikzcd}
F(A \times 1) \arrow[dr, "F(\mathsf{zip}.(\times 1).(K_A))"'] & & FA \times 1 \arrow[ll, "(\mathsf{zip}.(\times 1).F)K_A"'] \arrow[dl, "\mathsf{zip}.(\times 1).(K_{FA})"] \\
& FA &
\end{tikzcd}$$

Expressed as an equation, this is the requirement that

$$\mathsf{zip}.(\times 1).(K_{FA}) \;=\; F(\mathsf{zip}.(\times 1).(K_A)) \circ (\mathsf{zip}.(\times 1).F)K_A \tag{33}$$

Now we turn to the second diagram in the definition of strength. Just as we observed that rid is an instance of a broadcast and thus a zip, we also observe that ass is a broadcast and thus a zip. Specifically, $\mathsf{ass}_{A,B,C}$ is $(\mathsf{zip}.(\times C).(A\times))_B$. Once again, every edge in the diagram involves a zip operation! That is not all. Yet more zips can be added to the diagram. For our purposes it is crucial to observe that the bottom left and middle right nodes —the nodes labelled $F(A \times (B \times C))$ and $F(A \times B) \times C$— are connected by the edge $(\mathsf{zip}.(\times C).F(A\times))_B$.

$$\begin{tikzcd}
FA \times (B \times C) \arrow[dd, "(\mathsf{zip}.(\times(B\times C)).F)_A"'] & & (FA \times B) \times C \arrow[ll, "(\mathsf{zip}.(\times C).(FA\times))_B"'] \arrow[d, "(\mathsf{zip}.(\times B).F)_A \times \mathsf{id}_C"] \\
& & F(A \times B) \times C \arrow[dll, "(\mathsf{zip}.(\times C).F(A\times))_B"'] \arrow[d, "(\mathsf{zip}.(\times C).F)_{A\times B}"] \\
F(A \times (B \times C)) & & F((A \times B) \times C) \arrow[ll, "F(\mathsf{zip}.(\times C).(A\times))_B"]
\end{tikzcd}$$

[5] To be perfectly correct we should instantiate each of the transformations at some arbitrary B. We haven't done so because the choice of which B in this case is truly irrelevant.

This means that we can decompose the original coherence property into a combination of two properties of zips. These are as follows. First, the lower triangle:

$$(\mathsf{zip}.(\times\mathsf{C}).\mathsf{F}(\mathsf{A}\times))_{\mathsf{B}} \;=\; \mathsf{F}(\mathsf{zip}.(\times\mathsf{C}).(\mathsf{A}\times))_{\mathsf{B}} \circ (\mathsf{zip}.(\times\mathsf{C}).\mathsf{F})_{\mathsf{A}\times\mathsf{B}} \tag{34}$$

Second, the upper rectangle:

$$(\mathsf{zip}.(\times(\mathsf{B}\times\mathsf{C})).\mathsf{F})_{\mathsf{A}} \circ (\mathsf{zip}.(\times\mathsf{C}).(\mathsf{F}\mathsf{A}\times))_{\mathsf{B}} = (\mathsf{zip}.(\times\mathsf{C}).\mathsf{F}(\mathsf{A}\times))_{\mathsf{B}} \circ (\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_{\mathsf{A}}\times\mathsf{id}_{\mathsf{C}} \tag{35}$$

Note the strong similarity between (33) and (34). They are both instances of one equation parameterised by three different datatypes. There is also a similarity between these two equations and (30); the latter is an instance of the same parameterised equation after taking the converse of both sides and assuming that $\mathsf{zip}.\mathsf{F}.\mathsf{G} = (\mathsf{zip}.\mathsf{G}.\mathsf{F})^{\cup}$. Less easy to spot is the similarity between (31) and (35). As we shall see, however, both are instances of one equation parameterised again by three different datatypes except that (35) is obtained by applying the converse operator to both sides of the equation and again assuming that $\mathsf{zip}.\mathsf{F}.\mathsf{G} = (\mathsf{zip}.\mathsf{G}.\mathsf{F})^{\cup}$.

4 The Requirement

In this section we formulate precisely what we mean by two datatypes commuting.

Looking again at the examples above, the first step towards an abstract problem specification is clear enough. Replacing "list", "tree" etc. by "datatype F" the problem is to specify an operation $\mathsf{zip}.\mathsf{F}.\mathsf{G}$ for given datatypes F and G that maps FG-structures into GF-structures.

Note that the informal language we use here seems to imply that we consider only endo relators (relators of arity $1{\leftarrow}1$). After all, the composition FG is meaningless if the source arity of F is not the same as the target arity of G. If $\mathsf{F} : m \leftarrow k$ and $\mathsf{G} : n \leftarrow l$ then $({}^{n}\mathsf{F})(\mathsf{G}^{k})$ is a meaningful composition, as too is $(\mathsf{G}^{m})({}^{l}\mathsf{F})$, both having arity $n{*}m \leftarrow l{*}k$. (Recall that for $\mathsf{H} : l \leftarrow m$ we have $\mathsf{H}^{k} : l{*}k \leftarrow m{*}k$ and ${}^{k}\mathsf{H} : k{*}l \leftarrow k{*}m$.) Thus, to be perfectly precise we should talk about mapping $({}^{n}\mathsf{F})(\mathsf{G}^{k})$–structures to $(\mathsf{G}^{m})({}^{l}\mathsf{F})$–structures.

Being able to handle relators of arbitrary arity and not restricting ourselves to endorelators is an important element of our development —were we to restrict ourselves to just endorelators then we could not even handle the standard example of zipping a pair of lists since product is not endo— but nevertheless we often omit arity information in our informal motivation of some elements of our requirement. In all formal statements we do supply the arity information. The point is that these details can easily be inferred by a process of arity checking (using the rules given in section 2) but their inclusion in the first instance is a burdensome complication.

The first step may be obvious enough, subsequent steps are less obvious. The nature of our requirements is influenced by the relationship between parametric polymorphism and naturality properties discussed earlier but takes place at a higher level. We consider the datatype F to be fixed and specify a collection of operations $\mathsf{zip.F.G}$ indexed by the datatype G. (The fact that the index is a datatype rather than a type is what we mean by "at a higher level".) Such a family forms what we call a collection of "half-zips". The requirement is that the collection be "parametric" in G. That is, the elements of the family $\mathsf{zip.F}$ should be "logically related" to each other. The precise formulation of this idea leads us to three requirements on "half-zips". The symmetry between F and G, lost in the process of fixing F and varying G, is then restored by the simple requirement that a zip is both a half-zip and the converse of a half-zip.

The division of our requirements into "half-zips" and "zips" corresponds to the way that zips are constructed. Specifically, we construct a half-zip $\mathsf{zip.F.G}$ for each datatype F in the class of regular datatypes and an arbitrary datatype G. That is to say, for each datatype F we construct the function $\mathsf{zip.F}$ on datatypes which, for an arbitrary datatype G, gives the corresponding zip operation $\mathsf{zip.F.G}$. The function is constructed to meet the requirement that it define a collection of half-zips; subsequently we show that if the collection is restricted to regular datatypes G then each half-zip is in fact a zip.

A further subdivision of the requirements is into naturality requirements and requirements that guarantee that the algebraic structure of pointwise definition of relators is respected (for example, the associativity of functional composition of relators is respected). These we discuss in turn.

4.1 Naturality Requirements

Our first requirement is that $\mathsf{zip.F.G}$ be natural. That is to say, its application to an FG-structure should not in any way depend on the values in that structure. Suppose that $\mathsf{F} : \mathfrak{m} \leftarrow \mathsf{k}$ and $\mathsf{G} : \mathfrak{n} \leftarrow \mathfrak{l}$. Then we demand that

$$\mathsf{zip.F.G} \;:\; (\mathsf{G}^{\mathfrak{m}})({}^{\mathfrak{l}}\mathsf{F}) \leftarrow ({}^{\mathfrak{n}}\mathsf{F})(\mathsf{G}^{\mathsf{k}}) \quad . \tag{36}$$

Thus a zip is a proper natural transformation indexed by an $\mathfrak{l}{*}\mathsf{k}$ matrix of types each member of the family being an $\mathfrak{n}{*}\mathfrak{m}$ matrix of relations.

As forewarned, arity information is included in the formal statement (36) although not in the informal discussion preceding it. For endorelators the requirement is much simpler:

$$\mathsf{zip.F.G} \;:\; \mathsf{GF} \leftarrow \mathsf{FG} \quad .$$

Our advice is thus to ignore all tupling and projection operators (the superscripts in this case) on a first reading.

Note that we require $\mathsf{zip.F.G}$ to be a *proper* natural transformation since for a zip operation on a structure no loss or duplication of values should occur.

Demanding naturality is not enough. Somehow we want to express that all the members of the family $\mathsf{zip.F}$ of zip operations for different datatypes G and H are related. For instance, if we have a natural transformation $\alpha : \mathsf{G} \leftarrow \mathsf{H}$ then $\mathsf{zip.F.G}$ and $\mathsf{zip.F.H}$ should be "coherent" with the transformation α. That is to say, having both zips and α, there are two ways of transforming FH-structures into GF-structures; these should effectively be the same.

One way is first transforming an FH-structure into an FG-structure using $\mathsf{F}\alpha$, (i.e. applying the transformation α to each H-structure inside the F-structure) and then commuting the FG-structure into a GF-structure using $\mathsf{zip.F.G}$.

Another way is first commuting an FH-structure into an HF-structure with $\mathsf{zip.F.H}$ and then transforming this H-structure into a G-structure (both containing F-structures) using $\alpha\mathsf{F}$. So, we have the following diagram.

$$\begin{tikzcd}
\mathsf{FG} \arrow[d, "\mathsf{zip.F.G}"'] & \mathsf{FH} \arrow[l, "\mathsf{F}\alpha"'] \arrow[d, "\mathsf{zip.F.H}"] \\
\mathsf{GF} & \mathsf{HF} \arrow[l, "\alpha\mathsf{F}"]
\end{tikzcd}$$

One might suppose that an equality is required, i.e.

$$\alpha\mathsf{F} \circ \mathsf{zip.F.H} \quad = \quad \mathsf{zip.F.G} \circ \mathsf{F}\alpha \tag{37}$$

for all natural transformations $\alpha : \mathsf{G} \leftarrow \mathsf{H}$. But this requirement is too severe for two reasons.

The first reason is that if α is not functional, i.e. α is a non-deterministic transformation, the rhs of equation (37) may be more non-deterministic than the lhs because of the possible multiple occurrence of α. Take for instance $\mathsf{F} := \mathsf{List}$ and $\mathsf{G} = \mathsf{H} := \times$, i.e. $\mathsf{zip.F.G}$ and $\mathsf{zip.F.H}$ are both the inverse of the zip function on a pair of lists, and take $\alpha := \mathsf{id} \cup \mathsf{swap}$, i.e. α non-deterministically swaps the elements of a pair or not. Then $\alpha\mathsf{F} \circ \mathsf{zip.F.H}$ unzips a list of pairs into a pair of lists and swaps the lists or not. On the other hand, $\mathsf{zip.F.G} \circ \mathsf{F}\alpha$ first swaps some of the elements of a list of pairs and then unzips it into a pair of lists.

The second reason is that, due to the partiality of zips, the domain of the left side of (37) may be smaller than that of the right.

As a concrete example, suppose $\mathsf{listify}$ is a polymorphic function that constructs a list of the elements stored in a tree. The way that the tree is traversed (inorder, preorder etc.) is immaterial; what is important is that $\mathsf{listify}$ is a natural transformation of type $\mathsf{List} \leftarrow \mathsf{Tree}$. Now suppose we are given a list of trees. Then it can be transformed to a list of lists by "listify"ing each tree in the list, i.e. by applying the (appropriate instance of the) function $\mathsf{List}(\mathsf{listify})$. If all the trees in the list have the same shape, a list of lists can also be obtained by first commuting the list of trees to a tree of lists (all of the same length) and then "listify"ing the tree structure. That is we apply the (appropriate instance of the) function $(\mathsf{listify})\mathsf{List} \circ \mathsf{zip.List.Tree}$. The two lists of lists will not be the same: if the size of the original list is m and the size of each tree in the list is n then the first method will construct m lists each of length n whilst the second method will construct n lists each of length m. However the two lists of lists are "zips" of each other ("transposes" would be the more conventional terminology). This is expressed by the commutativity of the following diagram in the case that the input type $\mathsf{List}(\mathsf{Tree}A)$ is restricted to lists of trees of the same shape.

$$\begin{tikzcd}
\mathsf{List}(\mathsf{List}A) \arrow[d, "(\mathsf{zip.List.List})_A"'] & & \mathsf{List}(\mathsf{Tree}A) \arrow[ll, "\mathsf{List}(\mathsf{listify})_A"'] \arrow[d, "(\mathsf{zip.List.Tree})_A"] \\
\mathsf{List}(\mathsf{List}A) & & \mathsf{Tree}(\mathsf{List}A) \arrow[ll, "\mathsf{listify}_{\mathsf{List}A}"]
\end{tikzcd}$$

Note however that if we view both paths through the diagram as partial relations of type $\mathsf{List}(\mathsf{List}A) \leftarrow \mathsf{List}(\mathsf{Tree}A)$ then the upper path (via $\mathsf{List}(\mathsf{List}A)$) includes the lower path (via $\mathsf{Tree}(\mathsf{List}A)$). This is because the function $\mathsf{List}(\mathsf{listify})_A$ may construct a list of lists all of the same length (as required by the subsequent zip operation) even though all the trees in the given list of trees may not all have the same shape. The requirement on the trees is that they all have the same size, which is weaker than their all having the same shape.

Both examples show that we have to relax requirement (37) using an inclusion instead of equality. Having this inclusion, the requirement for α can be relaxed as well. So, the requirement becomes

$$\alpha F \circ \mathsf{zip}.F.H \subseteq \mathsf{zip}.F.G \circ F\alpha \quad \text{for all } \alpha : G \hookleftarrow H \quad .$$

Including arity information, the formal statement of the requirement is that for all relators $F : m \leftarrow k$ and $G , H : n \leftarrow l$, and all $\alpha : G \hookleftarrow H$,

$$(\alpha^m)(^l F) \circ \mathsf{zip}.F.H \subseteq \mathsf{zip}.F.G \circ (^n F)(\alpha^k) \quad . \tag{38}$$

4.2 Pointwise Integrity

The variable-free mechanism we have introduced for "pointwise closing" a class of relators allows some freedom in the manner in which relators are composed. Formally, the relators form a category under functional composition, and the tupling and projection operators are related by the characteristic equation

$$F = \Delta G_k \equiv \mathsf{Proj}_k F = G_k \quad .$$

(Note that the right side of this equivalence is an equation between mappings following the convention explained earlier. Thus it is true if for all i, $0 \leq i < k$, $\mathsf{Proj}_i F = G_i$.) Our second set of requirements guarantee that this algebraic structure is respected by the mapping $\mathsf{zip}.F$.

We begin with tupling and projection. In view of arity considerations the obvious requirements are:

$$\mathsf{zip.F.G} = \tau\Delta_n(\mathsf{zip.F.Proj}_n\mathsf{G}) \tag{39}$$

where n is the arity of the target of G —the zip of a tuple is the tuple of the zips— and

$$\mathsf{zip.F.Proj} = (\mathsf{id}^m)\mathsf{F}(\mathsf{Proj}^k) \tag{40}$$

for each projection relator $\mathsf{Proj} : 1 \leftarrow l$, assuming $\mathsf{F} : m \leftarrow k$.

In fact, (39) becomes redundant when we introduce requirement (41) on the composition of relators.

For our final requirement we consider the monoid structure of functors under composition. Fix functor F and consider the collection of zips, $\mathsf{zip.F.G}$, indexed by (endo)functor G. Since the (endo)functors form a monoid it is required that the mapping $\mathsf{zip.F}$ is a monoid homomorphism.

In order to formulate this requirement precisely we let ourselves be driven by type considerations. The requirement is that $\mathsf{zip.F.GH}$ be some composition of $\mathsf{zip.F.G}$ and $\mathsf{zip.F.H}$ of which $\mathsf{zip.F.Id}$ is the identity. But the type of $\mathsf{zip.F.GH}$,

$$\mathsf{zip.F.GH} \ : \ \mathsf{GHF} \leftarrow \mathsf{FGH} \ \ ,$$

demands that the datatype F has to be "pushed" through GH leaving the order of G and H unchanged. With $\mathsf{zip.F.G}$ we can swap the order of F and G, with $\mathsf{zip.F.H}$ the order of F and H. Thus transforming FGH to GHF can be achieved as shown below.

$$\mathsf{GHF} \xleftarrow{\mathsf{G(zip.F.H)}} \mathsf{FHG} \xleftarrow{\mathsf{(zip.F.G)H}} \mathsf{FGH}$$

So, informally, we demand that

$$\mathsf{zip.F.GH} = \mathsf{G(zip.F.H)} \circ \mathsf{(zip.F.G)H} \ \ .$$

Moreover, in order to guarantee that $\mathsf{zip.F.GId} = \mathsf{zip.F.G} = \mathsf{zip.F.IdG}$ we require that

$$\mathsf{zip.F.Id} = \mathsf{idF} \ \ .$$

Formally, the demand is that, for all $\mathsf{F} : m \leftarrow k$, $\mathsf{G} : n \leftarrow l$ and $\mathsf{H} : l \leftarrow o$,

$$\mathsf{zip.F.GH} \ = \ (\mathsf{G}^m)(\mathsf{zip.F.H}) \circ (\mathsf{zip.F.G})(\mathsf{H}^k) \ \ , \tag{41}$$

and, for $\mathsf{F} : m \leftarrow k$ and the identity relator $\mathsf{Id} : l \leftarrow l$,

$$\mathsf{zip.F.Id} = (\mathsf{id}^{l*m})({}^l\mathsf{F}) \ \ . \tag{42}$$

In order to verify that $\mathsf{zip.F}$ is indeed a monoid homomorphism we make the monoid explicit. Define (for fixed datatype F) the monoid $\mathcal{M}$ as follows. The elements are pairs consisting of a natural transformation, α , and a functor, G, where

$$(\alpha , \mathsf{G}) \in \mathcal{M} \ \ \equiv \ \ \alpha \ : \ \mathsf{GF} \xleftarrow{\mathsf{Fun}} \mathsf{FG}$$

Define composition in the following way:

$$(\alpha, G) \cdot (\beta, H) \triangleq (G\beta \circ \alpha H \ , \ GH)$$

That $(G\beta \circ \alpha H \ , \ GH)$ is an element of $\mathcal{M}$ is, by definition, $G\beta \circ \alpha H \ : \ GHF \leftarrow FGH$ which follows from $\beta \ : \ HF \leftarrow FH$ and $\alpha \ : \ GF \leftarrow FG$. It is easily seen that "$\cdot$" has unit $(\mathsf{id}F, \mathrm{Id})$ and is associative.

Now, define $f(G) = (\mathsf{zip}.F.G, G)$. Then $\mathsf{zip}.F$ is a monoid homomorphism if

$$f(GH) = f(G) \cdot f(H)$$

and

$$f(\mathrm{Id}) = (\mathsf{id}F, \mathrm{Id}) \quad .$$

Expanding the definition of f, we thus demand

$$\mathsf{zip}.F.GH = G(\mathsf{zip}.F.H) \circ (\mathsf{zip}.F.G)H$$

and

$$\mathsf{zip}.F.\mathrm{Id} = \mathsf{id}F \quad .$$

(Note that $\mathsf{id}F \ : \ \mathrm{Id}F \leftarrow F\mathrm{Id}$.) .

4.3 Half Zips and Commuting Relators

Apart from the very first of our requirements ((36), the requirement that $\mathsf{zip}.F.G$ be natural), all the other requirements have been requirements on the nature of the mapping $\mathsf{zip}.F$. Roughly speaking, (38) demands that it be parametric, (39) and (40) that it respect tupling and projection, and (41) and (42) that it be functorial. Of these requirements, (39) and (42) are redundant. ((39) can be derived from (40) and (41); it can then be used in combination with (40) to derive (42).) We find it useful to bundle the (non-redundant set of) requirements together into the definition of something that we call a "half zip".

Definition 43 (Half Zip) Consider a fixed relator $F : m \leftarrow k$ and a pointwise closed class of relators $\mathcal{G}$. Then the members of the collection $\mathsf{zip}.F.G$, where G ranges over $\mathcal{G}$, are called *half-zips* iff
(a) $\mathsf{zip}.F.G \ : \ (G^m)({}^lF) \leftarrow ({}^nF)(G^k)$, for each $G : n \leftarrow l$
(b) $\mathsf{zip}.F.\mathsf{Proj} = (\mathsf{id}^m)F(\mathsf{Proj}^k)$ for all $\mathsf{Proj} : 1 \leftarrow l$,
(c) $\mathsf{zip}.F.GH \ = \ (G^m)(\mathsf{zip}.F.H) \circ (\mathsf{zip}.F.G)(H^k)$ for all $G : n \leftarrow l$ and $H : l \leftarrow o$,
(d) $(\alpha^m)({}^lF) \circ \mathsf{zip}.F.H \subseteq \mathsf{zip}.F.G \circ ({}^nF)(\alpha^k)$ for each $\alpha : G \hookleftarrow H$ where $G, H : n \leftarrow l$.
□

Note that for $F : m \leftarrow k$ and $G : n \leftarrow l$, we have

$$\mathsf{zip}.F.G \ : \ n{*}m \leftarrow l{*}k$$

and

$$(\mathsf{zip.G.F})^{\cup} : m{*}n \leftarrow k{*}l \quad .$$

So for non-endo F and G they do not have the same arity. The source and target arities are clearly related by matrix transposition, i.e. the relator τ. That is,

$$\tau(\mathsf{zip.G.F})^{\cup}\tau : n{*}m \leftarrow l{*}k \quad .$$

So, the general definition becomes:

Definition 44 (Commuting Relators) The half-zip zip.F.G is said to be a *zip* of (F, G) if there exists a half-zip zip.G.F such that

$$\mathsf{zip.F.G} = \tau(\mathsf{zip.G.F})^{\cup}\tau$$

We say that datatypes F and G *commute* if there exists a zip for (F, G).
□

5 Consequences

In this section we address two concerns. First, it may be the case that our requirement is so weak that it has many trivial solutions. We show that, on the contrary, the requirement has a number of consequences that guarantee that there are no trivial solutions. On the other hand, it could be that our requirement for datatypes to commute is so strong that it is rarely satisfied. Here we show that the requirement can be met for all regular datatypes. (Recall that the "regular" datatypes are the sort of datatypes that one can define in a conventional functional programming language.) Moreover, we can even prove the remarkable result that for the regular relators our requirement has a *unique* solution.

5.1 Shape Preservation

Zips are partial operations: zip.F.G should map F–structures of (G–structures of the same shape) into G–structures of (F–structures of the same shape). This requirement is, however, not explicitly stated in our formalisation of being a zip. In this subsection we show that it is nevertheless a consequence of that formal requirement. In particular we show that a half zip always constructs G–structures of (F–structures of the same shape). We in fact show a more general result that forms the basis of the uniqueness result for regular relators.

Let us first recall how shape considerations are expressed. The function $!_{\mathsf{A}}$ is the function of type $1 \leftarrow \mathsf{A}$ that replaces a value by the unique element of the unit type, 1. Also, for an arbitrary function f, Ff maps an F–structure to an F–structure of the same shape, replacing each value in the input structure by the result of applying f to that value. Thus $\mathsf{F}!_{\mathsf{A}}$ maps an F–structure (of A's) to an F–structure of the same shape in which each value in the input structure has been replaced by the unique element of the unit type. We can say that

$(\mathsf{F}!_A)x$ *is* the shape of the F–structure x, and $\mathsf{F}!_A \circ f$ *is* the shape of the result of applying function f.

Now, for a natural transformation α of type $\mathsf{F} \leftarrow \mathsf{G}$, the shape characteristics of α in general are determined by α_1, since

$$\mathsf{F}!_A \circ \alpha_A = \alpha_1 \circ \mathsf{G}!_A$$

That is, the shape of the result of applying α_A is completely determined by the behaviour of α_1. The shape characteristics of $\mathsf{zip.F.G}$, in particular, are determined by $(\mathsf{zip.F.G})_1$ since

$$\mathsf{GF}!_A \circ (\mathsf{zip.F.G})_A = (\mathsf{zip.F.G})_1 \circ \mathsf{FG}!_A$$

Our shape requirement is that a half zip maps an F–G–shape into a G–F–shape in which all F–shapes equal the original F–shape. This we can express by a single equation relating the behaviour of $(\mathsf{zip.F.G})_1$ to that of $\mathsf{fan.G}$. Specifically, we note that $(\mathsf{fan.G})_{\mathsf{F}1}$ generates from a given F-shape, x, an arbitrary G-structure in which all elements equal x, and thus have the same F–shape. On the other hand, $\mathsf{F}(\mathsf{fan.G})_1$, when applied to x, generates F-structures with shape x containing arbitrary G-shapes. The shape requirement (for endorelators) is thus satisfied if we can establish the property

$$(\mathsf{fan.G})_{\mathsf{F}1} \;=\; (\mathsf{zip.F.G})_1 \circ \mathsf{F}(\mathsf{fan.G})_1 \quad . \tag{45}$$

This property is an immediate consequence of the following lemma (stated in full generality).

Suppose $\mathsf{F} : k \leftarrow l$ and $\mathsf{G} : m \leftarrow n$ are datatypes. Then, if $\mathsf{fan.G}$ is the canonical fan of G,

$$((\mathsf{fan.G})^k)\mathsf{F} \;=\; (\mathsf{zip.F.G})((\Delta_n)^l) \circ ({}^m\mathsf{F})((\mathsf{fan.G})^l) \quad . \tag{46}$$

From equation (45) it also follows that the range of $(\mathsf{zip.F.G})_1$ is the range of $(\mathsf{fan.G})_{\mathsf{F}1}$, i.e. arbitrary G-structures of which all elements are the same, but arbitrary, F-shape.

A more general version of (46) is obtained by considering the so-called *fan function*. Recalling the characterising property of the membership relation (20), we define the mapping $\widehat{\mathsf{F}}$ (with the same arity as F, namely $k \leftarrow l$) by

$$\widehat{\mathsf{F}}R = \mathsf{F}R \circ \sqcap(\mathsf{mem} \setminus ((\Delta_k)^l)\mathsf{id})_{\Delta_l B} \quad , \tag{47}$$

for all $R : A \leftarrow \Delta_l B$. (Note that $\widehat{\mathsf{F}}$ is a partial mapping since it is only defined on relations with source a vector of l instances of the same object.) Then the generalisation of (46) is the following lemma.

Suppose $\mathsf{F} : k \leftarrow l$ and $\mathsf{G} : m \leftarrow n$ are datatypes. Then, if $\widehat{\mathsf{G}}$ is the fan function of G,

$$(\widehat{\mathsf{G}}^k)({}^n\mathsf{F})R = (\mathsf{zip.F.G})_A \circ ({}^m\mathsf{F})(\widehat{\mathsf{G}}^l)R \quad , \tag{48}$$

for all $R : A \leftarrow ((\Delta_n)^l)B$.

It is (48) that often uniquely characterises $\mathsf{zip.F.G}$.

5.2 Commuting relators

One reason why our requirements might have trivial solutions is that they are expressed in terms of lax natural transformations. Requiring properness of a natural transformation is stronger. The next lemma establishes a properness result for zips on commuting datatypes; it proves to be the key in showing that certain zips are unique.

Let ξ denote a class of commuting datatypes. Then for all $\mathsf{F} : \mathsf{k} \leftarrow \mathsf{l}$, and $\mathsf{G}\,,\mathsf{H} : \mathsf{m} \leftarrow \mathsf{n}$ in ξ and all families of *functions* α such that $\alpha : \mathsf{G} \leftarrow \mathsf{H}$,

$$({}^{m}\mathsf{F})(\alpha^{l}) \circ \mathsf{zip.H.F} = \mathsf{zip.G.F} \circ (\alpha^{k})({}^{n}\mathsf{F}) \quad . \tag{49}$$

Note that the lemma does not imply that the zips are themselves simple. On the face of it, the property stated in the lemma is quite weak.

5.3 All regular datatypes commute

We now come to the main result of this paper, namely, that all regular relators commute. Morever, for each pair of regular relators F and G there is a *unique* natural transformation zip.F.G satisfying our requirements.

The regular relators are constructed from the constant relators, product and coproduct by pointwise extension and/or the construction of tree relators. The requirement that zip.F.G and zip.G.F be each other's converse (modulo transposition) demands the following definitions:

$$\mathsf{zip.Id.G} = \mathsf{idG} \tag{50}$$

$$\mathsf{zip.Proj.G} = \mathsf{idG}(\mathsf{Proj}^{k}) \quad \text{for all } \mathsf{G} : 1 \leftarrow \mathsf{k} \text{ and all } \mathsf{Proj} : 1 \leftarrow \mathsf{l} \tag{51}$$

$$\mathsf{zip.}\Delta\mathsf{F_k.G} = \tau\Delta(\mathsf{zip.F_k.G}) \tag{52}$$

$$\mathsf{zip.FG.H} = (\mathsf{zip.F.H})({}^{k}\mathsf{G}) \circ \mathsf{F}(\mathsf{zip.G.H}) \quad \text{for all } \mathsf{H} : 1 \leftarrow \mathsf{k} \tag{53}$$

The restriction to single-valued relators in these equations is made possible by the rule for $\mathsf{zip.G.}\Delta\mathsf{F_l}$.

For the constant relators and product and coproduct, the zip function is uniquely characterised by (48). One obtains the following definitions, for all $\mathsf{G} : 1 \leftarrow \mathsf{k}$:

$$\mathsf{zip.K_A.G} = (\mathsf{fan.G})(\mathsf{K_A}) \tag{54}$$

$$\mathsf{zip.+.G} = \mathsf{Ginl} \triangledown \mathsf{Ginr} \tag{55}$$

$$\mathsf{zip.}\times\mathsf{.G} = (\mathsf{Goutl} \vartriangle \mathsf{Goutr})^{\cup} \tag{56}$$

Note that, in general, $\mathsf{zip.K_A.G}$ and $\mathsf{zip.}\times\mathsf{.G}$ are not simple; moreover, the latter is typically partial. That is the right domain of $(\mathsf{zip.}\times\mathsf{.G})_{(\mathsf{A,B})}$ is typically a proper subset of $\mathsf{GA} \times \mathsf{GB}$. Datatypes defined in terms of these datatypes will thus also be non-simple and/or partial. Nevertheless, broadcast operations ("strengths") are always functional.

Tree relators are the last sort of relators in the class of regular relators. Let T be the tree relator induced by $\otimes$ as defined in section 2.2.6. Here the uniqueness of $\mathsf{zip.T.G}$ for all g is assured by (49) with α instantiated to in. One obtains:

$$\mathsf{zip.T.G} \;=\; (\!|\mathsf{id_G}\otimes\,;\;\mathsf{G}({}^{\mathsf{k}}\mathsf{in})\circ(\mathsf{zip}.\otimes.\mathsf{G})({}^{\mathsf{k}}(\mathrm{Id}\vartriangle\mathsf{T}))|\!) \quad \text{for all } \mathsf{G} : 1 \leftarrow \mathsf{k} \tag{57}$$

5.4 Broadcast and Structure Multiplication, Again

In our motivation of commuting datatypes, we said that the requirements for structure multiplication and "strength" would be met "almost by definition". In this section we observe in what sense that is indeed the case.

The requirements for structure multiplication are given by equations (30) and (31); those for broadcasts by (33), (34) and (35).

We begin with (30), (33) and (34). Note that all of these correspond to triangular diagrams. All are instances or simple consequences of the compositionality requirement of zips, 43(c). This is easiest to see in the case of (34) since it suffices to make the substitutions $\mathsf{F,G,H} := (\times\mathsf{C}),\mathsf{F},(\mathsf{A}\times)$. Next easiest to see is (33). Here the observation has to be made that $\mathsf{K_{FA}} = \mathsf{FK_A}$. Then make the substitutions $\mathsf{F,G,H} := (\times 1),\mathsf{F},\mathsf{K_A}$. Finally, (30) is a combination of 43(c) and (44) with the substitutions $\mathsf{F,G,H} := \mathsf{G},\mathsf{F},(\mathsf{A}\times)$. Thus all three requirements are satisfied, by definition, if it can be shown that all the relators involved belong to a class of commuting relators. In particular, since the sections $(\times\mathsf{C})$ and $(\mathsf{A}\times)$ are regular relators, all the requirements are met if in each case F is a regular relator.

The remaining two requirements, (31) and (35), are instances of (49) and 43(d), respectively. This is less easy to see. The key is to observe that the broadcast α where $\alpha_\mathsf{B} = (\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_\mathsf{A}$ is a proper, functional natural transformation of type $\mathsf{F}(\mathsf{A}\times) \leftarrow (\mathsf{FA})\times$ for each regular relator F and each A. (Note that the functionality is a special property of broadcasts. As mentioned before, zips are typically partial and nondeterministic. Hoogendijk [14] proves that $(\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_\mathsf{A}$ is functional for all regular relators F.) Property (31) is then an instance of (49) after making the substitutions $\mathsf{F,G,H} := \mathsf{G},\mathsf{F}(\mathsf{A}\times),(\mathsf{FA})\times$ and defining α as above. Property (35) is obtained from 43(d) using the substitutions $\mathsf{F,G,H},\alpha_\mathsf{B} := (\times\mathsf{C}),\mathsf{F}(\mathsf{A}\times),(\mathsf{FA})\times,(\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_\mathsf{A}$. This results in an inclusion —not an equality— but every term is a broadcast, and thus a function, and inclusion of functions is equivalent to their equality. We conclude that (31) and (35) are also met provided that F and all sections of the form $(\times\mathsf{C})$ and $(\mathsf{A}\times)$ are members of a class of commuting relators, and in particular if F is a regular relator.

6 Conclusion

Polytypism is a new concept in the repertoire of generic programming. In this paper we have made several innovatory contributions to the theoretical and practical development of polytypism. First, and arguably most importantly, we have provided strong evidence for the necessity of developing a theory of polytypism in a relational rather than a functional framework. Membership and fans can only be discussed at a metalevel in a functional framework and the fact that all regular relators commute is just not true in a functional framework since some of the transformations are necessarily nondeterministic. Second, we have demonstrated how to cope cleanly with non-endorelators thus overcoming a limitation of all other work in this field published to date that we know of (including our own). Third, we have illustrated a general approach to the specification of polytypic programs. Roughly summarised the approach is to require that the class of programs is compositional with respect to the pointwise definition of datatypes, and that the class is "higher order natural" in the sense that it maps related datatypes to related datatypes (just as polymorphic functions map related objects to related objects). This is a major advance on our earlier work [3] in which the commuting requirement was substantially more operational in flavour and hence *ad hoc*.

Several challenges remain. A major frustration is that we have been unable to establish a general unicity property of the "zip" operators even though in every individual case that we have studied we can prove unicity. This suggests that our requirements can be made stronger and, in the process, yet simpler and more elegant. Broader questions are how the notion of polytypism relates to, for example, design patterns [13] and adaptive object-oriented programming [19].

Acknowledgement The diagrams were drawn with the aid of Paul Taylor's commutative diagrams package.

References

[1] R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C.S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 303–326. Springer-Verlag, Workshops in Computing, 1992.

[2] R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, and J. van der Woude. Relational catamorphisms. In Möller B., editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers B.V., 1991.

[3] R.C. Backhouse, H. Doornbos, and P. Hoogendijk. Commuting relators. Available via World-Wide Web at `http://www.win.tue.nl/win/cs/wp/papers`, September 1992.

[4] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *J. of Functional Programming*, 6(1):1–28, January 1996.

[5] Richard S. Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall International, 1996.

[6] R.S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, pages 151–216. Springer-Verlag, 1989. NATO ASI Series, vol. F55.

[7] A. Carboni, G.M. Kelly, and R.J. Wood. A 2-categorical approach to geometric morphisms I. *Cahiers de Topologie et Geometrie Differentielle Categoriques*, 32(1):47–95, 1991.

[8] H. Doornbos. *Reductivity arguments and program construction.* PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, June 1996.

[9] Henk Doornbos and Roland Backhouse. Induction and recursion on datatypes. In B. Möller, editor, *Mathematics of Program Construction, 3rd International Conference*, volume 947 of *LNCS*, pages 242–256. Springer-Verlag, July 1995.

[10] Henk Doornbos and Roland Backhouse. Reductivity. *Science of Computer Programming*, 26(1–3):217–236, 1996.

[11] Achim Jung (Editor). Domains and denotational semantics: History, accomplishments and open problems. *Bulletin of the European Association for Computer Science*, 59:227–256, June 1996.

[12] P.J. Freyd and A. Scedrov. *Categories, Allegories.* North-Holland, 1990.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.

[14] Paul Hoogendijk. *A Generic Theory of Datatypes.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997.

[15] Paul Hoogendijk and Oege de Moor. What is a datatype? Technical Report 96/16, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1996. Submitted to Science of Computer Programming. Available via World-Wide Web at `http://www.win.tue.nl/win/cs/wp/papers`.

[16] J. Jeuring. Polytypic pattern matching. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995.

[17] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Proceedings of the Second International Summer School on Advanced Functional Programming Techniques*, pages 68–114. Springer-Verlag, 1996. LNCS 1129.

[18] Y. Kawahara. Notes on the universality of relational functors. *Memoirs of the Faculty of Science, Kyushu University, Series A, Mathematics*, 27(3):275–289, 1973.

[19] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Comm.A.C.M.*, 37(5):94–101, May 1994.

[20] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Conference on the Mathematics of Program Construction*, pages 335–347. Springer-Verlag LNCS 375, 1989.

[21] G. Malcolm. *Algebraic data types and program transformation.* PhD thesis, Groningen University, 1990.

[22] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, October 1990.

[23] Lambert Meertens. Calculate polytypically! In Herbert Kuchen and S. Doaitse Swierstra, editors, *Proceedings of the Eighth International Symposium PLILP '96 Programming Languages: Implementations, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 1996.

[24] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[25] R. Milner. A theory of type polymorphism in programming. *J. Comp. Syst. Scs.*, 17:348–375, 1977.

[26] R. Milner. The standard ML core language. *Polymorphism*, II(2), October 1985.

[27] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[28] Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, London, 1980.

[29] J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E. Mason, editor, *IFIP '83*, pages 513–523. Elsevier Science Publishers, 1983.

[30] J. Riguet. Relations binaires, fermetures, correspondances de Galois. *Bulletin de la Société Mathématique de France*, 76:114–155, 1948.

[31] Charles Simonyi. The death of computer languages, the birth of intentional programming. Proceedings of the 28th Annual International Seminar on the Teaching of Computing Science at University Level, Sponsored by ICL and University of Newcastle upon Tyne, Department of Computing Science, September 1995.

[32] C. Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967.