Generic Properties of Datatypes

Roland Backhouse*
Paul Hoogendijk†

August 28, 2002

Abstract

Generic programming adds a new dimension to the parametrisation of programs by allowing programs to be dependent on the structure of the data that they manipulate. Apart from the practical advantages of improved productivity that this offers, a major potential advantage is a substantial reduction in the burden of proof – by making programs more general we can also make them more robust and more reliable. These lectures discuss a theory of datatypes based on the algebra of relations which forms a basis for understanding datatype-generic programs. We review the notion of parametric polymorphism much exploited in conventional functional programming languages and show how it is extended to the higher-order notions of polymorphism relevant to generic programming.

*School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, England. rcb@cs.nott.ac.uk

†Philips Research, Prof. Holstlaan 4, 5655 AA Eindhoven, The Netherlands. Paul.Hoogendijk@philips.com

1 Introduction

Imagine that you bought a processor designed to perform floating-point calculations with the utmost efficiency but later you discovered that it would not work correctly for all input values. You would, of course, demand your money back — even if the manufacturer tried to claim that the occasions on which the processor would fail were very few and far between. Now imagine that you were offered some software that was described as "generic" but came with the caveat that it had only been tried on a few instances and the extent of its "genericity" could not be formulated precisely, let alone guaranteed. Would you buy it, or would you prefer to wait and let others act as the guinea pigs?

"Generic programming" is important to software designers as a way of improving programmer productivity. But "generic" programs can only have value if the full extent of their "genericity" can be described clearly and precisely, and if the programs themselves can be validated against their specifications throughout the full range of their applicability.

These lectures are about a novel parameterisation mechanism that allows programs to be dependent on the structure of the data that they manipulate, so-called "datatype genericity". The focus is on defining clearly and precisely what the word "generic" really means. The (potential) benefit is a substantial reduction in the burden of proof.

In the first lecture, we review the notion of "parametric polymorphism" in the context of functional programming languages. This is a form of genericity which has long since proved its worth. For us, the important fact is that the notion can be given a precise definition, thus allowing one to formally verify whether or not a given function complies with the definition.

The first lecture also introduces the notion of "commuting" two datatypes. This problem helps to illustrate the concepts introduced in the later lectures. We seek a non-operational specification based on fundamental building blocks for a generic theory of datatypes.

Defining parametric polymorphism involves extending the notion of "mapping" functions over a datatype to mapping *relations* over a datatype. (This is unavoidably the case even in the context of functional programming.) The second lecture, therefore, prepares the way for later lectures by introducing the fundamentals of a relational theory of datatypes. For brevity and effective, calculational reasoning we use "point-free" relation algebra. The most important concept introduced in this lecture is that of a "relator" — essentially a datatype. Other, more well-known concepts like weakest liberal precondition, are formulated in a point-free style.

The "membership" relation and the associated "fan" of a relator are introduced in the third lecture. We argue that a datatype is a relator with membership. The introduction of a membership relation allows us to give greater insight into what it means for a polymorphic function to be a natural transformation. It also helps to distinguish between "proper" natural transformations and "lax" natural transformation.

In the final lecture, we return to the issue of specifying when datatypes commute. This is formulated as the combination of two parametricity properties (one of which is high-level) and a homomorphism property (with respect to pointwise definition of relators).

2 Theorems for Free

In Pascal, it is necessary to define a different length function for lists of different types. For example, two different functions are needed to determine the length of a list of integers and to determine the length of a list of characters. This is a major inconvenience because, of course, the code for the two functions is identical; only their declarations are different. The justification given, at the time Pascal was designed, was that the increased redundancy facilitated type checking, thus assisting with the early detection of programming errors. In typed functional programming languages, beginning with ML [Mil77, Mil85], only one so-called "polymorphic" function is needed. Formally, the length function is a function with type $\mathbb{N} \leftarrow \mathsf{List}.\alpha$ for all instances of the type parameter α. That is, whatever the type A, there is an instance of the function length that maps a list of A's to a natural number.

A polymorphic function is said to be "parametric" if its behaviour does not depend on the type at which it is instantiated [Str67]. That the length function on lists is parametric can be expressed formally. Suppose a function is applied to each element of a list —the function may, for example, map each character to a number— . Clearly, this will have no effect on the length of the list. Formally[1], we have, for all types A and B and all functions f of type $A \leftarrow B$,

$$\mathit{length}_A \circ \mathsf{List}.f \;=\; \mathit{length}_B \quad .$$

Note that this equation does not characterise the length function. Indeed, postcomposing length with any function g from natural numbers to natural numbers, we get a function

[1]Function composition is denoted by an infix " $\circ$ " symbol. Function application is denoted by an infix " $.$ " symbol. By definition, $(f{\circ}g).x = f.(g.x)$. This is often called "backward composition" and " $\circ$ " is pronounced "after" (because f is executed "after" g). Because of this choice of order of evaluation, it is preferable to use a backward pointing arrow to indicate function types. The rule is that if f has type $A \leftarrow B$ and g has type $B \leftarrow C$ then $f{\circ}g$ has type $A \leftarrow C$. Functional programmers know the function " List " better by the name " `map` ". As we shall see, it is beneficial to give the type constructor and the associated " `map` " function the same name.

that satisfies an equation of the same form. For example, let sq denote the function that squares a number. Then

$$(sq \circ length_A) \circ \mathsf{List}.f \;=\; sq \circ length_B \quad .$$

Also precomposing the length function with any function that maps a list to list, possibly duplicating or omitting elements of the list, but not performing any computation dependent on the value of the list element, also satisfies an equation of the same form. For example, suppose $copycat$ is a (polymorphic) function that maps a list to the list concatenated with itself. (So, for example, $copycat.[0,1,2]$ equals $[0,1,2,0,1,2]$.) Then

$$(length_A \circ copycat_A) \circ \mathsf{List}.f \;=\; length_B \circ copycat_B \quad .$$

What is common to the functions $length$, $sq \circ length$ and $length \circ copycat$ is that they all have type $\mathbb{N} \leftarrow \mathsf{List}.\alpha$, for all instances of the type parameter α. Shortly, we define "parametric polymorphism" precisely. These three functions exemplify one instance of the definition. Specifically, g has so-called "polymorphic" type $\langle\forall\alpha :: \mathbb{N} \leftarrow \mathsf{List}.\alpha\rangle$ if for each type A there is a function g_A of type $\mathbb{N} \leftarrow \mathsf{List}.A$; g is so-called "parametrically" polymorphic if it has the property that, for all functions f of type $A \leftarrow B$,

$$g_A \circ \mathsf{List}.f \;=\; g_B \quad .$$

In general, a function will be called "parametrically polymorphic" if we can deduce a general equational property of the function from information about the type of the function alone.

A yet simpler example of a parametrically polymorphic function is the function fst that takes a pair of values and returns the first element of the pair. Using $\alpha \times \beta$ to denote the set of all pairs with first component of type α and second component of type β, the function fst has type $\langle\forall\, \alpha,\beta :: \alpha \leftarrow \alpha \times \beta\rangle$. Now suppose f and g are functions of arbitrary type, and suppose $f \times g$ denotes the function that maps a pair of values (x, y) (of appropriate type) to the pair $(f.x\,,\, g.y)$. Then, clearly,

$$\mathsf{fst} \circ f \times g \;=\; f \circ \mathsf{fst} \quad .$$

(Strictly, we should add type information by way of subscripts on the two occurrences of fst, just as we did for $length$. Later this type information will prove important, but for now we omit it from time to time.)

The simplest example of a parametrically polymorphic function is the identity function. For each type A there is an identity function id_A on A (the function such that, for all x of type A, $\mathsf{id}_A.x = x$.) Clearly, for all functions f of type $A \leftarrow B$,

$$\mathsf{id}_A \circ f \;=\; f \circ \mathsf{id}_B \quad .$$

What these examples illustrate is that a function is parametrically polymorphic if the function enjoys an algebraic property in common with all functions having the same type.

The key to a formal definition of "parametric polymorphism" is, in Wadler's words, "that types may be read as relations". This involves extending each type constructor —a function from types to types— to a relation constructor —a function from relations to relations— . The relations may, in fact, have arbitrary arity but we show how the extension is done just for binary relations.

For concreteness, we consider a language of types that comprises only function types and cartesian product. (In later lectures we extend the language of types to include dataypes like List .) Specifically, we consider the type expressions defined by the following grammar:

$$\mathit{Exp} \quad ::= \quad \mathit{Exp} \times \mathit{Exp} \;\mid\; \mathit{Exp} \leftarrow \mathit{Exp} \;\mid\; \mathit{Const} \;\mid\; \mathit{Var} \;\; .$$

Here, Const denotes a set of constant types, like $\mathbb{N}$ (the natural numbers) and $\mathbb{Z}$ (the integers). Var denotes a set of type variables. We use Greek letters to denote type variables.

The syntactic ambiguity in the grammar is resolved by assuming that $\times$ has precedence over $\leftarrow$, and both are left associative. For example, $\mathbb{N} \leftarrow \mathbb{N} \leftarrow \alpha \times \alpha \times \beta$ means $(\mathbb{N} \leftarrow \mathbb{N}) \leftarrow ((\alpha \times \alpha) \times \beta)$.

The inclusion of variables means that type expressions denote functions from types to types; we get a type for each instantiation of the type variables to types. The notation $T.A$ will be used to denote the type obtained by instantiating the variables in type expression T by the types A . (Generally, A will be a vector of types, one for each type variable in T .)

Type expressions are extended to denote functions from relations to relations as follows. The constant type A is read as the identity relation id_A on A . The product $R{\times}S$ of binary relations R and S of types $A \sim B$ and $C \sim D$ is defined to be the relation of type $A{\times}C \sim B{\times}D$ defined by

$$(1) \qquad ((a,c)\,,(b,d)) \in R{\times}S \quad \equiv \quad (a,b) \in R \;\wedge\; (c,d) \in S \;\; .$$

Finally, the function space constructor, " $\leftarrow$ ", is read as a mapping from a pair of relations R and S of types $A \sim B$ and $C \sim D$, respectively, to a binary relation $R{\leftarrow}S$ on functions f and g of type $A{\leftarrow}C$ and $B{\leftarrow}D$, respectively. Formally, suppose R and S are binary relations of type $A \sim B$ and $C \sim D$, respectively. Then $R{\leftarrow}S$ is the binary relation of type $(A \leftarrow C) \sim (B \leftarrow D)$ defined by, for all functions $f \in A \leftarrow C$ and $g \in B \leftarrow D$,

$$(2) \qquad (f,g) \in R \leftarrow S \quad \equiv \quad \langle \forall c,d :: (f.c\,,\, g.d) \in R \Leftarrow (c,d) \in S \rangle \quad .$$

In words, f and g construct R-related values from S-related values.

As an example, suppose $(A, \sqsubseteq)$ and $(B, \preceq)$ are partially ordered sets. Then we can instantiate R to $\sqsubseteq$ and S to $\preceq$ getting a relation $\sqsubseteq \leftarrow \preceq$ between functions f and g of type $A \leftarrow B$. In particular, switching to the usual infix notation for membership of an ordering relation,

$$(f,f) \in \sqsubseteq \leftarrow \preceq \quad \equiv \quad \langle \forall u,v :: f.u \sqsubseteq f.v \Leftarrow u \preceq v \rangle \quad .$$

So $(f,f) \in \sqsubseteq \leftarrow \preceq$ is the statement that f is a monotonic function. In Wadler's words, f maps $\preceq$-related values to $\sqsubseteq$-related values.

In this way, a type expression is extended to a function from relations to relations. We use $T.R$ to denote the relation obtained by instantiating the type variables in type expression T by the (vector of) relations R. Note that the construction we have given has the property that if relations R have type $A \sim B$ then $T.R$ has type $T.A \sim T.B$.

Definition 3 (Parametric Polymorphism) A term t is said to have *polymorphic* type $\langle \forall \alpha :: T.\alpha \rangle$, where T is a type expression parameterised by type variables α, if t assigns to each type A a value t_A of type $T.A$. A term t of polymorphic type $\langle \forall \alpha :: T.\alpha \rangle$ is said to be *parametrically polymorphic* if, for each instantiation of relations R to type variables, $(t_A\,,\, t_B) \in T.R$, where R has type $A \sim B$.

□

We sometimes abbreviate "parametrically polymorphic" to "parametric".

Exercise 4 The function fork creates a pair of values by simply copying its input value. That is $\mathsf{fork}.x = (x,x)$.

What is the type of fork? Formulate the property that fork is parametrically polymorphic and verify that this is indeed the case.

□

Exercise 5 Function application has type $\langle \forall \alpha,\beta :: \alpha \leftarrow (\alpha \leftarrow \beta) \times \beta \rangle$. That is, for each pair of types A and B, each function f of type $A \leftarrow B$ and each $b \in B$, $f.b \in A$. Formulate the statement that function application is parametrically polymorphic and verify that this is indeed the case.

□

Exercise 6 Currying has type $\langle\forall\,\alpha,\beta,\gamma :: \alpha \leftarrow \beta \leftarrow \gamma \leftarrow (\alpha \leftarrow \beta{\times}\gamma)\rangle$. Specifically,

$$curry.f.x.y \;=\; f.(x,y) \quad .$$

(Function application is assumed to be left-associative.) Formulate the statement that currying is parametrically polymorphic and verify that this is indeed the case.

□

As we have defined them, the notions of polymorphic and parametrically polymorphic are distinct. It is common for programming languages to permit the definition of polymorphic functions without their being parametric. This is known as *ad hoc* polymorphism [Str67]. For example, a language may have a polymorphic binary operator that is called the "equality" operator. That is, there is a binary operator, denoted say by $==$, such that for any pair of terms s and t in the language, $s == t$ denotes a boolean value. However, if the operator is parametric, we deduce from its type — $\mathrm{Bool} \leftarrow \alpha{\times}\alpha$ — that for all relations R and all u, v, x and y,

$$(u == v) \;=\; (x == y) \quad \Leftarrow \quad (u,x) \in R \,\wedge\, (v,y) \in R \quad .$$

In particular, taking the relation R to be an arbitrary function f definable in the programming language,

$$(f.x == f.y) \;=\; (x == y) \quad .$$

In other words, if the "equality" operator is parametrically polymorphic and truly tests for equality, all functions in the language are injective — which would make the language a very restrictive language indeed. In practice, of course, "equality" operators provided in programming languages are neither parametric nor true implementations of equality on all types.

If a language is such that all polymorphic terms are parametrically polymorphic, then, by definition, if term t has type T, for each instantiation of type variables to relations R of type $A \sim B$, $(t_A\,,\,t_B) \in T.R$. Instances of the property, for all relations R, $(t_A\,,\,t_B) \in T.R$, are called *free theorems*. For numerous examples of free theorems, see Wadler's paper [Wad89].

2.1 Verifiable Genericity

Polymorphism is a feature of programming languages that has many forms. Overloading and other *ad hoc* forms of polymorphism are introduced for the programmer's convenience, but offer little or no support for reasoning about programs. Parametric polymorphism, on the other hand, entails a meaningful and useful relationship between the different instances of the polymorphic object. Also, because the notion has a precise (and simple) formal definition, parametric polymorphism is a *verifiable* form of genericity.

In the remaining lectures, we introduce the idea of parameterising programs by datatypes. The emphasis is on ensuring that the genericity that is introduced is clearly and precisely specified so that all claims made about its use can be formally verified.

3 Commuting Datatypes — Introduction

In this section, we introduce an example of "higher order" parametricity — parametrisation with respect to type constructors (functions from types to types, like List) rather than types (like Int). Our purpose is to demonstrate the usefulness of making this sort of abstraction, and paving the way for the theoretical development that is to follow.

We use the term "datatype" without definition for the moment. (Later it will emerge that a datatype is a "relator with membership".) Think, however, of functions from type to types like pairing, and the formation of lists or arbitrary tree structures.

The section contains a number of examples of "commuting" two datatypes. This suggests a generic property of datatypes, namely that any two datatypes can be "commuted". We argue that one should define the notion of "commuting" datatypes by a precise formulation of the properties we require of the operation (by abstraction from a suitable collection of examples). In this way, reasoning about specific programs that exploit the generic property is substantially easier.

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

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

The function that performs this operation is known as the "zip" function to functional programmers. Zip commutes a pair of lists (of the same length) 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

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

— . 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, a broadcast is a parametrically polymorphic function of type $\langle\forall\alpha :: \mathsf{List}.(A\times\alpha) \leftarrow A \times \mathsf{List}.\alpha\rangle$; the two datatypes being "commuted" are thus $(A\times)$ and List. (Note that we use the Greek letter α and the Latin letter A in order to distinguish the roles of the two parameters.)

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. That is, using F to denote an arbitrary datatype and the infix operator "$\cdot$" to denote (backward) composition of datatypes, a broadcast is a parametrically polymorphic function of type $\langle\forall\alpha :: (F\cdot(A\times)).\alpha \leftarrow ((A\times)\cdot F).\alpha\rangle$.

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. (An F-structure is a value of type $F.X$ for some type X. So, a List-structure is just a list. Here, F could be, for example, List and G could be Tree.) 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 rather than a function. 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 two arguments whereas list formation has just one. (This is a complication that we will ignore in these lectures. See [HB97, Hoo97] for full details.)

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

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

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.

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

The point we want to make is that there is an obvious generalisation of this procedure: replace $\mathsf{List}.A$ by $F.A$ and $\mathsf{List}.B$ by $G.B$ for some arbitrary datatypes F and G. Doing so leads to the realisation that every step i nvolves commuting the order of a pair of datatypes.

This is made explicit in the diagram below where each arrow is labelled by two datatypes separated by the symbol "$\leftrightarrow$". For example, the middle, bottom arrow is labelled "$F \leftrightarrow G$". This indicates a transformation that "commutes" an "F-structure" of "G-structures" into a G-structure of F-structures. The rightmost arrow is labelled "$A\times \leftrightarrow G$". An $(A\times)$-structure is a pair of elements of which the first is of type A. Ignoring the "$F.$" for the moment, the transformation requires values of type $A \times G.B$ to be transformed into values of type $G.(A \times B)$. That is, an $(A\times)$-structure of G-structures has to be transformed into a G-structure of $(A\times)$-structures. Taking the "$F.$" into account simply means that the transformation has to take place at each place in the F-structure.

$$
\begin{tikzcd}
 & \mathsf{F.A} \times \mathsf{G.B} \arrow[dl, "((\mathsf{F.A})\times) \leftrightarrow \mathsf{G}"'] \arrow[dr, "(\times(\mathsf{G.B})) \leftrightarrow \mathsf{F}"] & \\
\mathsf{G.(F.A} \times \mathsf{B)} \arrow[d, "\times\mathsf{B} \leftrightarrow \mathsf{F}"'] & & \mathsf{F.(A} \times \mathsf{G.B)} \arrow[d, "\mathsf{A}\times \leftrightarrow \mathsf{G}"] \arrow[dll, "\mathsf{F}\cdot(\mathsf{A}\times) \leftrightarrow \mathsf{G}" description] \\
\mathsf{G.(F.(A} \times \mathsf{B))} & & \mathsf{F.(G.(A} \times \mathsf{B))} \arrow[ll, "\mathsf{F} \leftrightarrow \mathsf{G}"]
\end{tikzcd}
$$

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[2].

Filling out the requirement in more detail, let us suppose that, for datatypes F and G, $\mathsf{zip.F.G}$ is a family of relations indexed by types such that $(\mathsf{zip.F.G})_{\mathsf{A}}$ has type $(\mathsf{G}\cdot\mathsf{F}).\mathsf{A} \sim (\mathsf{F}\cdot\mathsf{G}).\mathsf{A}$. Then, assuming that, whenever $\mathsf{R}:\mathsf{U} \sim \mathsf{V}$, $\mathsf{F.R} : \mathsf{F.U} \sim \mathsf{F.V}$, the transformations that are used are as shown in the diagram below.

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

Now, in order to show that the whole diagram commutes (in the standard categorical sense of commuting diagram) it suffices to show that the two smaller diagrams commute. Specifically, the following two equalities must be established:

(7) $\quad (\mathsf{zip.(F\cdot(A\times)).G})_{\mathsf{B}} \ = \ (\mathsf{zip.F.G})_{\mathsf{A}\times\mathsf{B}} \circ \mathsf{F.(zip.(A\times).G)}_{\mathsf{B}}$

[2]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 $\mathsf{G.(F.A \times B)}$ and $\mathsf{F.(G.(A \times B))}$ but only one of these edges is needed in the argument that follows.

and

(8) $\quad (\mathsf{zip}.(\mathsf{F}\cdot(\mathsf{A}\times)).\mathsf{G})_{\mathsf{B}} \circ (\mathsf{zip}.(\times(\mathsf{G.B})).\mathsf{F})_{\mathsf{A}} \;=\; \mathsf{G}.(\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_{\mathsf{A}} \circ (\mathsf{zip}.((\mathsf{F.A})\times).\mathsf{G})_{\mathsf{B}}$.

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 Broadcasts

A family of functions bcst, where $\mathsf{bcst}_{\mathsf{A},\mathsf{B}} \;:\; \mathsf{F}.(\mathsf{A}\times\mathsf{B}) \leftarrow \mathsf{F.A}\times\mathsf{B}$, is said to be a *broadcast* for datatype F iff it is parametrically polymorphic in the parameters A and B and $\mathsf{bcst}_{\mathsf{A},\mathsf{B}}$ behaves coherently with respect to product in the following sense. First, the diagram

$$\begin{tikzcd}
\mathsf{F}.(\mathsf{A}\times\mathbb{1}) \arrow[dr, "(\mathsf{F}\cdot\mathsf{rid})_{\mathsf{A}}"'] & & \mathsf{F.A}\times\mathbb{1} \arrow[ll, "\mathsf{bcst}_{\mathsf{A},\mathbb{1}}"'] \arrow[dl, "(\mathsf{rid}\cdot\mathsf{F})_{\mathsf{A}}"] \\
& \mathsf{F.A} &
\end{tikzcd}$$

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

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

(where $\mathsf{ass}_{\mathsf{A},\mathsf{B},\mathsf{C}} \;:\; \mathsf{A}\times(\mathsf{B}\times\mathsf{C}) \leftarrow (\mathsf{A}\times\mathsf{B})\times\mathsf{C}$ is the obvious natural isomorphism) commutes as well.

The idea behind a "broadcast" is very simple. A broadcast for datatype F is a way of duplicating a given value of type B across every location in an F-structure of A's. The broadcasting operation is what Moggi [Mog91] calls the "strength" of a datatype.

The type of $\mathrm{bcst}_{A,B}$ of datatype F is the same as the type of $(\mathsf{zip}.(\times B).F)_A$, namely $F.(A \times B) \sim F.A \times B$. We give a generic specification of a zip such that, if F and the family of datatypes $(\times A)$ are included in a class of commuting datatypes, then any relation satisfying the requirements of $(\mathsf{zip}.(\times B).F)_A$ also satisfies the definition of $\mathrm{bcst}_{A,B}$ (and is a total function).

Let us begin with an informal scrutiny of the definition of broadcast. In the introduction to this section we remarked that a broadcast operation is an example of a zip. Specifically, a broadcast operation is a zip of the form $(\mathsf{zip}.(\times A).F)_B$. That is, a broadcast commutes the datatypes $(\times A)$ and F. Moreover, paying due attention to the fact that the datatype F is a parameter of the definition, we observe that all the transformations involved in the definition of a broadcast are themselves 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 datatype that is constantly A when applied to types. Thus rid_A is $(\mathsf{zip}.(\times \mathbb{1}).K_A)_B$ for some arbitrary B. In words, rid_A commutes the datatypes $(\times \mathbb{1})$ and K_A. Redrawing the first diagram above, using that all the arrows are broadcasts and thus zips, we get the following diagram[3].

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

Expressed as an equation, this is the requirement that

$$(9) \qquad \mathsf{zip}.(\times \mathbb{1}).(K_{F.A}) \;=\; F.(\mathsf{zip}.(\times \mathbb{1}).(K_A)) \circ ((\mathsf{zip}.(\times \mathbb{1}).F) \cdot K_A)$$

Now we turn to the second diagram in the definition of boradcasts. 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

[3] 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.

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 $\mathsf{F}.(A\times(\mathsf{B}\times\mathsf{C}))$ and $\mathsf{F}.(A\times\mathsf{B})\times\mathsf{C}$— are connected by the edge $(\mathsf{zip}.(\times\mathsf{C}).(\mathsf{F}.(A\times)))_{\mathsf{B}}$.

$$\begin{tikzcd}
\mathsf{F}.A\times(\mathsf{B}\times\mathsf{C}) \arrow[dd, "(\mathsf{zip}.(\times(\mathsf{B}\times\mathsf{C})).\mathsf{F})_A"'] & (\mathsf{F}.A\times\mathsf{B})\times\mathsf{C} \arrow[l, "(\mathsf{zip}.(\times\mathsf{C}).((\mathsf{F}.A)\times))_{\mathsf{B}}"'] \arrow[d, "(\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_A\times\mathsf{id}_{\mathsf{C}}"] \\
 & \mathsf{F}.(A\times\mathsf{B})\times\mathsf{C} \arrow[dl, "(\mathsf{zip}.(\times\mathsf{C}).(\mathsf{F}\cdot(A\times)))_{\mathsf{B}}"'] \arrow[d, "(\mathsf{zip}.(\times\mathsf{C}).\mathsf{F})_{A\times\mathsf{B}}"] \\
\mathsf{F}.(A\times(\mathsf{B}\times\mathsf{C})) & \mathsf{F}.((A\times\mathsf{B})\times\mathsf{C}) \arrow[l, "\mathsf{F}.(\mathsf{zip}.(\times\mathsf{C}).(A\times))_{\mathsf{B}}"]
\end{tikzcd}$$

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:

$$(10)\qquad (\mathsf{zip}.(\times\mathsf{C}).(\mathsf{F}\cdot(A\times)))_{\mathsf{B}} \;=\; \mathsf{F}.(\mathsf{zip}.(\times\mathsf{C}).(A\times))_{\mathsf{B}}\circ(\mathsf{zip}.(\times\mathsf{C}).\mathsf{F})_{A\times\mathsf{B}}\quad .$$

Second, the upper rectangle:

$$(11)\,(\mathsf{zip}.(\times(\mathsf{B}\times\mathsf{C})).\mathsf{F})_A\circ(\mathsf{zip}.(\times\mathsf{C}).((\mathsf{F}.A)\times))_{\mathsf{B}} = (\mathsf{zip}.(\times\mathsf{C}).(\mathsf{F}\cdot(A\times)))_{\mathsf{B}}\circ(\mathsf{zip}.(\times\mathsf{B}).\mathsf{F})_A\times\mathsf{id}_{\mathsf{C}}\ .$$

Note the strong similarity between (9) and (10). They are both instances of one equation parameterised by three different datatypes. There is also a similarity between these two equations and (7); 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 (8) and (11). As we shall see, however, both are instances of one equation parameterised again by three different datatypes except that (11) 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 Allegories and Relators

In this lecture, we begin the development of theory of datatypes in which the primitive mechanisms for defining programs are all parametrically polymorphic.

We use relation algebra (formally “allegory” theory [Fv90]) as the basis for our theory. There are many reasons why we choose relations as basis rather than functions. The most compelling reason is that, for us, programming is about constructing implementations of input-output relations. Program specifications are thus relations; but, also, programs themselves may be relations rather than functions — if one admits non-determinism, which we certainly wish to do.

Of more immediate relevance to this series of lectures is that, as we have seen, the formal definition of parametric polymorphism necessitates an extension of the definition of type constructors allowing them to map relations to relations. (The extension to relations is unavoidable because, even for functions f and g, $f \leftarrow g$ is a *relation* on functions and not a function from functions to functions. Also, the definition of commuting datatypes necessarily involves both nondeterminism and partiality — in particular, when one of the datatypes is the constant datatype K_A, for some type A.

In this section, we build up a language of terms defining relational specifications and implementations. The primitive relations in this language are not considered; our concern is with defining a number of constructs that build composite relations from given primitive relations and that preserve functionality and are parametrically polymorphic.

4.1 Allegories

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 id_A, and that every pair of arrows $R : A \leftarrow B$ and $S : B \leftarrow C$, with matching source and target[4], can be composed: $R \circ S : A \leftarrow C$. 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,

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

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

$$X \subseteq R \wedge X \subseteq S \quad \equiv \quad X \subseteq R \cap S \quad .$$

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

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

[4]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,

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

All three operators of an allegory are connected by the *modular law*, also known as Dedekind's law [Rig48]:

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

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".

(Note that we no longer use the symbol " $\sim$ " in addition to the symbol " $\leftarrow$ " as a means of distingushing relations from functions. From here on, relations are our prime interest, unless we specifically say otherwise.)

4.2 Relators

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

Definition 12 (Relator) A *relator* is a monotonic functor that commutes with converse. That is, let $\mathcal{A}$ and $\mathcal{B}$ be allegories. Then the mapping $F : \mathcal{A} \leftarrow \mathcal{B}$ is a relator iff,

(13) $\quad F.R : F.A \leftarrow F.B \quad \Leftarrow \quad R : A \leftarrow B \quad ,$

(14) $\quad F.R \circ F.S = F.(R \circ S) \quad$ for each $R : A \leftarrow B$ and $S : B \leftarrow C \quad ,$

(15) $\quad F.\mathsf{id}_A = \mathsf{id}_{F.A} \quad$ for each object $A \quad ,$

(16) $\quad F.R \subseteq F.S \quad \Leftarrow \quad R \subseteq S \quad$ for each $R : A \leftarrow B$ and $S : A \leftarrow B \quad ,$

(17) $\quad (F.R)^{\cup} = F.(R^{\cup}) \quad$ for each $R : A \leftarrow B \quad .$

□

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 [BBH+92] it was observed that all inductively-defined datatypes are relators.

A design requirement which led Backhouse to the above definition of a relator [BBH+92, BBM+91] 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 $R : A \leftarrow B$ is *total* iff

$$id_B \subseteq R^{\cup} \circ R \quad ,$$

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

$$R \circ R^{\cup} \subseteq id_A \quad .$$

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 $F : \mathcal{A} \leftarrow \mathcal{B}$ is a functor of type $Map(\mathcal{A}) \leftarrow Map(\mathcal{B})$. It is easily shown that our definition of relator guarantees this property. For instance, that relators preserve totality is proved as follows:

$$
\begin{array}{ll}
 & (F.R)^{\cup} \circ F.R \\
= & \{ \quad \text{converse and relators commute: (17)} \quad \} \\
 & F.(R^{\cup}) \circ F.R \\
= & \{ \quad \text{relators distribute through composition: (14)} \quad \} \\
 & F.(R^{\cup} \circ R) \\
\supseteq & \{ \quad \text{assume } id_B \subseteq R^{\cup} \circ R, \\
 & \quad\;\; \text{relators are monotonic: (16)} \quad \} \\
 & F.id_B \\
= & \{ \quad \text{relators preserve identities: (15)} \quad \} \\
 & id_{F.B} \quad .
\end{array}
$$

The proof that relators preserve simplicity is similar.

Note how this little calculation uses all four requirements for F to be a relator.

The following characterisation of functions proves to be very useful (for one reason because it is a Galois connection).

Definition 18 (Function) An arrow $f : A \leftarrow B$ is a *function* if, for all arrows, $R : C \leftarrow B$ and $S : A \leftarrow C$,

$$R \circ f^{\cup} \subseteq S \;\equiv\; R \subseteq S \circ f \quad .$$

□

Exercise 19 Show that definition 18 is equivalent to the conjunction of the two properties $f \circ f^{\cup} \subseteq id_A$ and $id_B \subseteq f^{\cup} \circ f$.

□

4.3 Composition and Relators are Parametric

As we go along, we introduce various building blocks for constructing specifications and programs. As these are introduced, we check whether they are parametric. Composition and relators are two such building blocks. So, in this section, we formulate and verify their parametricity properties.

This is, in fact, very straightforward. The advantage of using point-free relation algebra is that we can give a succinct definition of the arrow operator, leading to more succinct formulations of its basic properties and, hence, easier proofs.

There are several equivalent ways of defining the arrow operator, all of which are connected by definition 18. The one obtained directly from the pointwise definition is:

(20) $\quad (f,g) \in R{\leftarrow}S \;\equiv\; f^{\cup} \circ R \circ g \supseteq S \;.$

Another form, which we sometimes use, is:

(21) $\quad (f,g) \in R{\leftarrow}S \;\equiv\; R{\circ}g \supseteq f{\circ}S \;.$

Now, a relator F has type $\langle \forall \alpha,\beta :: (F.\alpha \leftarrow F.\beta) \leftarrow (\alpha \leftarrow \beta) \rangle$. (The two occurrences "$F$" in "$F.\alpha$" and "$F.\beta$" are maps from objects to objects, the first occurrence is a map from arrows to arrows.) Parametricity thus means that., for all relations R and S, and all functions f and g,

$$(F.f\,,\,F.g) \in F.R \leftarrow F.S \quad \Leftarrow \quad (f,g) \in R{\leftarrow}S \;.$$

Here we have used the pointwise definition (2) of the arrow operator. Now using the point-free definition (20), this reduces to:

$$(F.f)^{\cup} \circ F.R \circ F.g \supseteq F.S \quad \Leftarrow \quad f^{\cup} \circ R \circ g \supseteq S \;.$$

This we verify as follows:

$$
\begin{array}{ll}
 & (F.f)^{\cup} \circ F.R \circ F.g \\
= & \{ \quad \text{converse commutes with relators: (17)} \quad \} \\
 & F.(f^{\cup}) \circ F.R \circ F.g
\end{array}
$$

$=$ { relators distribute through composition: (14) }

$\quad F.(f^{\cup} \circ R \circ g)$

$\supseteq$ { assume $f^{\cup} \circ R \circ g \supseteq S$

relators are monotonic: (16) }

$\quad F.S$.

Note how this little calculation uses three of the requirements of F being a relator. So, by design, relators preserve functionality and are parametric.

Exercise 22 Formulate and verify the property that composition is parametrically polymorphic.

□

4.4 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 [Fv90] or [BdM96]. 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 $X : A \leftarrow B$,

$$\cup\mathcal{S} \subseteq X \equiv \forall(S \in \mathcal{S} :: \; S \subseteq X) \quad .$$

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 $1\!\!1$. We say that object $1\!\!1$ is a *unit* if $id_{1\!\!1}$ is the largest relation of its type and for every object A there exists a total relation $!_A : 1\!\!1 \leftarrow A$. 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 $T : A \leftarrow C$,

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

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

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

(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

$$(b,c) \in R\backslash T \;\equiv\; \forall(a:\; (a,b) \in R:\; (a,c) \in T) \;\;,$$

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

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

$$R = f \circ g^{\cup} \;\;\wedge\;\; f^{\cup} \circ f \cap g^{\cup} \circ g \;=\; id_C \;\;.$$

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 $outl : A \leftarrow C$ and $outr : B \leftarrow C$ is a tabulation of R.

If allegory $\mathcal{B}$ is tabular, a functor commutes with converse if it is monotonic. (Bird and De Moor [BdM96] claim an equivalence but their proof is flawed.) So, if we define a relator on a tabular allegory, one has to prove just requirement (16), from which (17) can be deduced. Property (17) is, however, central to many calculations and can usually be easily established without an appeal to tabularity. The same holds for property (14) which —in a tabulated allegory— is a consequence of (17) and the fact that relators are functors in the underlying category of maps: the property is also central to many calculations and can usually be easily established without an appeal to tabularity. For this reason we prefer to retain the definition that we originally proposed.

4.5 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:

(23) $\quad R \subseteq \top\!\!\top_{A,B} \circ X \;\equiv\; R_{>} \subseteq X \quad$ for each $X \subseteq id_B$.

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.

5 Datatype = Relator + Membership

This section defines a class of relators that we call the "regular" relators. This class contains a number of primitive relators —the constant relators, one for each type, product and coproduct— and a number of composite relators. We also introduce the notion of the "membership" relation. The regular relators correspond to the (non-nested) datatypes that one can define in a functional programming language. They all have an associated membership relation; this leads to the proposal, first made by De Moor and Hoogendijk [HdM00], that a datatype is a relator with membership.

There are two basic means for forming composite relators, namely induction and pointwise closure. We begin with a brief discussion of pointwise closure, since we need to say something about it, but do not go into it in depth in this series of lectures.

5.1 Pointwise Closure

For the purposes of this discussion, let us take for granted that List is an endorelator and product (denoted by an infix " $\times$ ") is a binary relator. By "endo", we mean that the source and target allegories of List are the same. By "binary", we mean that, for some allegory $\mathcal{A}$, product maps pairs of objects of $\mathcal{A}$ to an object of $\mathcal{A}$, and pairs of arrows of $\mathcal{A}$ to an arrow of $\mathcal{A}$.

From these two relators, we can form new relators by so-called "pointwise closure". One simple way is composition: defining $\mathsf{List}{\cdot}\mathsf{List}$ by $(\mathsf{List}{\cdot}\mathsf{List}).X = \mathsf{List}.(\mathsf{List}.X)$ (where X may be an arrow or an object), it is easily checked that $\mathsf{List}{\cdot}\mathsf{List}$ is a relator. Indeed, it is easy to verify that the functional composition of two relators $\mathsf{F} : \mathcal{A} \leftarrow \mathcal{B}$ and $\mathsf{G} : \mathcal{B} \leftarrow \mathcal{C}$, which we denote by $\mathsf{F}{\cdot}\mathsf{G}$, is a relator. There is also an identity relator for each allegory $\mathcal{A}$, which we denote by 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— .

Another way to form new relators is by tupling and projection. 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 $\mathsf{F}.R = R{\times}R$ 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 $R{\times}S{\times}S$ involve projection (of the pair (R, S) onto components R and S) as well as repetition (doubling), and product. 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.

In order to make all this precise, we need to 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.

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.*

To our collection of primitive relators, we need to add "duplicating" relators, Δ_k, of arity $k{*}m \leftarrow k$ (that simply duplicate their arguments m times) and projection relators, Proj_m, of arity $k \leftarrow k{*}m$ (that project vectors onto their components).

By using variables, tupling and projection are made invisible; functions defined on the primitive components are implicitly extended "pointwise" to composites formed by these relators.

Although the process of pointwise closure is seemingly straightforward, it is a non-trivial exercise to formulate it precisely, and the inclusion of arity information complicates the discussion substantially. For this reason, we restrict attention in these lectures almost exculsively to endorelators. Occasionally, we illustrate how tupling is handled using the case of a pair of relators. For this case, if F and G are both endorelators , we use $F\Delta G$ to denote the relator of arity $2 \leftarrow 1$ such that $(F\Delta G).R = (F.R\,,\,G.R)$. Complete accounts can be found in [Hoo97, HB97].

Definition 24 (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.
□

5.2 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 $\mathsf{K}_A.\mathsf{R} = \mathsf{id}_A$. Such relators are called *constant* relators.

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

(25) $\quad \mathsf{inl}^{\cup}_{A,B} \circ \mathsf{inl}_{A,B} = \mathsf{id}_A \quad \text{and} \quad \mathsf{inr}^{\cup}_{A,B} \circ \mathsf{inr}_{A,B} = \mathsf{id}_B \quad ,$

(26) $\quad \mathsf{inl}^{\cup}_{A,B} \circ \mathsf{inr}_{A,B} = \perp\!\!\!\perp_{A,B} \quad ,$

and

(27) $\quad (\mathsf{inl}_{A,B} \circ \mathsf{inl}^{\cup}_{A,B}) \cup (\mathsf{inr}_{A,B} \circ \mathsf{inr}^{\cup}_{A,B}) \;=\; \mathsf{id}_{A+B} \quad .$

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

(28) $\quad \mathsf{R} \triangledown \mathsf{S} \;=\; (\mathsf{R} \circ \mathsf{inl}^{\cup}_{A,B}) \cup (\mathsf{S} \circ \mathsf{inr}^{\cup}_{A,B}) \quad ,$

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

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

In the future, we adopt the convention that "$\triangledown$" and "$+$" have equal precedence, which is higher than the precedence of composition, which, in turn, is higher than the precedence of set union. We will also often omit type information (the subscripts attached to constants like inl).

Exercise 29 Prove the following properties of the junc operator:

$$\mathsf{R} \triangledown \mathsf{S} \subseteq \mathsf{X} \;\equiv\; \mathsf{R} \subseteq \mathsf{X} \circ \mathsf{inl} \wedge \mathsf{S} \subseteq \mathsf{X} \circ \mathsf{inr} \quad ,$$

the *computation* rules

$$\mathsf{R} \triangledown \mathsf{S} \circ \mathsf{inl} = \mathsf{R} \quad \text{, and}$$

$$\mathsf{R} \triangledown \mathsf{S} \circ \mathsf{inr} = \mathsf{S} \quad ,$$

and

$$\mathsf{X} \subseteq \mathsf{R} \triangledown \mathsf{S} \;\equiv\; \mathsf{X} \circ \mathsf{inl} \subseteq \mathsf{R} \wedge \mathsf{X} \circ \mathsf{inr} \subseteq \mathsf{S} \quad .$$

Hence conclude the *universal property*

$$\mathsf{X} = \mathsf{R} \triangledown \mathsf{S} \;\equiv\; \mathsf{X} \circ \mathsf{inl} = \mathsf{R} \wedge \mathsf{X} \circ \mathsf{inr} = \mathsf{S} \quad .$$

Prove, in addition, the *fusion* laws:

$$Q \circ R\triangledown S = (Q{\circ}R)\triangledown(Q{\circ}S) \quad ,$$

$$R\triangledown S \circ T{+}U = (R{\circ}T)\triangledown(S{\circ}U) \quad .$$

Formulate and verify the parametricity property of " $\triangledown$ " . Show also that " $+$ " is a relator.

□

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

$$(30) \qquad \mathsf{outl}_{A,B} \circ \mathsf{outr}_{A,B}^{\cup} = \top\!\!\top_{A,B} \quad ,$$

and

$$(31) \qquad \mathsf{outl}_{A,B}^{\cup} \circ \mathsf{outl}_{A,B} \cap \mathsf{outr}_{A,B}^{\cup} \circ \mathsf{outr}_{A,B} \;=\; \mathsf{id}_{A\times B} \quad .$$

Properties (30) and (31) are the same as saying that $A{\times}B$, $\mathsf{outl}_{A,B}$ and $\mathsf{outr}_{A,B}$, together, tabulate $\top\!\!\top_{A,B}$. Recalling that a unitary allegory is an allegory in which $\top\!\!\top_{A,B}$ exists for each pair of objects, A , B , it follows that products exist in a unitary, tabular allegory. (Note, however, that it is not necessary for all arrows to have a tabulation for products to exist.)

Having the projection functions outl and outr , we can define the *split* operator[5] on relations: for all $R : A \leftarrow C$ and $S : B \leftarrow C$

$$(32) \qquad R\vartriangle S \;=\; \mathsf{outl}_{A,B}^{\cup} \circ R \cap \mathsf{outr}_{A,B}^{\cup} \circ S \quad ,$$

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

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

Exercise 33 In category theory, coproduct and product are duals. Our definitions of product and coproduct are dual in the sense that compositions are reversed (as in category theory), set union is replaced by set intersection, and the empty relation, $\perp\!\!\perp$, is replaced by the universal relation, $\top\!\!\top$. The properties of the two are not completely

[5]The symbol " Δ " is conventionally used in category theory to denote the doubling functor. Its similarity with the symbol " $\vartriangle$ ", which we use to denote the split operator, is by design and not coincidence. Category theoreticians, however, use $\langle R,S\rangle$ instead of $R\vartriangle S$, thereby failing to highlight the correspondence. They also use $[R,S]$ where we use $R\triangledown S$, thus obscuring the duality.

dual, however, because composition distributes through set union but does not distribute through set intersection. This makes proofs about product and split more difficult than proofs about coproduct and junc. Nevertheless, crucial properties do dualise. In particular, product is a relator, and split is parametric.

This exercise is about proving these properties omitting the harder parts. For the full proofs see [BBH+92, BBM+91, Hoo97].

[Hard] Prove the *computation* rules

$$\mathsf{outl} \circ R \vartriangle S = R \circ S{>} \quad \text{, and}$$

$$\mathsf{outr} \circ R \vartriangle S = S \circ R{>} \quad .$$

(Hint: use the modular identity and the fact that outl is a function. You should also use the identity $R \circ S{>} = R \cap \top\!\!\top \circ S$.)

Assuming the *split-cosplit rule*

$$(R \vartriangle S)^{\cup} \circ T \vartriangle U = (R^{\cup} \circ T) \cap (S^{\cup} \circ U) \quad ,$$

prove the fusion laws

$$R \times S \circ T \vartriangle U = (R \circ T) \vartriangle (S \circ U) \quad \text{, and}$$

$$R \times S \circ T \times U = (R \circ T) \times (S \circ U) \quad .$$

Fill in the remaining elements of the proof that " $\times$ " is a relator. Finally, formulate and verify the parametricity property of " $\vartriangle$ " .

□

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

$$(34) \qquad X = (\![F ; R]\!) \quad \equiv \quad X \circ \mathsf{in} = R \circ F.X \quad .$$

Now, let $\otimes$ be a binary relator and assume that, for each A , $\mathsf{in}_A \; : \; T.A \leftarrow A \otimes T.A$ is an initial algebra of $(A\otimes)$[6]. Then the mapping T defined by, for all $R : A \leftarrow B$,

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

[6]Here and elsewhere we use the section notation $(A\otimes)$ for the relator $\otimes(K_A \,\Delta\, \mathrm{Id})$.

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

(Characterization (34) 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 [BdM96] refer to as the Eilenberg-Wright lemma.)

Exercise 35 The steps to show that T is a relator and catamorphisms are parametric are as follows.

1. Show that in is an isomorphism. The trick is to construct R and S so that $(\!|F;R|\!) = \mathsf{id}_A$ and $(\!|F;S|\!) \circ \mathsf{in} = (\!|F;R|\!)$. (For those familiar with initial algebras in category theory, this is a standard construction.)

2. Now observe that $(\!|F;R|\!)$ is the unique solution of the equation

$$X:: \quad X \;=\; R \circ F.X \circ \mathsf{in}^{\cup} \;\;.$$

By Knaster-Tarski, it is therefore the least solution of the equation

$$X:: \quad R \circ F.X \circ \mathsf{in}^{\cup} \subseteq X$$

and the greatest solution of the equation

$$X:: \quad X \subseteq R \circ F.X \circ \mathsf{in}^{\cup} \;\;.$$

(This relies on the assumption that the allegory is locally complete.) Use these two properties to derive (by mutual inclusion) the fusion property:

$$R \circ (\!|F;S|\!) \;=\; (\!|F;T|\!) \quad \Leftarrow \quad R \circ S \;=\; T \circ F.R \;\;.$$

3. Hence derive the fusion law

$$(\!|A\otimes\,;\,R|\!) \circ T.S \;=\; (\!|A\otimes\,;\; R \circ S \otimes \mathsf{id}|\!)$$

from which (and earlier properties) the fact that T is a relator and catamorphisms are parametric can be verified.

Complete this exercise.

□

5.3 Natural Transformations

Reynolds' characterisation of parametric polymorphism predicts that certain polymorphic functions are natural transformations. The point-free formulation of the definition of the arrow operator (21) helps to see this.

Consider, for example, the reverse function on lists, denoted here by rev. This has polymorphic type $\langle\forall\alpha :: \mathsf{List}.\alpha \leftarrow \mathsf{List}.\alpha\rangle$. So, it being parametric is the statement

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

for all relations $R : A \leftarrow B$. That is,

$$\mathsf{List}.R \circ \mathsf{rev}_B \ \supseteq\ \mathsf{rev}_A \circ \mathsf{List}.R$$

for all relations R. Similarly the function that makes a pair out of a single value, here denoted by fork, has type $\langle\forall\alpha :: \alpha\times\alpha \leftarrow \alpha\rangle$. So, it being parametric is the statement

$$R\times R \circ \mathsf{fork}_B \ \supseteq\ \mathsf{fork}_A \circ R$$

for all relations $R : A \leftarrow B$.

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 θ indexed by objects (equivalently, a mapping θ of objects to relations) to be *a natural transformation of type* $F \leftarrow G$, for relators F and G iff

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

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 $F \hookleftarrow G$ and $F \hookrightarrow G$, respectively. We define:

$$\theta : F \hookleftarrow G \quad = \quad F.R \circ \theta_B \supseteq \theta_A \circ G.R \quad \text{for each } R : A \leftarrow B$$

and

$$\theta : F \hookrightarrow G \quad = \quad F.R \circ \theta_B \subseteq \theta_A \circ G.R \quad \text{for each } R : A \leftarrow B \quad .$$

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 $\mathcal{M}ap$. The first lemma states that an allegorical natural transformation is a categorical natural transformation:

$$(F.f \circ \theta_B = \theta_A \circ G.f \quad \text{for each function } f : A \leftarrow B) \;\Leftarrow\; \theta : F \hookleftarrow G \;\; .$$

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:

$$\theta : F \hookleftarrow G \;\Leftarrow\; (F.f \circ \theta_B = \theta_A \circ G.f \quad \text{for each function } f : A \leftarrow B) \;\; .$$

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

$$\theta : F \hookleftarrow G \text{ in } \mathcal{A} \;\equiv\; \theta : F \leftarrow G \text{ in } \mathcal{M}ap(\mathcal{A})$$

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.

Exercise 36 Prove the above lemmas.

□

Since natural transformations of type $F \hookleftarrow G$ 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 θ is a *natural transformation* if $\theta : F \hookleftarrow G$ and we say that θ is a *proper* natural transformation if $\theta : F \leftarrow G$. (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 these lectures 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 $k \leftarrow l$ maps an l-tuple of objects to a k-tuple of relations. The governing rule is: if θ is a natural transformation to F from G (of whatever type — proper or not) then the arities of F and G and θ must be identical. Moreover, the composition $\theta \circ \kappa$ of two natural transformations (defined by $(\theta \circ \kappa)_A = \theta_A \circ \kappa_A$) is only valid if θ and κ have the same arity (since the composition is componentwise composition in the product allegory).

5.4 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 [HdM00]. Their argument, briefly summarised here, is based on the thesis that a datatype is a relator with a membership relation.

Suppose F is an endorelator. The interpretation of $F.R$ 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, $\mathsf{List}.R$ 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, $F.X$ 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 $\mathsf{mem}_A : A \leftarrow F.A$ and such that $F.X$ is the largest subset Y of $\mathsf{id}_{F.A}$ whose "members" are elements of the set X. Formally, mem is required to satisfy the property:

$$(37) \qquad \forall (X,Y\colon X \subseteq \mathsf{id}_A \wedge Y \subseteq \mathsf{id}_{F.A}\colon\ F.X \supseteq Y \ \equiv\ (\mathsf{mem}_A \circ Y)_< \subseteq X)$$

Note that (37) 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 [HdM00] 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 (37) 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 outline the case 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 $Y \subseteq \mathsf{id}_{A_0 \otimes A_1}$,

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

The rhs of (38) 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 relator: $\Delta_2 Y = (Y\,,Y)$. Now, writing $\mathsf{mem} = (\mathsf{mem}_0\,,\mathsf{mem}_1)$, $A = (A_0\,,A_1)$ and $X = (X_0\,,X_1)$, equation (38) becomes, for all partial identities $X \subseteq \mathsf{id}_A$ and $Y \subseteq \mathsf{id}_{(\otimes)A}$,

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

In fact, in [HdM00] (37) is not used as the defining property of membership. Instead the following definition is used, and it is shown that (37) is a consequence thereof. (As here, [HdM00] only considers the case $l = 1$.)

Arrow mem is a *membership* relation of endorelator F, if for object A

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

and for each pair of objects A and B and each $R : A \leftarrow B$,

(39) $$F.R \circ \mathsf{mem}_B \backslash \mathsf{id}_B = \mathsf{mem}_A \backslash R \quad .$$

Properties (39) and (37) are equivalent under the assumption of extensionality as shown by Hoogendijk [Hoo97].

Property (39) gives a great deal of insight into the nature of natural transformations. First, the property is easily generalised to:

(40) $$F.R \circ \mathsf{mem}_B \backslash S = \mathsf{mem}_A \backslash (R \circ S)$$

for all $R : A \leftarrow B$ and $S : B \leftarrow C$. Then, it is straightforward to show that the membership, mem, of relator $F : k \leftarrow l$ is a natural transformation. Indeed

(41) $\quad \mathsf{mem} : \mathrm{Id} \hookleftarrow \mathrm{F}$,

and also

(42) $\quad \mathsf{mem}\backslash\mathsf{id} : \mathrm{F} \hookleftarrow \mathrm{Id}$.

Having established these two properties, the —highly significant— observation that mem and mem\id are the *largest* natural transformations of their types can be made. Finally, and most significantly, suppose F and G are relators with memberships mem.F and mem.G respectively. Then the largest natural transformation of type $\mathrm{F} \hookleftarrow \mathrm{G}$ is mem.F\mem.G .

Theorem 43 A polymorphic relation mem is called the *membership* relation of relator F if it has type $\langle \forall \alpha :: \alpha \leftarrow \mathrm{F}.\alpha \rangle$ and satisfies (39), for all R. The relation mem\id is called the *fan* of F.

Assuming that id is the largest natural transformation of type $\mathrm{Id} \hookleftarrow \mathrm{Id}$, mem\id is the largest natural transformation of type $\mathrm{F} \hookleftarrow \mathrm{Id}$.

Suppose F and G are relators with memberships mem.F and mem.G respectively. Then the largest natural transformation of type $\mathrm{F} \hookleftarrow \mathrm{G}$ is mem.F\mem.G . (In particular, mem.F is the largest natural transformation of type $\mathrm{Id} \hookleftarrow \mathrm{F}$.)

Proof As mentioned, the proofs of (41) and (42) are left as exercises for the reader. (See below.)

Now,

$$
\begin{array}{ll}
 & \langle \forall A :: \theta_A \subseteq \mathsf{mem}_A \backslash \mathsf{id}_A \rangle \\
= & \quad \{ \quad \text{factors} \quad \} \\
 & \langle \forall A :: \mathsf{mem}_A \circ \theta_A \subseteq \mathsf{id}_A \rangle \\
\Leftarrow & \quad \{ \quad \text{identification axiom: id is the largest} \\
 & \quad \quad \text{natural transformation of type } \mathrm{Id} \hookleftarrow \mathrm{Id} \quad \} \\
 & \mathsf{mem} \circ \theta : \mathrm{Id} \hookleftarrow \mathrm{Id} \\
\Leftarrow & \quad \{ \quad \text{typing rule for natural transformations} \quad \} \\
 & \mathsf{mem} : \mathrm{Id} \hookleftarrow \mathrm{F} \wedge \theta : \mathrm{F} \hookleftarrow \mathrm{Id} \\
= & \quad \{ \quad (41) \quad \} \\
 & \theta : \mathrm{F} \hookleftarrow \mathrm{Id} \quad .
\end{array}
$$

We thus conclude that $\mathsf{mem}\backslash\mathsf{id}$ is the largest natural transformation of type $\mathsf{F} \hookleftarrow \mathsf{Id}$.

Now we show that mem is the largest natural transformation of its type. Suppose $\theta : \mathsf{Id} \hookleftarrow \mathsf{F}$. Then

$$
\begin{array}{ll}
 & \theta \\
\subseteq & \{ \quad \text{factors} \quad \} \\
 & \theta \circ \mathsf{mem}\backslash\mathsf{mem} \\
= & \{ \quad (39) \quad \} \\
 & \theta \circ \mathsf{F.mem} \circ \mathsf{mem}\backslash\mathsf{id} \\
\subseteq & \{ \quad \theta : \mathsf{Id} \hookleftarrow \mathsf{F} \quad \} \\
 & \mathsf{mem} \circ \theta \circ \mathsf{mem}\backslash\mathsf{id} \\
\subseteq & \{ \quad \theta : \mathsf{Id} \hookleftarrow \mathsf{F} \text{ and } \mathsf{mem}\backslash\mathsf{id} : \mathsf{F} \hookleftarrow \mathsf{Id}\,. \\
 & \quad\ \text{So } \theta \circ \mathsf{mem}\backslash\mathsf{id} : \mathsf{Id} \hookleftarrow \mathsf{Id}\,. \text{ Identification axiom.} \quad \} \\
 & \mathsf{mem} \quad .
\end{array}
$$

Finally, we show that $\mathsf{mem.F}\backslash\mathsf{mem.G}$ is the largest natural transformation of its type. Suppose $\theta : \mathsf{F} \hookleftarrow \mathsf{G}$. Then,

$$
\begin{array}{ll}
 & \mathsf{mem.F}\backslash\mathsf{mem.G} \supseteq \theta \\
= & \{ \quad \text{factors} \quad \} \\
 & \mathsf{mem.G} \supseteq \mathsf{mem.F} \circ \theta \\
\Leftarrow & \{ \quad \mathsf{mem.G} \text{ is the largest natural transformation of its type} \quad \} \\
 & \mathsf{mem.F} \circ \theta : \mathsf{Id} \hookleftarrow \mathsf{G} \\
= & \{ \quad \mathsf{mem.F} : \mathsf{Id} \hookleftarrow \mathsf{F} \text{ and } \theta : \mathsf{F} \hookleftarrow \mathsf{G} \\
 & \quad\ \text{composition of natrual transformations} \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

□

Exercise 44 Verify (41) and (42).

□

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 $\mathrm{Id} \hookleftarrow \mathrm{F}$ constructs values of type A given a structure of type $\mathrm{F}.A$. The fact that the membership relation for F is the largest natural transformation of type $\mathrm{Id} \hookleftarrow \mathrm{F}$ says that all values created by such a natural transformation must be members of the structure $\mathrm{F}.A$. Similarly, a natural transformation θ of type $\mathrm{F} \hookleftarrow \mathrm{G}$ constructs values of type $\mathrm{F}.A$ given a structure of type $\mathrm{G}.A$. The fact that $\mathsf{mem.F} \backslash \mathsf{mem.G}$ is the largest natural transformation of type $\mathrm{F} \hookleftarrow \mathrm{G}$ means that every member of the F-structure created by θ is a member of the input G-structure. A proper natural transformation $\theta : \mathrm{F} \leftarrow \mathrm{G}$ has types $\mathrm{F} \hookleftarrow \mathrm{G}$ and $\mathrm{F} \hookrightarrow \mathrm{G}$. So θ has type $\mathrm{F} \hookleftarrow \mathrm{G}$ and $\theta \cup$ has type $\mathrm{G} \hookleftarrow \mathrm{F}$. Consequently, a proper natural transformation copies values without loss or duplication.

The natural transformation $\mathsf{mem} \backslash \mathsf{id}$, the largest natural transformation of type $\mathrm{F} \hookleftarrow \mathrm{Id}$, 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".)

$$
\begin{array}{lrcl}
(45) & \mathsf{fan.Proj} & = & \mathsf{id} \\
(46) & \mathsf{fan.}(\mathrm{F}\Delta\mathrm{G}) & = & \mathsf{fan.F}\ \Delta\ \mathsf{fan.G} \\
(47) & \mathsf{fan.}(\mathrm{F}\cdot\mathrm{G}) & = & \mathrm{F}.(\mathsf{fan.G}) \circ \mathsf{fan.F} \\
(48) & \mathsf{fan.K}_A & = & \top\!\!\top_{A,\_} \\
(49) & \mathsf{fan.}+ & = & (\mathsf{id} \triangledown \mathsf{id})\cup \\
(50) & \mathsf{fan.}\times & = & \mathsf{id} \vartriangle \mathsf{id} \\
(51) & \mathsf{fan.T} & = & (\!|\, \mathsf{id}\otimes ; (\mathsf{fan.}\otimes)\cup \,|\!)\cup
\end{array}
$$

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

6 Commuting Datatypes — Formal Specification

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 $\mathsf{zip.F.G}$ for given datatypes F and G that maps $\mathrm{F}\cdot\mathrm{G}$-structures into $\mathrm{G}\cdot\mathrm{F}$-structures.

We consider only endo relators (relators of arity $1 \leftarrow 1$) here. For a full account, see [Hoo97, HB97].

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.

6.1 Naturality Requirements

Our first requirement is that $\mathsf{zip.F.G}$ be natural. That is to say, its application to an $\mathsf{F{\cdot}G}$-structure should not in any way depend on the values in that structure. Suppose that F and G are relators. Then we demand that

(52) $\quad \mathsf{zip.F.G} \; : \; \mathsf{G{\cdot}F} \leftarrow \mathsf{F{\cdot}G} \quad .$

Thus a zip is a *proper* natural transformation. It transforms one structure to another without loss or duplication of values.

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 $\theta : \mathsf{G} \hookleftarrow \mathsf{H}$ then $\mathsf{zip.F.G}$ and $\mathsf{zip.F.H}$ should be "coherent" with the transformation θ. That is to say, having both zips and θ, there are two ways of transforming $\mathsf{F{\cdot}H}$-structures into $\mathsf{G{\cdot}F}$-structures; these should effectively be the same.

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

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

$$\begin{tikzcd}
\mathsf{F{\cdot}G} \arrow[d, "\mathsf{zip.F.G}"'] & \mathsf{F{\cdot}H} \arrow[l, "\mathsf{F}{\cdot}\theta"'] \arrow[d, "\mathsf{zip.F.H}"] \\
\mathsf{G{\cdot}F} & \mathsf{H{\cdot}F} \arrow[l, "\theta{\cdot}\mathsf{F}"]
\end{tikzcd}$$

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

$$(53) \qquad (\theta{\cdot}\mathsf{F}) \circ \mathsf{zip.F.H} \;=\; \mathsf{zip.F.G} \circ (\mathsf{F}{\cdot}\theta)$$

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

The first reason is that if θ is not functional, i.e. θ is a non-deterministic transformation, the rhs of equation (53) may be more non-deterministic than the lhs because of the possible multiple occurrence of θ. 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 $\theta := \mathsf{id} \cup \mathsf{swap}$, i.e. θ non-deterministically swaps the elements of a pair or not. Then $(\theta{\cdot}\mathsf{F}) \circ \mathsf{zip.F.H}$ unzips a list of pairs into a pair of lists and swaps the lists or not. On the other hand, $\mathsf{zip.F.G} \circ (\mathsf{F}{\cdot}\theta)$ 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 (53) 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 $\mathsf{List.listify}$. 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 $(\mathsf{listify}\cdot\mathsf{List})\circ\mathsf{zip.List.Tree}$. 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 $\mathsf{List.(Tree.}A)$ is restricted to lists of trees of the same shape.

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

Note however that if we view both paths through the diagram as partial relations of type $\mathsf{List.(List.}A)\leftarrow\mathsf{List.(Tree.}A)$ then the upper path (via $\mathsf{List.(List.}A)$) includes the lower path (via $\mathsf{Tree.(List.}A)$). This is because the function $\mathsf{List.(listify}_A)$ 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 (53) using an inclusion instead of equality. Having this inclusion, the requirement for θ can be relaxed as well. So, the requirement becomes

(54) $\quad (\theta\cdot\mathsf{F})\circ\mathsf{zip.F.H}\subseteq\mathsf{zip.F.G}\circ(\mathsf{F}\cdot\theta)\quad$ for all $\theta : \mathsf{G}\hookleftarrow\mathsf{H}$.

6.2 Composition

For our final requirement we consider the monoid structure of relators under composition. Fix relator F and consider the collection of zips, $\mathsf{zip.F.G}$, indexed by (endo)relator G. Since the (endo)relators 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 $\mathsf{zip.F.(G{\cdot}H)}$ 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 $\mathsf{zip.F.(G{\cdot}H)}$,

$$\mathsf{zip.F.(G{\cdot}H)} \;:\; \mathsf{G{\cdot}H{\cdot}F} \leftarrow \mathsf{F{\cdot}G{\cdot}H} \quad ,$$

demands that the datatype F has to be "pushed" through G·H leaving the order of G and H unchanged. With zip.F.G we can swap the order of F and G, with zip.F.H the order of F and H. Thus transforming F·G·H to G·H·F can be achieved as shown below.

$$\mathsf{G{\cdot}H{\cdot}F} \xleftarrow{\mathsf{G \cdot zip.F.H}} \mathsf{G{\cdot}F{\cdot}H} \xleftarrow{\mathsf{(zip.F.G) \cdot H}} \mathsf{F{\cdot}G{\cdot}H}$$

So, we demand that

$$(55) \qquad \mathsf{zip.F.(G{\cdot}H)} \;=\; (\mathsf{G \cdot zip.F.H}) \circ (\mathsf{zip.F.G \cdot H}) \quad .$$

Moreover, in order to guarantee that $\mathsf{zip.F.(G{\cdot}Id)} = \mathsf{zip.F.G} = \mathsf{zip.F.(Id{\cdot}G)}$ we require that

$$(56) \qquad \mathsf{zip.F.Id} = \mathsf{id{\cdot}F} \quad .$$

.

6.3 Half Zips and Commuting Relators

Apart from the very first of our requirements ((52), the requirement that zip.F.G be natural), all the other requirements have been requirements on the nature of the mapping zip.F. Roughly speaking, (54) demands that it be parametric, and (55) and (56) that it be functorial. We find it useful to bundle the requirements together into the definition of something that we call a "half zip".

Definition 57 (Half Zip) Consider a fixed relator F and a pointwise closed class of relators $\mathcal{G}$. Then the members of the collection zip.F.G, where G ranges over $\mathcal{G}$, are called *half-zips* iff
(a) $\mathsf{zip.F.G} \;:\; \mathsf{G{\cdot}F} \leftarrow \mathsf{F{\cdot}G}$, for each G,
(b) $\mathsf{zip.F.(G{\cdot}H)} \;=\; (\mathsf{G \cdot zip.F.H}) \circ (\mathsf{zip.F.G \cdot H})$ for all G and H,
(c) $(\theta{\cdot}\mathsf{F}) \circ \mathsf{zip.F.H} \subseteq \mathsf{zip.F.G} \circ (\mathsf{F}{\cdot}\theta)$ for each $\theta : \mathsf{G} \hookleftarrow \mathsf{H}$.

(In addition, zips should respect the pointwise closure of the class $\mathcal{G}$, but this aspect of the definition is omitted here.)
□

Definition 58 (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

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

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

7 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.

7.1 Shape Preservation

Zips are partial operations: $\mathsf{zip.F.G}$ 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 $!_A$ is the function of type $\mathbb{1} \leftarrow A$ that replaces a value by the unique element of the unit type, $\mathbb{1}$. Also, for an arbitrary function f, $F.f$ 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 $F.!_A$ 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 $(F.!_A).x$ *is* the shape of the F–structure x, and $F.!_A \circ f$ *is* the shape of the result of applying function f.

Now, for a natural transformation θ of type $F \leftarrow G$, the shape characteristics of α in general are determined by $\theta_{\mathbb{1}}$, since

$$F.!_A \circ \theta_A = \theta_{\mathbb{1}} \circ G.!_A$$

That is, the shape of the result of applying θ_A is completely determined by the behaviour of $\theta_{\mathbb{1}}$. The shape characteristics of $\mathsf{zip.F.G}$, in particular, are determined by $(\mathsf{zip.F.G})_{\mathbb{1}}$ since

$$(G{\cdot}F).!_A \circ (\mathsf{zip.F.G})_A = (\mathsf{zip.F.G})_{\mathbb{1}} \circ (F{\cdot}G).!_A$$

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 $(\mathsf{zip.F.G})_{\mathbb{1}}$ to that of $\mathsf{fan.G}$. Specifically, we note that $(\mathsf{fan.G})_{\mathsf{F}.\mathbb{1}}$ 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, $\mathsf{F}.((\mathsf{fan.G})_{\mathbb{1}})$, 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

$$(59)\qquad (\mathsf{fan.G})_{\mathsf{F}.\mathbb{1}} \;=\; (\mathsf{zip.F.G})_{\mathbb{1}} \circ \mathsf{F}.((\mathsf{fan.G})_{\mathbb{1}}) \quad .$$

This property is an immediate consequence of the following lemma.

Suppose F and G are datatypes. Then, if $\mathsf{fan.G}$ is the canonical fan of G,

$$(60)\qquad \mathsf{fan.G} \cdot \mathsf{F} \;=\; \mathsf{zip.F.G} \circ (\mathsf{F} \cdot \mathsf{fan.G}) \quad .$$

From equation (59) it also follows that the range of $(\mathsf{zip.F.G})_{\mathbb{1}}$ is the range of $(\mathsf{fan.G})_{\mathsf{F}.\mathbb{1}}$, i.e. arbitrary G-structures of which all elements are the same, but arbitrary, F-shape.

A more general version of (60) is obtained by considering the so-called *fan function*. Recalling the characterising property of the membership relation (39), we define the mapping $\hat{\mathsf{F}}$ (with the same arity as F, namely $k \leftarrow l$) by

$$(61)\qquad \hat{\mathsf{F}}.R = \mathsf{F}.R \circ \mathsf{mem}\backslash\mathsf{id} \quad ,$$

for all $R : A \leftarrow B$. Then the generalisation of (60) is the following lemma.

Suppose F and G are datatypes. Then, if $\hat{\mathsf{G}}$ is the fan function of G,

$$(62)\qquad (\hat{\mathsf{G}} \cdot \mathsf{F}).R = (\mathsf{zip.F.G})_A \circ (\mathsf{F} \cdot \hat{\mathsf{G}}).R \quad ,$$

for all $R : A \leftarrow B$.

It is (62) that often uniquely characterises $\mathsf{zip.F.G}$. (In fact, our initial study of zips [BDH92] was where the notion of a fan was first introduced, and (62) was proposed as one of the defining properties of a zip.)

7.2 All regular datatypes commute

We conclude this discussion by showing that all regular relators commute. Morever, for each pair of regular relators F and G there is a *unique* natural transformation $\mathsf{zip.F.G}$ 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:

(63) $\quad \mathsf{zip.Id.G} \;=\; \mathsf{id{\cdot}G}$,

(64) $\quad \mathsf{zip.Proj.G} \;=\; \mathsf{id}$,

(65) $\quad \mathsf{zip.(F\,\Delta\,G).H} \;=\; (\mathsf{zip.F.H}\,,\,\mathsf{zip.G.H})$,

(66) $\quad \mathsf{zip.(F{\cdot}G).H} \;=\; (\mathsf{zip.F.H}\cdot\mathsf{G})\circ(\mathsf{F}\cdot\mathsf{zip.G.H})$.

For the constant relators and product and coproduct, the zip function is uniquely characterised by (62). One obtains the following definitions, for all G :

(67) $\quad \mathsf{zip.K_A.G} \;=\; \mathsf{fan.G}\cdot\mathsf{K_A}$,

(68) $\quad \mathsf{zip.{+}.G} \;=\; \mathsf{G.inl}\triangledown\mathsf{G.inr}$,

(69) $\quad \mathsf{zip.{\times}.G} \;=\; (\mathsf{G.outl}\vartriangle\mathsf{G.outr})^{\cup}$.

Note that, in general, $\mathsf{zip.K_A.G}$ and $\mathsf{zip.{\times}.G}$ are not simple; moreover, the latter is typically partial. That is the right domain of $(\mathsf{zip.{\times}.G})_{(A,B)}$ is typically a proper subset of $G.A \times G.B$. Zips of datatypes defined in terms of these datatypes will thus also be non-simple and/or partial. Nevertheless, broadcast operations 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 5.2. Then,

(70) $\quad \mathsf{zip.T.G} \;=\; (\!|\mathsf{id_G}\otimes\,;\; \mathsf{G.in}\circ(\mathsf{zip.}\otimes\mathsf{.G}\,\cdot\,(\mathrm{Id}\,\Delta\,\mathrm{T}))|\!)$.

8 Conclusion

The purpose of these lectures has been to introduce and study datatype-generic programs, emphasising genericity of specifications and proof. A relational theory of datatypes was introduced, motivated by the relational formulation of parametric polymorphism. Central to this theory is a (generic) formulation of the notion of a relator with membership. A fundamental insight, formally expressible within the theory, is that a natural transformation is a structural transformation that rearranges information (but does not create or alter information) possibly replicating or losing information in the process. A proper natural transformation rearranges information without loss or replication.

The problem of "commuting" two datatypes (transforming an F-structure of G-structures into a G-structure of F-structures) has been used to illustrate the effectiveness of a datatype-generic theory of programming. The specification of this problem is based on abstraction from the properties of brodacasting and matrix transposition, and is entirely non-operational. In this way, very general properties of the constructed programs are derived with a minimum of effort.

The development of a constructive theory of datatype genericity is a challenging area of research. Almost all literature on generic programming is about implementations with little or no regard to generic specification or proof. But finding the right abstractions to make the right building blocks demands that specification and proof be given as much consideration as implementation.

References

[BBH⁺92] 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.

[BBM⁺91] 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.

[BDH92] R.C. Backhouse, H. Doornbos, and P. Hoogendijk. Commuting relators. Available via World-Wide Web at `http://www.cs.nott.ac.uk/~rcb/MPC/papers`, September 1992.

[BdM96] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall International, 1996.

[Fv90] P.J. Freyd and A. Ščedrov. *Categories, Allegories*. North-Holland, 1990.

[HB97] Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In Eugenio Moggi and Giuseppe Rosolini, editors, *Category Theory and Computer Science, 7th International Conference*, volume 1290 of *LNCS*, pages 242–260. Springer-Verlag, September 1997.

[HdM00] Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

[Hoo97] Paul Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997.

[Mil77] R. Milner. A theory of type polymorphism in programming. *J. Comp. Syst. Scs.*, 17:348–375, 1977.

[Mil85] R. Milner. The standard ML core language. *Polymorphism*, II(2), October 1985.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Rig48] J. Riguet. Relations binaires, fermetures, correspondances de Galois. *Bulletin de la Société Mathématique de France*, 76:114–155, 1948.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967.

[Wad89] P. Wadler. Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture, ACM, London*, September 1989.

9 Solutions to Exercises

4 The type of fork is $\langle\forall\alpha :: \alpha\times\alpha\leftarrow\alpha\rangle$. It is parametric if, for all relations R ,

$$(\mathsf{fork},\mathsf{fork}) \in R\times R\leftarrow R \quad .$$

We determine that this is the case as follows:

$$
\begin{array}{ll}
 & (\mathsf{fork},\mathsf{fork}) \in R\times R\leftarrow R \\
= & \{ \quad \text{definition 2 of “}\leftarrow\text{” on relations} \quad \} \\
 & \langle\forall\, a{,}b :: (\mathsf{fork}.a\,,\,\mathsf{fork}.b) \in R\times R \;\Leftarrow\; (a,b)\in R\rangle \\
= & \{ \quad \text{definition of fork} \quad \} \\
 & \langle\forall\, a{,}b :: ((a,a)\,,(b,b)) \in R\times R \;\Leftarrow\; (a,b)\in R\rangle \\
= & \{ \quad \text{definition 1 of “}\times\text{” on relations} \quad \} \\
 & \langle\forall\, a{,}b :: (a,b)\in R\wedge(a,b)\in R \;\Leftarrow\; (a,b)\in R\rangle \\
= & \{ \quad \text{idempotency of } \wedge \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

□

5 Function application is parametrically polymorphic if for all types A , B , C and D , all relations R of type $A\sim B$ and S of type $C\sim D$, all functions f of type $A\leftarrow C$ and functions g of type $B\leftarrow D$ and all values c of type C and d of type D ,

$$(f.c\,,\,g.d)\in R \quad\Leftarrow\quad (f\,,\,g)\in R\leftarrow S \;\wedge\; (c\,,\,d)\in S \quad .$$

Now,

$$(f\,,\,g)\in R\leftarrow S \;\equiv\; \langle\forall c{,}d :: (f.c\,,\,g.d)\in R \Leftarrow (c\,,\,d)\in S\rangle \quad .$$

So, parametricity of function application is an application of modus ponens.
□

6 Currying is parametrically polymorphic if for all types A , B , C and D , E and F , all relations R of type $A\sim B$ and S of type $C\sim D$, and T of type $E\sim F$, all functions f of type $A\leftarrow C\times E$, g of type $B\leftarrow D\times F$ and all values c of type C , d of type D , e of type E and f of type F ,

$$
\begin{array}{ll}
 & (((\mathsf{curry}.f.c.d\,,\,\mathsf{curry}.g.e.f)\in R \Leftarrow (c\,,\,d)\in S) \;\Leftarrow\; (e\,,\,f)\in T) \\
\Leftarrow & \\
 & (f\,,\,g) \in R\leftarrow S\times T \quad .
\end{array}
$$

Now,

$$
\begin{array}{ll}
 & (f\,,\,g) \in R \leftarrow S{\times}T \\
\equiv & \\
 & \langle\forall\, c,d,e,f :: (f.(c,d)\,,\, g.(e,f)) \in R \;\Leftarrow\; (c\,,\,d) \in S \wedge (e\,,\,f) \in T\rangle \quad .
\end{array}
$$

So, parametricity of currying is immediate from its definition:

$$\mathit{curry}.f.c.d \;=\; f.(c,d) \;\;\wedge\;\; \mathit{curry}.g.e.f \;=\; g.(e,f) \quad .$$

□

19 The property $f \circ f^{\cup} \subseteq \mathit{id}_A$ is obtained from 18 by making the substitution f for R and id_A for S. Similarly, for $\mathit{id}_B \subseteq f^{\cup} \circ f$.

For the converse, we have:

$$
\begin{array}{ll}
 & R \circ f^{\cup} \subseteq S \\
\Rightarrow & \{ \quad \text{composition is monotonic} \quad \} \\
 & R \circ f^{\cup} \circ f \subseteq S \circ f \\
\Rightarrow & \{ \quad \mathit{id}_B \subseteq f^{\cup} \circ f,\ \text{transitivity of } \subseteq \quad \} \\
 & R \subseteq S \circ f \\
\Rightarrow & \{ \quad \text{composition is monotonic} \quad \} \\
 & R \circ f^{\cup} \subseteq S \circ f \circ f^{\cup} \\
\Rightarrow & \{ \quad f \circ f^{\cup} \subseteq \mathit{id}_A,\ \text{transitivity of } \subseteq \quad \} \\
 & R \circ f^{\cup} \subseteq S \quad .
\end{array}
$$

□

22 That composition is parametrically polymorphic is the statement

$$(f{\circ}g\,,\, h{\circ}k) \in R{\leftarrow}T \quad \Leftarrow \quad (f,h) \in R{\leftarrow}S \;\wedge\; (g,k) \in S{\leftarrow}T \quad .$$

Now,

$$
\begin{array}{ll}
 & (f{\circ}g\,,\, h{\circ}k) \in R{\leftarrow}T \\
= & \{ \quad \text{definition of the arrow operator} \quad \} \\
 & (f{\circ}g)^{\cup} \circ R \circ h \circ k \;\supseteq\; T \\
= & \{ \quad \text{converse over composition} \quad \} \\
 & g^{\cup} \circ f^{\cup} \circ R \circ h \circ k \;\supseteq\; T
\end{array}
$$

$$\Leftarrow \quad \{ \quad \text{transitivity of } \supseteq \quad \}$$

$$g^{\cup} \circ f^{\cup} \circ R \circ h \circ k \ \supseteq \ g^{\cup} \circ S \circ k \ \supseteq \ T$$

$$\Leftarrow \quad \{ \quad \text{monotonicity of composition} \quad \}$$

$$f^{\cup} \circ R \circ h \ \supseteq \ S \ \wedge \ g^{\cup} \circ S \circ k \ \supseteq \ T$$

$$= \quad \{ \quad \text{definition of the arrow operator} \quad \}$$

$$(f,h) \in R \leftarrow S \ \wedge \ (g,k) \in S \leftarrow T \ .$$

□

29 We have:

$$R \triangledown S \subseteq X$$

$$= \quad \{ \quad \text{definition: (28)} \quad \}$$

$$(R \circ \mathsf{inl}^{\cup}) \ \cup \ (S \circ \mathsf{inr}^{\cup}) \ \subseteq \ X$$

$$= \quad \{ \quad \text{set union} \quad \}$$

$$R \circ \mathsf{inl}^{\cup} \ \subseteq \ X \ \wedge \ S \circ \mathsf{inr}^{\cup} \ \subseteq \ X$$

$$= \quad \{ \quad \mathsf{inl} \text{ and } \mathsf{inr} \text{ are functions, definition 18} \quad \}$$

$$R \subseteq X \circ \mathsf{inl} \ \wedge \ S \subseteq X \circ \mathsf{inr} \ .$$

Also,

$$R \triangledown S \circ \mathsf{inl}$$

$$= \quad \{ \quad \text{definition: (28)} \quad \}$$

$$((R \circ \mathsf{inl}^{\cup}) \ \cup \ (S \circ \mathsf{inr}^{\cup})) \circ \mathsf{inl}$$

$$= \quad \{ \quad \text{composition distributes through union} \quad \}$$

$$R \circ \mathsf{inl}^{\cup} \circ \mathsf{inl} \ \cup \ S \circ \mathsf{inr}^{\cup} \circ \mathsf{inl}$$

$$= \quad \{ \quad \text{(25) and (26)} \quad \}$$

$$R \ .$$

Similarly for $R \triangledown S \circ \mathsf{inr} = S$. Now,

$$X \subseteq R \triangledown S$$

$$= \quad \{ \quad \text{(27)} \quad \}$$

$$X \circ ((\mathsf{inl} \circ \mathsf{inl}^{\cup}) \ \cup \ (\mathsf{inr} \circ \mathsf{inr}^{\cup})) \ \subseteq \ R \triangledown S$$

$$= \quad \{ \quad \text{composition distributes through union,}$$

$$
\begin{array}{ll}
 & \qquad\quad \text{definition: (28)} \quad \} \\
 & (X{\circ}\mathsf{inl}){\triangledown}(X{\circ}\mathsf{inr}) \;\subseteq\; R{\triangledown}S \\
= & \qquad \{ \quad \text{above} \quad \} \\
 & X{\circ}\mathsf{inl} \subseteq R{\triangledown}S \circ \mathsf{inl} \;\wedge\; X{\circ}\mathsf{inr} \subseteq R{\triangledown}S \circ \mathsf{inr} \\
= & \qquad \{ \quad \text{computation rules (proved above)} \quad \} \\
 & X{\circ}\mathsf{inl} \subseteq R \;\wedge\; X{\circ}\mathsf{inr} \subseteq S \quad .
\end{array}
$$

The universal property now follows from the antisymmetry of $\subseteq$.

We use the universal property to prove the fusion laws:

$$
\begin{array}{ll}
 & Q \circ R{\triangledown}S = (Q{\circ}R){\triangledown}(Q{\circ}S) \\
= & \qquad \{ \quad \text{universal property} \quad \} \\
 & \quad Q \circ R{\triangledown}S \circ \mathsf{inl} = (Q{\circ}R){\triangledown}(Q{\circ}S) \circ \mathsf{inl} \\
 & \wedge\;\; Q \circ R{\triangledown}S \circ \mathsf{inr} = (Q{\circ}R){\triangledown}(Q{\circ}S) \circ \mathsf{inr} \\
= & \qquad \{ \quad \text{computation rules} \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

The type of junc is

$$\langle \forall\, \alpha,\beta,\gamma :: (\alpha \leftarrow \beta{+}\gamma) \leftarrow ((\alpha \leftarrow \beta){\times}(\alpha \leftarrow y)) \rangle$$

parametricity of junc is the property that, for all f, g, h and k, and all R, S and T,

$$(f{\triangledown}g\,,\, h{\triangledown}k) \;\in\; R{\leftarrow}S{+}T \quad \Leftarrow \quad (f,h) \in R{\leftarrow}S \wedge (g,k) \in R{\leftarrow}T \quad .$$

We verify the property as follows:

$$
\begin{array}{ll}
 & (f{\triangledown}g\,,\, h{\triangledown}k) \;\in\; R{\leftarrow}S{+}T \\
= & \qquad \{ \quad \text{(20) and (18)} \quad \} \\
 & R \circ h{\triangledown}k \;\supseteq\; f{\triangledown}g \circ S{+}T \\
= & \qquad \{ \quad \text{fusion laws} \quad \} \\
 & (R{\circ}h){\triangledown}(R{\circ}k) \;\supseteq\; (f{\circ}S){\triangledown}(g{\circ}T) \\
= & \qquad \{ \quad \text{for all } X \text{ and } Y \\
 & \qquad\qquad\quad U{\triangledown}V \supseteq X{\triangledown}Y \\
 & \qquad\qquad = \qquad \{ \quad \text{above} \quad \} \\
 & \qquad\qquad\quad U{\triangledown}V \circ \mathsf{inl} \supseteq X \;\wedge\; U{\triangledown}V \circ \mathsf{inr} \supseteq Y
\end{array}
$$

$$= \quad \{ \quad \text{computation rule} \quad \}$$
$$U \supseteq X \wedge V \supseteq Y \quad \}$$
$$R \circ h \supseteq f \circ S \;\wedge\; R \circ k \supseteq g \circ T$$
$$= \quad \{ \quad (20) \text{ and } (18) \quad \}$$
$$(f, h) \in R \leftarrow S \wedge (g, k) \in R \leftarrow T \quad .$$

□

33 We have:

$$\mathsf{outl} \circ R \vartriangle S$$
$$= \quad \{ \quad \text{definition: } (32) \quad \}$$
$$\mathsf{outl} \circ (\mathsf{outl}^{\cup} \circ R \,\cap\, \mathsf{outr}^{\cup} \circ S)$$
$$\subseteq \quad \{ \quad \text{modular identity} \quad \}$$
$$\mathsf{outl} \circ \mathsf{outl}^{\cup} \circ (R \,\cap\, \mathsf{outl} \circ \mathsf{outr}^{\cup} \circ S)$$
$$\subseteq \quad \{ \quad \mathsf{outl} \text{ is simple, } (30) \quad \}$$
$$R \,\cap\, \mathsf{outl} \circ \mathsf{outr}^{\cup} \circ S$$
$$= \quad \{ \quad (30) \quad \}$$
$$R \,\cap\, \top\!\top \circ S$$
$$= \quad \{ \quad (30) \quad \}$$
$$R \,\cap\, \mathsf{outl} \circ \mathsf{outr}^{\cup} \circ S$$
$$\subseteq \quad \{ \quad \text{modular identity} \quad \}$$
$$\mathsf{outl} \circ (\mathsf{outl}^{\cup} \circ R \,\cap\, \mathsf{outr}^{\cup} \circ S)$$
$$= \quad \{ \quad \text{definition: } (32) \quad \}$$
$$\mathsf{outl} \circ R \vartriangle S \quad .$$

The proof of the fusion laws is straightforward (using the given property!) and omitted.

The type of split is

$$\langle \forall\, \alpha,\beta,\gamma :: (\beta \times \gamma \leftarrow \alpha) \leftarrow ((\beta \leftarrow \alpha) \times (y \leftarrow \alpha)) \rangle \quad .$$

Parametricity of split is the property that, for all f, g, h and k, and all R, S and T,

$$(f \vartriangle g \,,\, h \vartriangle k) \in S \times T \leftarrow R \quad \Leftarrow \quad (f, h) \in S \leftarrow R \wedge (g, k) \in T \leftarrow R \quad .$$

We verify the property as follows:

$$
\begin{array}{ll}
 & (f\vartriangle g\,,\,h\vartriangle k)\ \in\ S\times T \leftarrow R \\
= & \qquad \{ \qquad (21) \quad \} \\
 & S\times T \circ h\vartriangle k\ \supseteq\ f\vartriangle g \circ R \\
= & \qquad \{ \qquad \text{fusion laws} \quad \} \\
 & (S\circ h)\vartriangle(T\circ k)\ \supseteq\ (f\circ R)\vartriangle(g\circ R) \\
\Leftarrow & \qquad \{ \qquad \text{split is monotonic} \quad \} \\
 & S\circ h\ \supseteq\ f\circ R\ \wedge\ T\circ k\ \supseteq\ g\circ R \\
= & \qquad \{ \qquad (21) \quad \} \\
 & (f,h) \in S\leftarrow R \wedge (g,k) \in T\leftarrow R \quad .
\end{array}
$$

□

35 For part (1), we have:

$$
\begin{array}{ll}
 & (\!|F\,;R|\!) = \mathsf{id}_A \\
= & \qquad \{ \qquad \text{universal property} \quad \} \\
 & \mathsf{id}_A\circ\mathsf{in}\ =\ R\circ F.\mathsf{id}_A \\
= & \qquad \{ \qquad \text{relators preserve identities} \quad \} \\
 & \mathsf{in}\ =\ R \quad .
\end{array}
$$

So $(\!|F\,;\mathsf{in}|\!) = \mathsf{id}_A$. Now,

$$
\begin{array}{ll}
 & \mathsf{in}\circ(\!|F\,;S|\!)\ =\ \mathsf{id}_A \\
= & \qquad \{ \qquad \text{above} \quad \} \\
 & \mathsf{in}\circ(\!|F\,;S|\!)\ =\ (\!|F\,;\mathsf{in}|\!) \\
= & \qquad \{ \qquad \text{universal property} \quad \} \\
 & \mathsf{in}\circ(\!|F\,;S|\!)\circ\mathsf{in}\ =\ \mathsf{in}\circ F.(\mathsf{in}\circ(\!|F\,;S|\!)) \\
= & \qquad \{ \qquad \text{relators distribute through composition} \quad \} \\
 & \mathsf{in}\circ(\!|F\,;S|\!)\circ\mathsf{in}\ =\ \mathsf{in}\circ F.\mathsf{in}\circ F.(\!|F\,;S|\!) \\
= & \qquad \{ \qquad \text{computation rule (derived from universal property)} \quad \} \\
 & \mathsf{in}\circ S\circ F.(\!|F\,;S|\!)\ =\ \mathsf{in}\circ F.\mathsf{in}\circ F.(\!|F\,;S|\!) \\
\Leftarrow & \qquad \{ \qquad \text{Leibniz} \quad \} \\
 & S\ =\ F.\mathsf{in} \quad .
\end{array}
$$

So $\mathsf{in} \circ (\!|\mathsf{F}\,;\,\mathsf{F.in}|\!) = \mathsf{id}_\mathsf{A}$. It is now easy to verify that $(\!|\mathsf{F}\,;\,\mathsf{F.in}|\!) \circ \mathsf{in} = \mathsf{id}_{\mathsf{F.A}}$. Thus, $\mathsf{in}^\cup = (\!|\mathsf{F}\,;\,\mathsf{F.in}|\!)$.

For part (2), we have:

$$
\begin{array}{ll}
 & \mathsf{R} \circ (\!|\mathsf{F}\,;\mathsf{S}|\!) \;\subseteq\; (\!|\mathsf{F}\,;\mathsf{T}|\!) \\
\Leftarrow & \{ \quad (\!|\mathsf{F}\,;\mathsf{T}|\!) \text{ is the greatest solution of the equation} \\
 & \qquad \mathsf{X}{::}\quad \mathsf{X} \;\subseteq\; \mathsf{T} \circ \mathsf{F.X} \circ \mathsf{in}^\cup \quad \} \\
 & \mathsf{R} \circ (\!|\mathsf{F}\,;\mathsf{S}|\!) \;\subseteq\; \mathsf{T} \circ \mathsf{F.}(\mathsf{R} \circ (\!|\mathsf{F}\,;\mathsf{S}|\!)) \circ \mathsf{in}^\cup \\
= & \{ \quad \text{relators distribute through composition} \quad \} \\
 & \mathsf{R} \circ (\!|\mathsf{F}\,;\mathsf{S}|\!) \;\subseteq\; \mathsf{T} \circ \mathsf{F.R} \circ \mathsf{F.}(\!|\mathsf{F}\,;\mathsf{S}|\!) \circ \mathsf{in}^\cup \\
= & \{ \quad \mathsf{in} \text{ is a bijection} \quad \} \\
 & \mathsf{R} \circ (\!|\mathsf{F}\,;\mathsf{S}|\!) \circ \mathsf{in} \;\subseteq\; \mathsf{T} \circ \mathsf{F.R} \circ \mathsf{F.}(\!|\mathsf{F}\,;\mathsf{S}|\!) \\
= & \{ \quad \text{computation rule} \quad \} \\
 & \mathsf{R} \circ \mathsf{S} \circ \mathsf{F.}(\!|\mathsf{F}\,;\mathsf{S}|\!) \;\subseteq\; \mathsf{T} \circ \mathsf{F.R} \circ \mathsf{F.}(\!|\mathsf{F}\,;\mathsf{S}|\!) \\
\Leftarrow & \{ \quad \text{monotonicity of composition} \quad \} \\
 & \mathsf{R} \circ \mathsf{S} \;\subseteq\; \mathsf{T} \circ \mathsf{F.R} \quad .
\end{array}
$$

The reverse inequality is proved similarly. Combining the two

$$
\mathsf{R} \circ (\!|\mathsf{F}\,;\mathsf{S}|\!) \;=\; (\!|\mathsf{F}\,;\mathsf{T}|\!) \quad \Leftarrow \quad \mathsf{R} \circ \mathsf{S} = \mathsf{T} \circ \mathsf{F.R} \quad .
$$

Moving on to part (3),

$$
\begin{array}{ll}
 & (\!|\mathsf{A}\otimes\,;\,\mathsf{R}|\!) \circ \mathsf{T.S} \;=\; (\!|\mathsf{A}\otimes\,;\,W|\!) \\
= & \{ \quad \text{definition of } \mathsf{T.S} \quad \} \\
 & (\!|\mathsf{A}\otimes\,;\,\mathsf{R}|\!) \circ (\!|\mathsf{A}\otimes\,;\,\mathsf{in} \circ \mathsf{S}\otimes\mathsf{id}|\!) \;=\; (\!|\mathsf{A}\otimes\,;\,\mathsf{R} \circ \mathsf{id}\otimes\mathsf{S}|\!) \\
\Leftarrow & \{ \quad \text{above (with } \mathsf{F} := A\otimes\text{)} \quad \} \\
 & (\!|\mathsf{A}\otimes\,;\,\mathsf{R}|\!) \circ \mathsf{in} \circ \mathsf{S}\otimes\mathsf{id} \;=\; W \circ \mathsf{id}\otimes(\!|\mathsf{A}\otimes\,;\,\mathsf{R}|\!) \\
= & \{ \quad \text{computation rule} \quad \} \\
 & \mathsf{R} \circ \mathsf{id}\otimes(\!|\mathsf{A}\otimes\,;\,\mathsf{R}|\!) \circ \mathsf{S}\otimes\mathsf{id} \;=\; W \circ \mathsf{id}\otimes(\!|\mathsf{A}\otimes\,;\,\mathsf{R}|\!) \\
= & \{ \quad \text{relators distribute through composition} \quad \} \\
 & \mathsf{R} \circ \mathsf{S}\otimes\mathsf{id} \circ \mathsf{id}\otimes(\!|\mathsf{A}\otimes\,;\,\mathsf{R}|\!) \;=\; W \circ \mathsf{id}\otimes(\!|\mathsf{A}\otimes\,;\,\mathsf{R}|\!) \\
\Leftarrow & \{ \quad \text{Leibniz} \quad \} \\
 & \mathsf{R} \circ \mathsf{S}\otimes\mathsf{id} \;=\; W \quad .
\end{array}
$$

The type of a catamophism is $\langle\forall\beta :: (\beta \leftarrow A) \leftarrow (\beta \leftarrow F.\beta)\rangle$. So its, parametricity property is $\langle\forall R :: (R \leftarrow \mathrm{id}_A) \leftarrow (R \leftarrow F.R)\rangle$. That is, for all functions f and g,

$$(\!|F\,;f|\!) \circ \mathrm{id}_A \supseteq R \circ (\!|F\,;g|\!) \;\Leftarrow\; f \circ F.R \supseteq R \circ g \quad .$$

This is a special case of the property proved in part (2).
□

36 For the first lemma, we assume that $f : A \leftarrow B$ is a function. Then,

$$
\begin{array}{cl}
 & F.f \circ \theta = \theta \circ G.f \\
= & \{ \quad \text{set inclusion} \quad \} \\
 & F.f \circ \theta \subseteq \theta \circ G.f \;\wedge\; F.R \circ \theta \supseteq \theta \circ G.R \\
= & \{ \quad \text{assume } \theta : F \hookleftarrow G \quad \} \\
 & F.f \circ \theta \subseteq \theta \circ G.f \\
= & \{ \quad \text{relators respect functionality, (18)} \quad \} \\
 & \theta \circ (G.f)^{\cup} \subseteq (F.f)^{\cup} \circ \theta \\
= & \{ \quad \text{relators commute with converse} \quad \} \\
 & \theta \circ G.(f^{\cup}) \subseteq F.(f^{\cup}) \circ \theta \\
= & \{ \quad \text{assume } \theta : F \hookleftarrow G \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

For the second lemma, suppose (f, g) is a tabulation of R. Then,

$$
\begin{array}{cl}
 & F.R \circ \theta \supseteq \theta \circ G.R \\
= & \{ \quad R = f \circ g^{\cup}, \\
 & \quad\; \text{relators distribute through composition and converse} \quad \} \\
 & F.f \circ (F.g)^{\cup} \circ \theta \supseteq \theta \circ G.f \circ (G.g)^{\cup} \\
= & \{ \quad \text{assume } \theta : F \leftarrow G \text{, that is} \\
 & \quad\; \text{in particular, } F.f \circ \theta = \theta \circ G.f \quad \} \\
 & F.f \circ (F.g)^{\cup} \circ \theta \supseteq F.f \circ \theta \circ (G.g)^{\cup} \\
\Leftarrow & \{ \quad \text{monotonicity of composition} \quad \} \\
 & (F.g)^{\cup} \circ \theta \supseteq \theta \circ (G.g)^{\cup} \\
= & \{ \quad \text{relators respect functionality, (18)} \quad \} \\
 & \theta \circ G.g \supseteq F.g \circ \theta
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \quad \text{assume } \theta : F \leftarrow G\,, \\
 & \quad \text{in particular, } F.g \circ \theta = \theta \circ G.g \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

The converse ($\theta : F \hookleftarrow G$ in $\mathcal{A}$ implies $\theta : F \leftarrow G$ in $Map(\mathcal{A})$) follows because inclusion of functions is the same as equality. (That is, for functions f and g of the same type, $f \subseteq g \equiv f = g$.)

□

44 For (41), we have to prove that $R{\circ}\mathsf{mem} \supseteq \mathsf{mem} \circ F.R$ for all R. Now,

$$
\begin{array}{ll}
 & R{\circ}\mathsf{mem} \supseteq \mathsf{mem} \circ F.R \\
= & \{ \quad \text{factors} \quad \} \\
 & \mathsf{mem}\backslash(R{\circ}\mathsf{mem}) \supseteq F.R \\
= & \{ \quad (40) \quad \} \\
 & F.R \circ \mathsf{mem}\backslash\mathsf{mem} \supseteq F.R \\
\Leftarrow & \{ \quad \text{monotonicity of composition} \quad \} \\
 & \mathsf{mem}\backslash\mathsf{mem} \supseteq \mathsf{id} \\
= & \{ \quad \text{factors, reflexivity of } \supseteq \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

For (42) we have to prove that $F.R \circ \mathsf{mem}\backslash\mathsf{id} \supseteq \mathsf{mem}\backslash\mathsf{id} \circ R$ for all R. Now,

$$
\begin{array}{ll}
 & F.R \circ \mathsf{mem}\backslash\mathsf{id} \supseteq \mathsf{mem}\backslash\mathsf{id} \circ R \\
= & \{ \quad (40) \quad \} \\
 & \mathsf{mem}\backslash R \supseteq \mathsf{mem}\backslash\mathsf{id} \circ R \\
= & \{ \quad \text{factors} \quad \} \\
 & R \supseteq \mathsf{mem} \circ \mathsf{mem}\backslash\mathsf{id} \circ R \\
= & \{ \quad \text{cancellation of factors,} \\
 & \quad \text{monotonicity of composition} \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

□