EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computing Science

MASTER'S THESIS

Positions: A Contribution to the Development of Generic Programming

by

Sebastiaan van Erk

Supervisor: Prof. Dr. R. C. Backhouse

November 2, 2000

Contents

Preface

It has always been my philosophy that while studying at the university, I should follow as many theoretical courses as I could, while it was still possible, and that I would get enough chances to learn how to effectively put the theory into practice after graduation. This philosophy resulted in my decision to do my master's thesis in the area of programming methodology, because it was probably the most abstract and theoretical approach to computing science that I came in contact with during my study. It also intrigued me because of the many strange symbols that were used to write things down, and because I knew I would not be happy until I could use them too.

I started my graduation by reading a lot and coming to grips with the complex algebraic model of relational programming. Even though the basic mathematical structure is simple and consists of only a few rules, the amount of calculational rules that result is overwhelming. Futhermore, it has been my experience that even with the simplest concepts, I kept on finding new angles and new insights, and kept on developing a better understanding of the subject matter.

After reading many articles, reports, and books on generic datatypes and generic programs, it turned out that a concept of positions of datatypes could be very useful, and that it was not discussed anywhere in this formal setting. This is when I started thinking that finding a generic theory of positions could be an interesting topic of my master's thesis, since I really wanted to contribute something new myself.

I found that doing research was a slow and painful process, where a lot of work often only resulted in tiny steps forward, and even worse, often it would only point out the mistakes I was making. This would help me pick a new direction to go in, only to be confronted with the same situation again. In the end, I do have some positive results toward a theory of positions, but it is still incomplete and there is a lot of room for improvement.

To help me through this mathematical jungle, I wish to thank both my supervisor Roland Backhouse and Paul Hoogendijk, who helped me keep on track and to stay motivated when things did not quite go as smoothly as I would have liked. Their help was invaluable and the meetings we had gave me a lot of good ideas to work with. Furthermore, Paul's Ph. D. thesis has been a text book and a reference manual at the same time, teaching me most of what I know about generic programs and datatypes.

I would also like to thank my supervisor Roland Backhouse, Paul Hoogendijk, and Jaap van der Woude for taking place in the examining board.

Last but not least, I would like to thank my family and friends for all their support, especially my girlfriend Morena who has had to put up with a lot in these last months of work.

Chapter 1

Introduction

Computing science is a very new science, whose two primary concerns are the development of algorithms and the processing of information. Although certain specific algorithms were developed well before computers existed (such as *Euclid's Algorithm* for finding the greatest common factor of two numbers, which dates back to 300 BC), the general concept of an algorithm was not formalized until the 1930's. It was only in 1935-1936 that Alan Turing developed his general purpose computing machine (which was purely mathematical), and that the concept of an algorithm was identified with this Turing machine by the Church-Turing thesis.

Information processing is even newer. The concept of a relational database dates back to 1971 when E.F. Codd published two short papers in which they were introduced, and before 1971 databases were only part of computing science in an ad hoc fashion. With the recent development of Internet, databases have grown to many terrabytes in size, and information processing is playing an ever more important role.

The computing science industry is probably the only industry in which it is normal and legal to sell substandard products, plagued by bugs and failure, and where it is normal for customers to pay for bug fixes and patches. Software development is clearly still in a state of infancy and there is still a lot to learn. Although many formal methods exist to construct and verify programs, in practice these are often very difficult or infeasible to apply. It is a long cycle of interaction between theoretical research and applying the theory which will slowly take computing science to a higher level.

This thesis discusses a generic theory of programs and datatypes which uses the formal language of mathematics to allow the construction of programs which are provably correct, and as part of the theoretical research, we attempt to extend the theory of generic datatypes.

1.1 Generic Programs

A generic program is an algorithm which is applicable in many situations and can be used to solve many specific problems, instead of being restricted to one single problem. For example, a program which sorts a list of integers is less generic than a program which sorts a list of any type of values, given a linear order on those values. If one only has the first algorithm, every time one needs to sort a new kind of data, a new algorithm must be developed, whereas in the second case, one can repeatedly take the generic sort algorithm "off the shelf" and apply it directly to the new situation.

Generic programs are important for many reasons. As we mentioned, generic programs allow a library of standard algorithms to be created so one does not have to reinvent the wheel every time.

Furthermore, since the algorithms in a standard library have (hopefully) been derived and proven correct (or at least tested thoroughly), the chances of introducing error are much reduced.

Also very important of generic programs is that they unify what were previously thought of as different algorithms and show them to be different instances of a more generic algorithm. This form of abstraction eliminates unnecessary detail and allows a more concise and precise formulation of a problem. Using generic programs greatly reduces the burden of proof, for if the generic program is proved correct, then it is no longer necessary to show the correctness of any specific instance of the program.

1.2 Generic Datatypes

To write generic programs it is useful to have a generic characterization of what a datatype actually is. If we capture enough of the properties of an arbitrary datatype in a generic theory of datatypes, we can then derive programs which make use only of these properties, and therefore do not depend on any specific datatype. For example, a property that one would expect all datatypes to have is the ability to check if a data structure of that type contains a specific value. Thus we could check to see if the value 1 is contained in the list of integers [3,5,4] or the string "Tom" is contained in the set {"Tom","Dick","Harry"}.

Obviously, we do not want to have a too generic description of datatypes, leaving us very little to work with and making it impossible to derive many interesting results, but we also do not want it to be too specific, as this would limit the applicability of the theory.

1.3 A Relational Model

Since programs are in essence nothing more than relations which relate input values (possibly nondeterministically) to corresponding output values, we can model programs by relations. Furthermore, we can use a relational algebra which consists of a set of mathematical objects, operations and axioms, to calculate with programs. These calculations are then located in a formal setting and make it possible to derive and prove properties about programs in general.

To incorporate datatypes into the relational model it is only necessary to have typed relations, where the types simply represent datatypes. Then it is possible, also using an axiomatic approach, to require certain conditions to hold for these types, such as the ability to combine them in certain ways and to take them apart again.

Using typed relations to model programs and datatypes is a very powerful calculational tool with a strong mathematical foundation. It allows us to model both the specification and the final implementation in the same setting, and to use mathematical transformation rules to rewrite the specification into an implementation. If this transformation process can be automated in an interactive manner, it would allow us derive programs from specifications in such a way that the programs are provably correct, which is a powerful concept. Although there have been initial steps in this direction, such as for example the MAG [4] system, there is much work to be done.

1.4 Overview

This thesis consists of two parts. In the first part we will introduce the mathematical background and the theory of generic datatypes and generic programs. These chapters are based largely on the theory as presented by Paul Hoogendijk in his Ph.D. thesis [6], and partly on the books *Algebra of Programming* by Richard S. Bird and Oege de Moor [3] and *Mathematics of Program Construction* by Roland Backhouse [2]. In these chapters we attempt to present the theory in such a manner that no previous knowledge is required. Furthermore, we attempt to motivate and give an intuitive feel for the theory through the concrete and easy to understand model of binary relations over sets.

In chapter 2 we will first introduce a relational model for programs and datatypes called an allegory and we will present all the mathematical background that is necessary to read this thesis. Although we attempt to present the theory in such a fashion that it is not necessary to have any background in the subject matter, due to a lack of space, we will not be able to go into very much depth. For the final chapter it is probably useful to be familiar with the theory.

In chapter 3 we will introduce the theory of generic datatypes using the notions of functors and relators. After that, we will show how we can combine relators to model all the well-known datatypes in the allegorical setting, including the recursive datatypes.

In chapter 4 we will introduce the notion of a natural transformation which is a mathematical means of defining when a program is generic. Then we will introduce a generic membership relation as developed by Paul Hoogendijk in [6], which is an example of a natural transformation, and extends the notion of a generic datatype.

The second part of this thesis consists of the development of a generic theory of positions. Positions break down the membership relation of a generic datatype so that the information of where an element came from is not lost. Using positions we can decompose complex datatypes instead of only form them. Furthermore, we can split a data structure cleanly into its shape and contents in a unique way, so that given only the shape and contents we can reconstruct the original data structure. Positions also allow us to better characterize the notion of a generic program and the polymorphic map relation of a datatype, and in fact, it allows us to generalize this polymorphic map relation.

In chapter 5 we will introduce the concept of positions of datatypes, and investigate how these can be modelled in the allegorical setting. First we will analyse the requirements of positions, and attempt to find a suitable definition. After that we will find a collection of positions for all the well known datatypes, and suggest that a generic datatype can be defined as a relator with a collection of positions.

Chapter 2

Mathematical Framework

In this chapter we introduce the notions of categories and allegories, which are the algebraic structures we use to model functional and relational programming in an abstract and mathematically precise way. One advantage of categories and allegories is that we can calculate directly with functions and relations, instead of calculating with the effects functions and relations have on points. Stepping away from this extensional approach gives rise to point-free calculations which are short and elegant. Another advantage of categories and allegories is that we can abstract away from all the unnecessary details of specific models.

We will also introduce the other important categorical and allegorical concepts which we will need later on in this thesis, such as division, and ways to model sets using partial identities with the domain operators or the unit object with left and right conditions. Futhermore we will introduce product categories and allegories which allow calculations with vectors of relations.

2.1 Categories

Categories are a first step in the formalization of an algebraic model of functional and relational programming, and they are the most generic algebraic structures we will use. A category defines only a limited part of the structure we will need, but since the extra structure is only useful for a relational model, categories are all we have if we restrict ourselves to functions. However, many concepts can be introduced in the simple categorical setting first, after which they can be extended to a relational setting, and thus categories form a good basis for the relational model.

2.1.1 Definition

A category is an algebraic structure which, like every algebraic structure such as a group or a vector space, consists of a collection of mathematical entities, a collection of operations on these entities, and a collection of axioms which gives rise to the structure in the algebra. In a *category*, there are two kinds of mathematical entities, called *objects* and *arrows*. The *objects* in a category are denoted by the capital letters A, B, C, $\ldots$, and the *arrows* by the small letters f, g, h, $\ldots$.

The first pair of operations we discuss are the *target* and *source* operations, denoted by the postfix operators $\triangleleft$ and $\triangleright$ respectively. Both these operations assign a unique object to every arrow. If for an arrow f we have that $f\triangleleft = A$ and $f\triangleright = B$ we write $f : A \leftarrow B$ and say that f is of type $A \leftarrow B$, pronounced as "A from B".

The second pair of operations define the well known *monoid* structure. First we have an operation called *composition* which is a partial operation that maps a pair of arrows f and g with matching source and target, i.e. $f^{\triangleright} = g^{\triangleleft}$, to a single arrow $f \cdot g$, called the composition of f and g, such that $f \cdot g : f^{\triangleleft} \leftarrow g^{\triangleright}$. For composition we have an axiom, namely that it is *associative*. Associativity states that for all arrows $f : A{\leftarrow}B$, $g : B{\leftarrow}C$ and $h : C{\leftarrow}D$ we have that

$$(f \cdot g) \cdot h = f \cdot (g \cdot h)$$

The second operation of the monoid structure is the *identity* operation which maps each object A to the *identity* arrow $\mathsf{id}_A : A{\leftarrow}A$. For the identity arrow we have the axiom that it is both the left and right unit of composition, or formally, for every arrow $f : A{\leftarrow}B$ we have that

$$\mathsf{id}_A \cdot f = f = f \cdot \mathsf{id}_B$$

It is easy to verify that for every object there can exist at most one arrow which satisfies this axiom, and thus identity arrows are unique.

2.1.2 The Categories Map and Rel

We will now give two concrete instances of categories, namely the category Map of total functions between sets and the category Rel of binary relations between sets. While there are many other categories, these provide the motivation for using categories to model functional and relational programming in an abstract way.

In the category Map the objects are sets and the arrows are triples (f,A,B) such that A and B are sets and f is a total (not necessarily surjective) function from B to A. The composition of two arrows (f,A,B) and (g,B,C) is the arrow $(f \cdot g\,,A\,,C)$ where $f \cdot g$ is defined in the usual way on functions, i.e. $(f \cdot g).c := f.(g.c)$ for every element $c{\in}C$. The identity arrow on a set A is the triple (id_A,A,A) where id_A is the usual identity function on the set A, i.e. $\mathsf{id}_A.a := a$ for all elements $a{\in}A$.

In the category Rel the objects are also sets and the arrows are triples (R,A,B) such that A and B are sets and R is a subset of the Cartesian product $A{\times}B$. The set R represents a binary relation relating elements of the set B to elements of the set A. Whenever an ordered pair $(a,b){\in}R$ we say that b is related to a by R and we write $a\langle R\rangle b$. The composition of two arrows (R,A,B) and (S,B,C) is the arrow $(R \cdot S\,,A\,,C)$ where $R \cdot S$ is defined in the usual way, i.e.,

$$R \cdot S := \{(a,c) : \exists(b : b{\in}B : (a,b){\in}R \wedge (b,c){\in}S)\}$$

The identity arrow on a set A is the triple (id_A,A,A) where id_A is the usual identity relation on the set A, i.e. $\mathsf{id}_A := \{(a,a) : a{\in}A\}$.

To give a concrete example of the category Rel, consider a relation isSonOf where the target and source are the set of all people, and a relation isMarriedTo (with the same target and source). Given three people Tom, Harry and Jill, assume that Jill$\langle$isMarriedTo$\rangle$Tom and Tom$\langle$isSonOf$\rangle$Harry (note that these expressions must be read backwards, i.e., "Tom is married to Jill" and "Harry is a son of Tom.") Relations are not necessarily symmetric, i.e., we do not have that Harry$\langle$isSonOf$\rangle$Tom and thus order is important. Furthermore, there is no totality requirement, i.e., not everybody necessarily has a son. Similarly there is no functionality requirement, in other words, Harry may have more than one son. If we look at the interpretation of the composition of these two relations we get that $a\langle \mathsf{isMarriedTo} \cdot \mathsf{isSonOf}\rangle b$ if and only if there exists some person c such that $a\langle \mathsf{isMarriedTo}\rangle c$ and $c\langle \mathsf{isSonOf}\rangle b$, in other words $\mathsf{isMarriedTo} \cdot \mathsf{isSonOf}$ represents the relation "is the son of someone married to". Clearly Jill and Harry fit the bill.

2.2 Allegories

The category Rel has much more structure than the category Map, and much more structure than is captured by the axioms of a category. To capture some of this extra structure we define an allegory. Basically an allegory is a category in which arrows of the same type are ordered and form a complete lattice, and arrows are made reversible.

Before we define the structure of an allegory we will give a quick introduction to the theory of Galois connections, as they form an invaluable tool for creating short and calculationally powerful definitions.

2.2.1 Galois Connections

In this section we give a brief introduction of Galois connections as presented by Paul Hoogendijk in his Ph. D. thesis [6]. Since Galois connections are a very powerful and useful mathematical theory, we encourage the reader who is not familiar with them to read the master's thesis of C. J. Aarts [1], which discusses them in depth.

Given two partial orders (reflexive, anti-symmetric and transitive relations) $\sqsubseteq$ and $\preceq$ then a Galois connection is a pair of mappings f and g such that for all x and y the following equation holds,

$$f.x \sqsubseteq y \;\equiv\; x \preceq g.y$$

The mapping f is called the lower adjoint of the Galois connection, and the mapping g is called the upper adjoint. The power of the above equation is the ease with which it lends itself to calculations, and the number of properties which one can derive for f and g without knowing any specific details about these mappings. For example, by instantiating $f.x$ for y in the above equation, and similarly, by instantiating $g.y$ for x we get the following two cancellation properties,

$$x \preceq g.f.x$$

and

$$f.g.y \sqsubseteq y$$

Using these properties, it is easy to prove that both f and g are monotonic; for f we have,

$$\begin{array}{ll} & f.x_1 \sqsubseteq f.x_2 \\ \equiv & \quad \{\text{ shunting }\} \\ & x_1 \preceq g.f.x_2 \\ \Leftarrow & \quad \{\text{ cancellation, transitivity }\} \\ & x_1 \preceq x_2 \end{array}$$

The hint "shunting" is a direct application of the defining equation of Galois connections, which allows one to "shunt" a lower adjoint from the lower side of the first partial order to the corresponding upper adjoint on the upper side of the other partial order and vice versa. The proof that g is monotonic is practically identical.

An important fact about Galois connections is that adjoints are unique, that is, a specific lower adjoint can have at most one unique upper adjoint, and vice versa. For the first half of the proof, let g and h both be upper adjoints of f, then

$$
\begin{array}{ll}
 & x \preceq g.y \\
\equiv & \quad \{ \text{ shunting } \} \\
 & f.x \sqsubseteq y \\
\equiv & \quad \{ \text{ shunting } \} \\
 & x \preceq h.y
\end{array}
$$

Now by instantiating $h.y$ for x we get that $h.y \preceq g.y$ and by instantiating $g.y$ for x we get $g.y \preceq h.y$. Using anti-symmetry we have that $g.y = h.y$ for all y and thus by extensionality we have that $g{=}h$. The proof that an upper adjoint has at most one lower adjoint can be proved in exactly the same way, but it is more useful to realize that every upper adjoint becomes a lower adjoint when the orderings are reversed. That is, if f and g form a Galois connection, then so do g and f as follows,

$$g.y \succeq x \;\equiv\; y \sqsupseteq f.x$$

which is called the dual Galois connection. Whenever we have a statement about a lower adjoint we can simply reverse the orders and we get an equivalent dual statement about the upper adjoint for free.

One of the most important properties of Galois connections is that lower adjoints distribute over arbitrary unions and upper adjoints distribute over arbitrary intersections (assuming the relevant unions and intersections exist). We only prove the first statement, since the second one is the identical statement for the dual Galois connection. Let $\sqcup$ and $\cup$ denote the union operations of the partial orders $\sqsubseteq$ and $\preceq$ respectively, then for all collections of values X and all values y we have,

$$
\begin{array}{ll}
 & f \,.\, \sqcup X \;\preceq\; y \\
\equiv & \quad \{ \text{ shunting } \} \\
 & \sqcup X \sqsubseteq g.y \\
\equiv & \quad \{ \text{ union } \} \\
 & \forall(x :\; x{\in}X :\; x \sqsubseteq g.y) \\
\equiv & \quad \{ \text{ shunting } \} \\
 & \forall(x :\; x{\in}X :\; f.x \preceq y) \\
\equiv & \quad \{ \text{ union } \} \\
 & \cup(x :\; x{\in}X :\; f.x) \preceq y
\end{array}
$$

Using the anti-symmetry of $\preceq$ we get $f \,.\, \sqcup X = \cup(x :\; x{\in}X :\; f.x)$.

Since adjoints are unique, one can define a new operation in a unique way as an adjoint of an existing operation. Furthermore, we then immediately have the cancellation properties, monotonicity, and distribution over union or intersection. In combination with the shunting property of the definition we then have definition with many useful calculational rules for free.

2.2.2 Definition

The definition of an allegory given here is based on the definition given by Freyd and Ščedrov in [5] with the exception of the lattice structure, where we assume some extra structure based on the definition of the untyped relation algebra introduced in the book Mathematics of Program Construction by Roland Backhouse [2].

An allegory is a category with some extra structure. The arrows in an allegory will be called relations and denoted by the capital letters $R, S, T, \ldots$. We present an allegory as a combination of three distinct algebraic structures with special interface axioms which connect the different structures. Since an allegory is an extension of a category, the first structure is the category structure, with which we are already familiar.

The second structure of an allegory is a *partial order* $\subseteq$ on relations of the same type. Using the partial order we can define the infimum and supremum operations which we call *meet* and *join* and which we denote by the operators $\cap$ and $\cup$ respectively. The meet and join operations are defined by the following two (disguised) Galois connections; for all relations X, R, and S of the same type,

$$X \subseteq R{\cap}S \;\equiv\; X{\subseteq}R \wedge X{\subseteq}S$$

and

$$R{\cup}S \subseteq X \;\equiv\; R{\subseteq}X \wedge S{\subseteq}X$$

These definitions can be extended so that meet and join operate on arbitrary bags of relations; for all bags of relations $\mathcal{S}$ and relations X we require that,

$$X \subseteq \cap\mathcal{S} \;\equiv\; \forall(S : S{\in}\mathcal{S} : X{\subseteq}S)$$

and

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

We require relations of the same type to form a *complete lattice*, that is, for all bags of relations we require both the meet and the join to exist. In particular, if we instantiate $\mathcal{S}$ with the empty bag $\emptyset$ in the above equations we get that $\cup\emptyset \subseteq X$ and $X \subseteq \cap\emptyset$ for all relations X. Thus for every type $A{\leftarrow}B$ there exist a least and a greatest relation called bottom and top denoted by $\perp\!\!\!\perp_{A,B}$ and $\top\!\!\!\top_{A,B}$ respectively. If the subscripts are obvious from the context, they will be (silently) dropped. Finally, we require relations of the same type to form a *universally distributive lattice*, which simply means that meets distribute over arbitrary joins, i.e. for a relation R and a bag of relations $\mathcal{S}$,

$$R{\cap}(\cup\mathcal{S}) = \cup(S : S{\in}\mathcal{S} : R{\cap}S)$$

and that joins distribute over arbitrary meets, i.e. for a relation R and a bag of relations $\mathcal{S}$,

$$R{\cup}(\cap\mathcal{S}) = \cap(S : S{\in}\mathcal{S} : R{\cup}S)$$

Another way of formulating this is to require that $(R\cap)$ has an upper adjoint and that $(R\cup)$ has a lower adjoint, for all relations R.

The complete lattice structure is very well-documented in literature and one can derive an overwhelming amount of theory from this seemingly small basis. Simple properties of meet and join such as associativity, commutativity, idempotence $(X{\cap}X = X)$ and absorbtion $(X{\cup}(X{\cap}Y) = X)$ are easy to derive from the defining Galois connections and will be used in the rest of this thesis. The proofs of these properties are left to the reader.

Now that we have two different structures, namely the category structure and the complete, universally distributive lattice structure, we can define the interface between them. There are only two requirements which dictate how composition should behave in the lattice structure. First of all, composition is required to be monotonic in both its arguments, that is, for all relations R_1, R_2, S_1, and S_2 (of the correct types) we require

$$R_1 \cdot S_1 \subseteq R_2 \cdot S_2 \Leftarrow R_1 \subseteq R_2 \wedge S_1 \subseteq S_2$$

Secondly, composition is required to distribute over arbitrary joins from both the left and from the right, i.e., for all relations R and all bags of relations $\mathcal{S}$ we require that

$$R \cdot (\cup \mathcal{S}) = \cup(S : S \in \mathcal{S} : R \cdot S)$$

and

$$(\cup \mathcal{S}) \cdot R = \cup(S : S \in \mathcal{S} : S \cdot R)$$

(which can also be formulated as the requirement that $(R\cdot)$ has an upper adjoint and $(\cdot R)$ has a lower adjoint).

The third structure of an allegory is the reverse or converse structure. We define a reverse operation denoted by the postfix operator $^\circ$ which takes relations of type $A \leftarrow B$ to relations of type $B \leftarrow A$. The interface of the reverse structure with the category structure is as one expects, namely the reverse of a composition is the reversed composition of the reverses, i.e., for relations R and S we have,

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ$$

Furthermore, the reverse of the identity relation id is the identity relation itself, but this is not an axiom as we will be able to prove it.

The interface of the reverse structure with the lattice structure is given by the following Galois connection from which a multitude of properties can be derived; for all relations R and S,

$$R^\circ \subseteq S \equiv R \subseteq S^\circ$$

From this Galois connection we can deduce that reverse is a lattice isomorphism, that it is its own inverse, that it is monotonic, and that it distributes over arbitrary meets and joins which in turn implies that $\perp\!\!\perp^\circ = \perp\!\!\perp$ and $\top\!\!\top^\circ = \top\!\!\top$. Since we will make abundant use of these properties, the reader is encouraged to verify them.

Using the properties mentioned above, we can prove that $\mathsf{id}^\circ = \mathsf{id}$, although the proof is a bit tricky for such a simple result:

$$
\begin{array}{ll}
 & \mathsf{id}^\circ \\
= & \quad \{ \text{ identity } \} \\
 & \mathsf{id}^\circ \cdot \mathsf{id} \\
= & \quad \{ \text{ reverse is its own inverse } \} \\
 & \mathsf{id}^\circ \cdot (\mathsf{id}^\circ)^\circ \\
= & \quad \{ \text{ reverse of composition } \} \\
 & (\mathsf{id}^\circ \cdot \mathsf{id})^\circ \\
= & \quad \{ \text{ identity } \} \\
 & (\mathsf{id}^\circ)^\circ \\
= & \quad \{ \text{ reverse is its own inverse } \} \\
 & \mathsf{id}
\end{array}
$$

As a final interface axiom we have the so-called *modular law* which is a weak form of distributivity of composition over meets. Full distribution of composition over meet does not hold in general, not even in Rel, so we have to settle for the following weaker form; for all relations R, S, and T,

$$R \cdot S \cap T \subseteq R \cdot (S \cap R^{\circ} \cdot T)$$

Note that we give composition a higher precedence than the meet and join operations. Using the fact that reverse is a lattice isomorphism we can take the reverse of both sides of the inclusion in the above property to obtain the equivalent property (after some dummy renaming),

$$R \cap S \cdot T \subseteq (R \cdot T^{\circ} \cap S) \cdot T$$

This property is said to be the dual of the first property, and one can often use this dualization technique to get a second property for free.

2.2.3 The Allegory Rel

We defined allegories to capture the extra structure of Rel, and we will now show how all the operations of an allegory are defined in Rel. We will also give an equation which expresses when two points are related to each other, since we will use these equations to abstract well-known concepts of Rel to their more generic allegoric counterparts.

For the category structure we already gave the definition of the relation id and the relation $R \cdot S$. However, we will now give the equations which show the semantics of these operations. For the composition of two relations $R : A \leftarrow B$ and $S : B \leftarrow C$ we have that,

$$a\langle R \cdot S\rangle c \equiv \exists(b : b \in B : a\langle R\rangle b \wedge b\langle S\rangle c)$$

For the identity relation we have,

$$a_1\langle \mathsf{id}\rangle a_2 \equiv a_1 = a_2$$

These equations can be derived by straightforward calculations from the definitions of composition and the identity relation in Rel and the definition of $a\langle R\rangle b$.

For the lattice structure we first consider the partial order on relations on the same type. In Rel the inclusion relation $\subseteq$ is simply the subset relation $\subseteq$, and this motivates the choice of the notation. The semantics of the inclusion relation is as follows,

$$R \subseteq S \equiv \forall(a, b : a\langle R\rangle b : a\langle S\rangle b)$$

An intuitive example would be that $\mathsf{isSonOf} \subseteq \mathsf{isChildOf}$ since if Harry is a son of Tom, then certainly Harry is also a child of Tom.

Next we consider the meet and join operations, for which we just take the set intersection and set union operations $\cap$ and $\cup$ respectively, again motivating the notation we chose for the allegorical meet and join operations. The semantics of the meet and join are given by

$$a\langle R \cap S\rangle b \equiv a\langle R\rangle b \wedge a\langle S\rangle b$$

and

$$a\langle R \cup S\rangle b \equiv a\langle R\rangle b \vee a\langle S\rangle b$$

For the bottom and top relations $⫫_{A,B}$ and $\top\!\!\top_{A,B}$ we have that $⫫_{A,B} = \emptyset$ and $\top\!\!\top_{A,B} = A \times B$, where the semantics are given by $a\langle ⫫ \rangle b \equiv$ *false* and $a\langle \top\!\!\top \rangle b \equiv$ *true* for all a and b.

As a specific example of the meet operation, we have $a\langle \mathsf{isMarriedTo} \cap \mathsf{loveEachOther} \rangle b$ if and only if $a\langle \mathsf{isMarriedTo} \rangle b$ and $a\langle \mathsf{loveEachOther} \rangle b$ and thus if $\mathrm{Tom}\langle \mathsf{isMarriedTo} \cap \mathsf{loveEachOther} \rangle \mathrm{Jill}$ then they probably will not be getting a divorce in the near future. As an example of the join operation, if $a\langle \mathsf{isSonOf} \cup \mathsf{isDaughterOf} \rangle b$ then person b is a son of person a or person b is a daughter of person a, and thus we have $\mathsf{isSonOf} \cup \mathsf{isDaughterOf} = \mathsf{isChildOf}$. Note that in this case we also have that $\mathsf{isSonOf} \cap \mathsf{isDaughterOf} = ⫫$ since nobody can be someone's son and daughter at the same time. Finally, if we consider the relations isMale and $\mathsf{isFemale}$ which relate persons to the truth values *true* or *false*, then we clearly have that $\mathsf{isMale} \cup \mathsf{isFemale} = \top\!\!\top$, since every person is either male or female. Also note that these relations happen to be (total) functions, since they map every person to exactly one truth value.

Next we look at the reverse structure of an allegory. For a relation $R : A \leftarrow B$ the reverse is given by

$$R^\circ = \{(b,a) \ : \ (a,b) \in R\}$$

The semantics of the reverse are as follows,

$$a\langle R \rangle b \equiv b\langle R^\circ \rangle a$$

As an example, if we have the relation loves such that $a\langle \mathsf{loves} \rangle b$ if person b loves person a then the reverse loves° would be the relation $\mathsf{isLovedBy}$. Note that we can describe the $\mathsf{loveEachOther}$ relation in terms of the loves relation using the reverse structure, as $\mathsf{loveEachOther} = \mathsf{loves} \cap \mathsf{loves}^\circ$. In fact we have that $\mathsf{loveEachOther} = \mathsf{loveEachOther}^\circ$ and $\mathsf{loveEachOther}$ is a symmetric relation. Whenever $R = R^\circ$ we say that the relation R is symmetric.

2.3 The Link Between Categories and Allegories

If we restrict ourselves to functions in an allegory, then the significance of the partial order and the reverse structure disappear. In general, equality and inclusion of functions coincide, and functions are not closed under the meet, join and reverse operations. Furthermore, functions in an allegory are closed under composition and the identity relation is a function, so the functions of an allegory form a subcategory. In fact, an allegory is an extension of a category in much the same way that relations are an extension of functions. In this section we will motivate this link. First we must define the notion of a function in an allegory.

2.3.1 Functions

To define a function in an allegory we can use the defining predicate of a function in Rel to derive an equivalent point-free allegoric predicate. In Rel a function is a total and simple relation. Aiming to derive a similar definition in an allegory we will look at these two concepts seperately. A total relation in Rel maps every single point in its domain to at least one point in the range. Formally, a relation R is total if and only if

$$\forall(b :: \exists(a :: a\langle R \rangle b))$$

We will now use the definition of the allegorical operations in Rel to express totality in a point-free allegorical form:

$$\forall(b :: \exists(a :: a\langle R\rangle b))$$

$$\equiv \qquad \{ \text{ converse } \}$$

$$\forall(b :: \exists(a :: b\langle R^\circ\rangle a \wedge a\langle R\rangle b))$$

$$\equiv \qquad \{ \text{ composition } \}$$

$$\forall(b :: b\langle R^\circ \cdot R\rangle b)$$

$$\equiv \qquad \{ \text{ calculus, dummies } \}$$

$$\forall(b_1, b_2 : b_1{=}b_2 : b_1\langle R^\circ \cdot R\rangle b_2)$$

$$\equiv \qquad \{ \text{ identity } \}$$

$$\forall(b_1, b_2 : b_1\langle \mathsf{id}\rangle b_2 : b_1\langle R^\circ \cdot R\rangle b_2)$$

$$\equiv \qquad \{ \text{ inclusion } \}$$

$$\mathsf{id} \subseteq R^\circ \cdot R$$

Thus in Rel a relation R is total if and only if $\mathsf{id} \subseteq R^\circ \cdot R$, a predicate which is now expressed in terms of the lattice, composition, and reverse structures of an allegory. We will take this predicate to be the definition of a total relation in an allegory. Notice that we have found a point-free and quantification-free form which is much more compact than the original definition in Rel, and as we will see, lends itself very well to compact and elegant computations.

It is easy to verify, using this new allegorical definition of a total relation, that the composition of two total relations is again a total relation. We will give the proof as an example of the power of the allegorical definition. Let R and S be total relations (such that their composition exists, i.e. $R\triangleright = S\triangleleft$), then we wish to show that $\mathsf{id} \subseteq (R \cdot S)^\circ \cdot (R \cdot S)$. We start by taking the greater side of the inclusion and calculating as follows:

$$(R \cdot S)^\circ \cdot (R \cdot S)$$

$$= \qquad \{ \text{ converse of composition } \}$$

$$S^\circ \cdot R^\circ \cdot R \cdot S$$

$$\supseteq \qquad \{ \text{ R is total, monotonicity of composition } \}$$

$$S^\circ \cdot \mathsf{id} \cdot S$$

$$= \qquad \{ \text{ identity } \}$$

$$S^\circ \cdot S$$

$$\supseteq \qquad \{ \text{ S is total } \}$$

$$\mathsf{id}$$

Now we have the desired result. Note that normally we do not include trivial steps such as the step including only the identity relation, and similarly, we omit trivial hints such as the monotonicity of composition. Steps like this will be silently omitted.

As we did for totality above, we can take the definition of a simple relation expressed in a pointwise form in Rel, and rewrite it as an allegorical predicate. In Rel a relation is simple (or functional) if it maps every point in its domain to at most one point in its range. The standard way to express this for a relation R is as follows:

$$\forall(a_1, a_2, b : a_1\langle R\rangle b \wedge a_2\langle R\rangle b : a_1{=}a_2)$$

Again we use the definition of the allegorical operations in Rel to express simplicity in a point-free allegorical form:

$$
\begin{array}{ll}
 & \forall(a_1, a_2, b : a_1\langle R\rangle b \wedge a_2\langle R\rangle b : a_1{=}a_2) \\
\equiv & \qquad \{ \text{ converse, identity } \} \\
 & \forall(a_1, a_2, b : a_1\langle R\rangle b \wedge b\langle R^\circ\rangle a_2 : a_1\langle \mathsf{id}\rangle a_2) \\
\equiv & \qquad \{ \text{ calculus } \} \\
 & \forall(a_1, a_2 : \exists(b :: a_1\langle R\rangle b \wedge b\langle R^\circ\rangle a_2) : a_1\langle \mathsf{id}\rangle a_2) \\
\equiv & \qquad \{ \text{ composition } \} \\
 & \forall(a_1, a_2 : a_1\langle R \cdot R^\circ\rangle a_2 : a_1\langle \mathsf{id}\rangle a_2) \\
\equiv & \qquad \{ \text{ inclusion } \} \\
 & R \cdot R^\circ \subseteq \mathsf{id}
\end{array}
$$

Thus in an allegory we define a relation to be simple if and only if $R \cdot R^\circ \subseteq \mathsf{id}$. It is easy to show that the composition of two simple relations is again simple, and since the proof is practically identical to the above proof about totality, we leave it as an exercise for the reader.

It is interesting to notice that using these definitions it is also easy to define surjectivity and injectivity, since a relation R is surjective if and only if R° is total, and similarly, R is injective if and only if R° is simple. This also shows a problem with these definitions; as we now have four different predicates, namely $\mathsf{id} \subseteq R^\circ \cdot R$ (totality), $R \cdot R^\circ \subseteq \mathsf{id}$ (simplicity), $\mathsf{id} \subseteq R \cdot R^\circ$ (surjectivity) and $R^\circ \cdot R \subseteq \mathsf{id}$ (injectivity), it becomes easy to get them confused. One advantage of their longer pointwise counterparts is that they are easier to understand. However, with a little bit of practice, one quickly gets used to the allegorical forms.

Now that we have the appropriate allegorical definitions for totality and simplicity, we can define a function in an allegory in exactly the same way as we did in Rel, namely as a total and simple relation.

There is another way to define functions, which is entirely equivalent to the above definition, but which has some extra calculational benefits. This definition is in the form of an equivalence which says that a relation f is a function if and only if, for all relations R and S,

$$f \cdot R \subseteq S \;\equiv\; R \subseteq f^\circ \cdot S$$

This definition allows functions to be shunted from one side of the partial order to the other in much the same way as the adjoints of a Galois connection can be shunted. By taking the converse of both sides of the equivalence one gets the equivalent statement, for all relations R and S,

$$R \cdot f^\circ \subseteq S \;\equiv\; R \subseteq S \cdot f$$

The proof that these statements are equivalent to the original definition of functions is left to the reader.

2.3.2 The Subcategory *Map*

Since both totality and simplicity are preserved by composition we have that functions are also preserved by composition. Furthermore, since $\mathsf{id} = \mathsf{id}^\circ$, it is easy to verify that id is also a function. As functions are closed under compositions and the identity relation is a function, we have that if we restrict an allegory to functions, we get a subcategory of our allegory. We do not have a suballegory however, for functions are not necessarily closed under meet, join, or reverse. Given an allegory $\mathcal{C}$ we denote the subcategory of functions by $Map(\mathcal{C})$. For Rel we have that $Map(\mathsf{Rel}) = \mathsf{Map}$. (Note that we represented relations in Rel as sets, and we gave no specific representation of functions in Map. Hence, we consider every category which behaves like Map to *be* the category Map and by the equality we actually mean an isomorphism).

As we have seen, relations are an extension of functions in the sense that every function is also a relation. Similarly we have shown that relations are modelled by an allegory, but if we restrict ourselves to functions, then we must also restrict ourselves to a category. In that sense, allegories are the relational extension of categories. In the rest of this thesis, we will often first consider functions and categories, and then show how the concepts we are discussing can be extended to relations and allegories. To do this we will need an extra link between functions and relations that holds between Map and Rel but that we do not assume in general. This link is called tabularity.

2.3.3 Tabulations

Tabularity is a concept which helps us extend ideas formulated for functions in a category to an equivalent concept for relations in an allegory. By an extension we mean that, if we restrict ourselves to the subcategory of functions in our allegory, we should get back the original concept for functions. Often we can extend a concept on functions in more than one way. By using tabulations, which describe a link between relations and functions in Rel and Map, we can often show that for a given functional concept there exists at most one relational extension. Although we do not assume tabulations in general, tabulations do allow us to choose a relational extension that is canonical in some sense.

In Rel a relation $R : A \leftarrow B$ is represented by a subset C of the Cartesian product of A and B where the set C is again an object of Rel. Using the projection arrows $\mathsf{outl} : A \leftarrow C$ and $\mathsf{outr} : B \leftarrow C$ with $\mathsf{outl}.(a,b) = a$ and $\mathsf{outr}.(a,b) = b$ we have that $a\langle R\rangle b$ if and only if there exists an element $c \in C$ such that $\mathsf{outl}.c = a$ and $\mathsf{outr}.c = b$. Thus we can represent the relation R using only the *object* C and the functions outl and outr, thereby representing the relation in Map, the subcategory of functions. Generalizing away from a specific object (the Cartesian product of A and B) and a specific pair of arrows (the projection functions outl and outr) we say that an object C and a pair of functions $f : A \leftarrow C$ and $g : B \leftarrow C$ is a *tabulation* of a relation $R : A \leftarrow B$ if for all $a \in A$ and $b \in B$,

$$a\langle R\rangle b \equiv \exists(c :\ c \in C :\ f.c = a \ \wedge\ g.c = b)$$

and furthermore, that there exists at most one element $c \in C$ which satisfies this condition.

To lift the definition of a tabulation into an allegory we once again make use of the definition of the allegorical operations in Rel. First we lift the condition that $a\langle R\rangle b$ if and only if there exists an element $c \in C$ such that $f.c = a$ and $g.c = b$. We calculate as follows:

$$\begin{array}{ll}
 & \forall(a,b ::\ a\langle R\rangle b \equiv \exists(c ::\ a\langle f\rangle c \wedge b\langle g\rangle c)) \\
\equiv & \quad \{ \text{ converse } \} \\
 & \forall(a,b ::\ a\langle R\rangle b \equiv \exists(c ::\ a\langle f\rangle c \wedge c\langle g^\circ\rangle b)) \\
\equiv & \quad \{ \text{ composition } \} \\
 & \forall(a,b ::\ a\langle R\rangle b \equiv a\langle f \cdot g^\circ\rangle b) \\
\equiv & \quad \{ \text{ equality } \} \\
 & R \ = \ f \cdot g^\circ
\end{array}$$

The other condition, that there can be only one element $c \in C$ such that $f.c = a$ and $g.c = b$ is a bit more tricky. Instead we use the equivalent condition that if we have two elements for which this condition holds, then they must necessarily be equal. This can be lifted into a point free form as follows:

$$
\begin{array}{ll}
 & \forall(c_1, c_2 :\ \exists(a, b ::\ (a\langle f\rangle c_1 \wedge b\langle g\rangle c_1) \wedge (a\langle f\rangle c_2 \wedge b\langle g\rangle c_2)) :\ c_1{=}c_2) \\
\equiv & \qquad \{ \text{ calculus, converse, identity } \} \\
 & \forall(c_1, c_2 :\ \exists(a ::\ c_1\langle f^\circ\rangle a \wedge a\langle f\rangle c_2) \wedge \exists(b ::\ c_1\langle g^\circ\rangle b \wedge b\langle g\rangle c_2) :\ c_1\langle \mathsf{id}\rangle c_2) \\
\equiv & \qquad \{ \text{ composition } \} \\
 & \forall(c_1, c_2 :\ c_1\langle f^\circ \cdot f\rangle c_2 \wedge c_1\langle g^\circ \cdot g\rangle c_2 :\ c_1\langle \mathsf{id}\rangle c_2) \\
\equiv & \qquad \{ \text{ meet } \} \\
 & \forall(c_1, c_2 :\ c_1\langle f^\circ \cdot f \cap g^\circ \cdot g\rangle c_2 :\ c_1\langle \mathsf{id}\rangle c_2) \\
\equiv & \qquad \{ \text{ inclusion } \} \\
 & f^\circ \cdot f \cap g^\circ \cdot g \subseteq \mathsf{id}
\end{array}
$$

Thus in an allegory we say that an object C and a pair of functions $f : A \leftarrow C$ and $g : B \leftarrow C$ is a tabulation of a relation $R : A \leftarrow B$ if

$$R = f \cdot g^\circ \quad \wedge \quad f^\circ \cdot f \cap g^\circ \cdot g \subseteq \mathsf{id}$$

If every relation in an allegory has a tabulation we say that the allegory is *tabular*. As we have seen, the allegory Rel is tabular since every relation $R : A \leftarrow B$ has at least the tabulation with object $C = A \times B$ and the projection functions outl and outr.

2.4 Important Allegorical Concepts

It is necessary to define some extra concepts for allegories, as they will be needed later in this thesis. One of these concepts is division, which represents a universal quantification in much the same way that composition represented an existential quantification. Furthermore, we will introduce two different ways of modeling subsets of sets and domains of relations, namely using partial identities and the domain operators, or a unit object and conditions.

2.4.1 Division

We have two extra operations in an allegory induced by the requirement that composition distributes over arbitrary unions, which play an important role in the theory to come. These operations are called the division operators (or factors). Given two relations $R : A \leftarrow C$ and $S : A \leftarrow B$ with the same target, we can define the left division operation, denoted by the operator $\backslash$, by the following Galois connection; for all relations $X : C \leftarrow B$,

$$X \subseteq R\backslash S \ \equiv\ R \cdot X \subseteq S$$

Note that in this case the lower adjoint is the mapping $(R\,\cdot)$ which maps a relation S to the relation $R \cdot S$, and the upper adjoint is the mapping $(R\backslash)$ which maps a relation S to the relation $R\backslash S$. Thus we have the cancellation properties $R \cdot (R\backslash S) \subseteq S$ and $X \subseteq R\backslash(R \cdot X)$, which are one reason for the name division. Futhermore, $\mathsf{id}\backslash R = R$ for all relations R, which gives another reason for the name.

Similarly, for relations $R : A \leftarrow C$ and $S : B \leftarrow C$ with the same source we can define the right division operation, denoted by the operator $/$, by the following Galois connection; for all $X : A \leftarrow B$,

$$X \subseteq R/S \equiv X \cdot S \subseteq R$$

For the semantics of the division operations in Rel we can derive the following properties,

$$c\langle R\backslash S\rangle b \equiv \forall(a : a\langle R\rangle c : a\langle S\rangle b)$$

and

$$a\langle R/S\rangle b \equiv \forall(c : b\langle S\rangle c : a\langle R\rangle c)$$

To give a concrete example, $c\langle \mathsf{isChildOf}\backslash\mathsf{isOlderThan}\rangle b$ if and only if b is older than all people that c is a child of, in other words, person b must be older than the parents of person c. Similarly, $a\langle \mathsf{isOlderThan}/\mathsf{isChildOf}\rangle b$ if and only if all children of person b are older than person a.

It is possible to derive a huge number of properties about these division operators using the defining Galois connections, and combining the definitions of left and right division. Futhermore, if one combines these definitions with the shunting properties of functions, then one soon has over 20 rules to work with. When these rules are used, they are given the hint "factors" , and it is left as an excercise to the reader to verify the property if it is new.

2.4.2 Partial Identities

A partial identity is a relation contained in id, in other words, a relation X is a partial identity if and only if $X \subseteq \mathsf{id}$. A partial identity $X : A \leftarrow A$ represents a subset of values of type A, namely the set of values $\{a : a\langle X\rangle a\}$. If we have two partial identities X and Y, then $X \subseteq Y$ can actually be interpreted as "X is a subset of Y".

For partial identities we have that $X = X^{\circ}$, which can be proved using the modular law (hint: first prove that $R \subseteq R \cdot R^{\circ} \cdot R$ for every relation R), and thus partial identities are symmetric. Furthermore, composition of partial identities is the same as taking their intersection, i.e., $X \cdot Y = X \cap Y$, which can also be proved using the modular law.

2.4.3 Domains

Using the notion of a partial identity we can define the left and right domain operators which return the subsets of the target and source on which a relation is defined. For example, for a relation $R : A \leftarrow B$ we expect the right domain, denoted by $R>$ to be a partial identity of type $B \leftarrow B$ such that $b\langle R>\rangle b$ if and only if there exists an $a \in A$ such that $a\langle R\rangle b$. In Rel we can calculate as follows,

$$
\begin{array}{ll}
 & \forall(b :: b\langle R>\rangle b \equiv \exists(a :: a\langle R\rangle b)) \\
\equiv & \quad \{ \text{ converse } \} \\
 & \forall(b :: b\langle R>\rangle b \equiv \exists(a :: b\langle R^{\circ}\rangle a \wedge a\langle R\rangle b)) \\
\equiv & \quad \{ \text{ composition } \} \\
 & \forall(b :: b\langle R>\rangle b \equiv b\langle R^{\circ} \cdot R\rangle b) \\
\equiv & \quad \{ \text{ calculus } \} \\
 & \forall(b_1, b_2 :: b_1\langle R>\rangle b_2 \equiv b_1\langle R^{\circ} \cdot R\rangle b_2 \wedge b_1\langle \mathsf{id}\rangle b_2) \\
\equiv & \quad \{ \text{ meet, equality } \} \\
 & R> = R^{\circ} \cdot R \cap \mathsf{id}
\end{array}
$$

Thus we have a closed formula for the right domain operator in an allegory. Since the left domain is the dual concept of the right domain, i.e. the left domain $R<$ of a relation R is simply the right domain of R°, we have,

$$R< \;=\; R \cdot R^\circ \cap \mathsf{id}$$

It is also possible to define the domain operators in terms of a Galois connection. First of all, consider the relation $R \cdot X$ for a partial identity X; here the partial identity X restricts the domain of R to the subset of values that X represents. However, if $R> \subseteq X$ then X should not restrict R, and similarly if X does not restrict R we should have that $R> \subseteq X$. Since for every relation R we have that $R \subseteq \top\!\top$, we can define the right domain operator by the following Galois connection, namely for all partial identities X and relations R,

$$R> \subseteq X \;\equiv\; R \subseteq \top\!\top \cdot X$$

It is easy to verfiy that the closed formula for $R>$ given above does indeed solve this equation, and since the right domain operator is the lower adjoint of the function $(\top\!\top \cdot)$ we know this solution must be unique. By taking the converse of both sides of the above Galois connection we get a Galois connection for the left domain operator. Again we can use the Galois connections to derive a multitude of properties for the domain operators, such as their monotonicity, but we will not do so here.

2.4.4 Unit

Another important allegorical concept which we will need later on, is the unit. A unit represents an object with only one (anonymous) value, which we will denote by $*$. In a category, there should be exactly one arrow from every object A to a unit object, namely the constant arrow which maps every single value in A to the unique value $*$.

In the category Map any singleton set can serve as a unit, since it only has one element, and thus in a category there may be more than one object which is a unit. Let us pick an arbitrary unit, however, and denote it by $\mathsf{1}$. Let us denote the unique function to $\mathsf{1}$ from a given object A by $!_A$, then the requirement that $!_A$ is unique is given by the equivalence, for all f,

$$f : \mathsf{1} \leftarrow A \;\equiv\; f = !_A$$

To extend the concept of a unit to an allegory we want to express the fact that it contains only one element in an allegorical setting. This can be done by the following two conditions,

$$\top\!\top_{A,1} \cdot \top\!\top_{1,B} = \top\!\top_{A,B}$$

and

$$\top\!\top_{1,1} \subseteq \mathsf{id}_1$$

The first expression simply states that $\mathsf{1}$ cannot be empty, which can easily be seen by looking at the interpretation of this expression in Rel, and the second expression says that $\mathsf{1}$ can only have a single element. Using this definition and by taking $!_A := \top\!\top_{1,A}$ one can show that $!_A$ is a function, and using the fact that $f = g \equiv f \subseteq g$ for functions f and g one can easily show that $!_A$ is the unique function of its type. Thus $\mathsf{1}$ is also a unit in the subcategory of functions, and the definition of a unit in an allegory is a relational extension of the categorical concept.

Using tabulations, if we have that $\mathsf{1}$ is a unit in the subcategory of functions, we can show that it is also an allegorical unit. To do this we first prove that for all relations $R : \mathsf{1} \leftarrow \mathsf{1}$ we have that $R \subseteq \mathsf{id}_1$. Let (f,g) be a tabulation of R, i.e., $R = f \cdot g^\circ$, then

$$
\begin{array}{ll}
 & R \subseteq \mathsf{id}_1 \\
\equiv & \quad \{ \text{ tabulations } \} \\
 & f \cdot g^{\circ} \subseteq \mathsf{id}_1 \\
\equiv & \quad \{ \text{ shunting of functions } \} \\
 & f \subseteq g \\
\equiv & \quad \{ \text{ functions } \} \\
 & f = g \\
\equiv & \quad \{ \ f : \mathsf{1} \leftarrow C \text{ and } g : \mathsf{1} \leftarrow C\text{, unit } \} \\
 & \mathit{true}
\end{array}
$$

Furthermore, it is easy to show that $!^{\circ}_{A} \cdot !_{B} = \top\!\!\top_{A,B}$, and since $!_{A} = \top\!\!\top_{1,A}$, the other condition, namely that $\top\!\!\top_{A,1} \cdot \top\!\!\top_{1,B} = \top\!\!\top_{A,B}$, follows by substitution. Thus we have that $\mathsf{1}$ is also an allegoric unit, and our definition of a unit in an allegory is a good definition.

2.4.5 Conditions

We can use a specific unit type to model subsets. A subset of an object A can be modelled by a relation C such that $C \subseteq !_{A}$, where C represents the set of values $\{a \ : \ *\langle C \rangle a\}$. Such a relation is called a right condition. Dually a subset of A can be modelled by a relation $C \subseteq !^{\circ}_{A}$, called a left condition, where C represents the set of values $\{a \ : \ a\langle C \rangle *\}$. Note that there exists an isomorphism between partial identities and conditions. For right conditions we have that, given a partial identity X and a right condition C,

$$X = C_{>} \ \equiv \ ! \cdot X = C$$

For left conditions we only need to take the converse of both sides. Note that we can model the right domain of a relation R by the right condition $! \cdot R$ since,

$$
\begin{array}{ll}
 & *\langle ! \cdot R \rangle a \\
\equiv & \quad \{ \text{ composition } \} \\
 & \exists (b :: \ *\langle ! \rangle b \wedge b \langle R \rangle a) \\
\equiv & \quad \{ \ ! = \top\!\!\top \ \} \\
 & \exists (b :: \ b \langle R \rangle a)
\end{array}
$$

Similarly we can use left conditions to model left domains.

2.5 Product Categories and Allegories

We need one more concept before we continue with a discussion of generic datatypes, namely that of product categories and allegories. Given two categories $\mathcal{C}$ and $\mathcal{D}$ we can define the product category $\mathcal{C} \times \mathcal{D}$ by letting the objects of $\mathcal{C} \times \mathcal{D}$ be pairs (A,B) of objects A from $\mathcal{C}$ and B from $\mathcal{D}$, by letting arrows of $\mathcal{C} \times \mathcal{D}$ be pairs (f,g) of arrows f from $\mathcal{C}$ and g from $\mathcal{D}$, and by defining all operations componentwise, i.e., for target and source we have $(f,g)_{\triangleleft} = (f_{\triangleleft}\, , g_{\triangleleft})$ and $(f,g)_{\triangleright} = (f_{\triangleright}\, , g_{\triangleright})$, for composition we have $(f,g) \cdot (h,k) = (f \cdot h\, , g \cdot k)$, and for the identity we have that $\mathsf{id}_{(A,B)} = (\mathsf{id}_{A}, \mathsf{id}_{B})$. It is easy to verify that $\mathcal{C} \times \mathcal{D}$ defined in this fashion is indeed a category.

Extending the idea of products to allegories, given two allegories $\mathcal{C}$ and $\mathcal{D}$ we can define their product $\mathcal{C}\times\mathcal{D}$ in the same way, namely by defining all the extra operations of an allegory componentwise as well. Thus we have, for example, that $(R,S)\cap(T,U) = (R\cap T\,,S\cap U)$ and that $(R,S)^\circ = (R^\circ\,,S^\circ)$. Again it is easy to verify that $\mathcal{C}\times\mathcal{D}$ defined in this way is indeed an allegory.

Obviously we need not restrict ourselves to vectors of length two, and we can take products of an arbitrary number of categories or allegories. All operations on vectors are simply defined componentwise.

Chapter 3

Generic Datatypes

Categories and allegories are the algebraic structures we use to model functional and relational programming respectively. The arrows in categories and allegories represent functions and relations respectively, and the objects of categories and allegories represent the datatypes on which the programs (arrows) operate. Just how we use objects to represent complex datatypes such as pairs, lists, trees and natural numbers in a generic way, is the topic of this chapter.

The interpretation of objects is that they represent a collection of values of a certain type. Thus we have an object which represents pairs of integers, another object which represents trees of lists of characters, and yet another which represents pairs where the first element is a character and the second element is a rational number.

To create a useful and uniform theory of datatypes, it must be possible to do two things. First of all, if we have pairs where the first element is an integer and the second element is a character, then we should have pairs of lists of integers, pairs of trees, and pairs where the first element is a tree of strings and the second element is a list of integer pairs as well. In fact, given any datatypes or objects A and B we should also have an object C which is the datatype of pairs of A's and B's. This principle, that there should not be an arbitrary restriction on the datatypes we have, is called the type completeness principle.

The second requirement is that we should be able to take composite types apart again and inspect the parts. For example, if C is the datatype of pairs of A's and B's and (a,b) is such a value in C where a is a value of A and b is a value of B, then we expect to have the projection functions $\mathsf{outl} : A \leftarrow C$ and $\mathsf{outr} : B \leftarrow C$ where $\mathsf{outl}.(a,b) = a$ and $\mathsf{outr}.(a,b) = b$.

Apart from the aspects of datatype formation and decomposition, we will also require the existence of a generic *map* operation. On the datatype of lists, the *map* operation is very well known; it takes a function of type $A \leftarrow B$ and a list of B's and applies the function to every element of the list to obtain a list of A's. We require all datatypes to have a map operation which allows us to map a function or relation on every single element contained in a data structure of that type.

3.1 Functors and Relators

In categories and allegories we combine the notion of a datatype former and the generic map function into a single concept, namely the notion of a functor in a category and that of a relator in an allegory. We will show that relators are an extension of functors, that is, if we restrict relators to functions we get a functor on functions, and we will show that if we assume tabulations, then every functor has at most one relational extension.

3.1.1 Definition

A functor consists of two mappings to a category $\mathcal{C}$ from a category $\mathcal{D}$, one which maps the arrows of $\mathcal{D}$ to the arrows of $\mathcal{C}$, and one which maps the objects of $\mathcal{D}$ to the objects of $\mathcal{C}$. We will use the capital sans-serif letters F, G, H, ..., and capitalized sans-serif identifiers like Id and List to denote functors, and we will use the same symbol to denote both the object map and the arrow map. If a functor maps arrows and objects to a category $\mathcal{C}$ from a category $\mathcal{D}$, we write $\mathsf{F} : \mathcal{C} \leftarrow \mathcal{D}$.

Since the object map of a functor represents a datatype former, for example, $\mathsf{List}A$, represents the datatype of lists over a datatype A, and since the arrow map represents the generic *map* function, we require the following properties to hold for a functor F; for all arrows $f : A \leftarrow B$

$$(\mathsf{F}f){\lhd} = \mathsf{F}(f{\lhd}) = \mathsf{F}A$$

and

$$(\mathsf{F}f){\rhd} = \mathsf{F}(f{\rhd}) = \mathsf{F}B$$

Furhermore, since the identity function id does not change elements, we do not expect a value of datatype to change if we map id on every value it contains. In other words, for a functor F and all objects A we require that

$$\mathsf{F}\mathsf{id}_A = \mathsf{id}_{\mathsf{F}A}$$

Finally, given two functions $f : A \leftarrow B$ and $g : B \leftarrow C$ it should not make any difference if we first map the function g on a value of our datatype and then map the function f on this value, or if we directly map the composition $f \cdot g$ on our value. In other words, we require that

$$\mathsf{F}(f \cdot g) \;=\; \mathsf{F}f \;\cdot\; \mathsf{F}g$$

A relator is a functor between two allegories $\mathcal{C}$ and $\mathcal{D}$ with some extra requirements induced by the fact that we are mapping relations on data structures instead of only functions. As an example of a relator, for two lists of the same length we have,

$$[a_1,a_2,\ldots,a_n]\langle\mathsf{List}R\rangle[b_1,b_2,\ldots,b_n] \equiv \forall(i :: \; a_i\langle R\rangle b_i)$$

The first requirement on a relator F is that if $R \subseteq S$ then mapping S on a data structure should produce at least those data structures produced by mapping R on it, and thus,

$$\mathsf{F}R \subseteq \mathsf{F}S \;\Leftarrow\; R \subseteq S$$

In other words, relators are required to be monotonic. The second requirement concerns the reverse structure; if we consider the above example, we have that,

$$
\begin{array}{ll}
 & [a_1,a_2,\ldots,a_n]\langle\mathsf{List}(R^\circ)\rangle[b_1,b_2,\ldots,b_n] \\
\equiv & \qquad \{ \text{ definition } \} \\
 & \forall(i :: \; a_i\langle R^\circ\rangle b_i) \\
\equiv & \qquad \{ \text{ converse } \} \\
 & \forall(i :: \; b_i\langle R\rangle a_i) \\
\equiv & \qquad \{ \text{ definition } \} \\
 & [b_1,b_2,\ldots,b_n]\langle\mathsf{List}R\rangle[a_1,a_2,\ldots,a_n] \\
\equiv & \qquad \{ \text{ converse } \} \\
 & [a_1,a_2,\ldots,a_n]\langle(\mathsf{List}R)^\circ\rangle[b_1,b_2,\ldots,b_n]
\end{array}
$$

In other words, $\mathsf{List}(R^\circ) = (\mathsf{List}R)^\circ$, and the reverse operation distributes over List. Considering the interpretation of the map operation we expect this result to hold for every relator F, and thus we require,

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

Note that we can now safely write $\mathsf{F}R^\circ$ to mean either $\mathsf{F}(R^\circ)$ or $(\mathsf{F}R)^\circ$, and we can omit the parantheses. Note that in a tabular allegory it is possible to derive this requirement from the monotonicity requirement, but in general it is not.

3.1.2 The Link Between Functors and Relators

We will now show that relators are a relational extension of functors, and that if we assume tabularity, every functor has at most one relational extension, suggesting a canonical extension which one should choose. To prove the first assertion, we need only prove that $\mathsf{F}f$ is a function if f is function, since distribution over composition and preservation of identities hold by the definition of a relator. To prove that $\mathsf{F}f$ is a function we need to prove that it is a simple and total relation; first we prove that $\mathsf{F}f$ is simple,

$$\begin{aligned}
& \mathsf{F}f \cdot (\mathsf{F}f)^\circ \\
= \quad & \{ \text{ relators (reverse, composition) } \} \\
& \mathsf{F}(f \cdot f^\circ) \\
\subseteq \quad & \{ \text{ } f \text{ simple, relators (monotonicity) } \} \\
& \mathsf{F}\mathsf{id} \\
= \quad & \{ \text{ relators (identity) } \} \\
& \mathsf{id}
\end{aligned}$$

The proof that $\mathsf{F}f$ is total is practically identical, and is left to the reader. Note that we needed every one of the four defining requirements of a relator to prove this result.

To prove that there exists at most one extension of a functor F we need to show that given two relators F and G which behave the same on functions, i.e. $\mathsf{F}f = \mathsf{G}f$ for all functions f, then they must behave the same on relations as well. Assuming tabularity, let (f,g) be a tabulation of a relation R, then,

$$\begin{aligned}
& \mathsf{F}R \\
= \quad & \{ \text{ tabulations } \} \\
& \mathsf{F}(f \cdot g^\circ) \\
= \quad & \{ \text{ relators } \} \\
& \mathsf{F}f \cdot (\mathsf{F}g)^\circ \\
= \quad & \{ \text{ } \mathsf{F} \text{ and } \mathsf{G} \text{ agree on functions } \} \\
& \mathsf{G}f \cdot (\mathsf{G}g)^\circ \\
= \quad & \{ \text{ as above } \} \\
& \mathsf{G}R
\end{aligned}$$

Thus, under the assumption of tabularity there can be at most one relational extension of a functor. Note that not every functor has a relational extension, as Oege de Moor shows in [9].

3.1.3 F-Structures and Shapes

The object map of a relator F maps an object A to an object representing a datatype over the type A. Thus a relator is a polymorphic datatype former, since for every object A we have a datatype $\mathsf{F}A$. We call a value of the type $\mathsf{F}A$ for some specific object A an $\mathsf{F}A$-structure, and a value of the type $\mathsf{F}A$ for an arbitrary A will be called an F-structure. By filling in the specific relator List, the interpretation should be clear.

An important property of F-structures is their shape. To define the shape of an F-structure we can make use of the unit type $\mathbb{1}$. To compare the shape of two F-structures we want to somehow disregard the values stored in them, and this can be done by applying the function ! to both of them. Recall that ! maps every value in the structure to the anonymous value $*$ of the unit type, and thus two $\mathsf{F}\mathbb{1}$-structures are equal if and only if they have the same shape. By identifying the shape of an $\mathsf{F}A$-structure x with the $\mathsf{F}\mathbb{1}$-structure $(\mathsf{F}!).x$ we can define a generic notion of shape.

Since $\mathsf{F}R \subseteq \mathsf{F}\top\!\!\top$, and since $\top\!\!\top = !^\circ \cdot !$, we have that for all F-structures x and y,

$$
\begin{array}{ll}
 & x\langle \mathsf{F}R\rangle y \\
\Rightarrow & \quad \{ \text{ above } \} \\
 & x\langle \mathsf{F}(!^\circ \cdot !)\rangle y \\
\equiv & \quad \{ \text{ relators } \} \\
 & x\langle \mathsf{F}!^\circ \cdot \mathsf{F}!\rangle y \\
\equiv & \quad \{ \text{ composition, converse } \} \\
 & \exists(s :: s\langle \mathsf{F}!\rangle x \wedge s\langle \mathsf{F}!\rangle y)
\end{array}
$$

And thus, since relators preserve functions, x and y have the same shape, and relators preserve shape. Furthermore, by instantiating R with $\top\!\!\top$, the above implication becomes an equivalence and we see that two F-structures have the same shape if and only if they are related by $\mathsf{F}\top\!\!\top$.

3.1.4 Categories of Functors and Relators

Given a relator $\mathsf{F}:\mathcal{C}\leftarrow\mathcal{D}$ and a relator $\mathsf{G}:\mathcal{D}\leftarrow\mathcal{E}$ then we can define a new relator FG such that $\mathsf{FG}A = \mathsf{F}(\mathsf{G}A)$ and $\mathsf{FG}R = \mathsf{F}(\mathsf{G}R)$. It is easy to verify that this is indeed a relator, namely of type $\mathcal{C}\leftarrow\mathcal{E}$. Futhermore, it is possible to define a relator $\mathsf{Id}:\mathcal{C}\leftarrow\mathcal{C}$ for an arbitrary allegory $\mathcal{C}$ such that $\mathsf{Id}A = A$ and $\mathsf{Id}R = R$. If we then consider relators to be arrows and categories to be objects, we have a category of relators. Since relators are the extension of functors, we can also define a category of functors in the same way. Note that it does not make sense to define an allegory of relators since relators are functions (from objects to objects and relations to relations) and not relations.

3.2 The Standard Datatypes

Now that we have defined the notions of functors and relators as the abstract notion of a datatype former paired with a polymorphic map operation, we can introduce specific functors and their relational extensions. With these functors and relators it will be possible to define all the well-known datatypes, such as Cartesian products, disjoint unions, power sets, and recursive datatypes such as lists and trees. Since functors and relators form a category, we can combine them (i.e., compose them) to form complex datatypes.

3.2.1 The Identity Relator

One of the most trivial datatype formers is the identity functor $\mathsf{Id}:\mathcal{C}\leftarrow\mathcal{C}$ which leaves both objects and arrows identical, i.e., $\mathsf{Id}A=A$ and $\mathsf{Id}f=f$ for all objects A and arrows f. In other words, an element of an $\mathsf{Id}A$-structure is simply a value of A and mapping an arrow f on this structure is accomplished by simply mapping the arrow f on that value. It is trivial to verify that the identity functor is also a relator.

3.2.2 The Constant Relators

Another trivial example of datatype formers are the constant functors. Given an object A of a category $\mathcal{C}$, the constant functor $\mathsf{K}_A:\mathcal{C}\leftarrow\mathcal{D}$ takes an arbitrary object in the category $\mathcal{D}$ and maps it to the object A, i.e. $\mathsf{K}_A B=A$ for all objects B in $\mathcal{D}$. The arrow map is defined by $\mathsf{K}_A f=\mathsf{id}_A$ for every arrow f in $\mathcal{D}$. The interpretation of a K_A-structure is that it is a data structure containing a constant value of type A. The structure contains no data in the sense that mapping a function on the data structure leaves the data structure unchanged. In other words, the structure contains no values which can be changed, and thus it is a constant. The constant functors are also relators.

3.2.3 The Product Relator

The product functor is the datatype former and the map operation for the Cartesian product of two datatypes. To model the Cartesian product of two datatypes in a category $\mathcal{C}$, we require the existence of a binary mapping $\times:\mathcal{C}\leftarrow\mathcal{C}^2$ (where $\mathcal{C}^2$ represents the product category $\mathcal{C}\times\mathcal{C}$) which associates a product object $A\times B$ to a pair of objects A and B. Furthermore, we want to be able to decompose every product object $A\times B$ into its left and right components A and B respectively, and thus we require the existence of two projection arrows $\mathsf{outl}_{A,B} : A\leftarrow A\times B$ and $\mathsf{outr}_{A,B} : B\leftarrow A\times B$. Finally, given two functions $f:A\leftarrow C$ and $g:B\leftarrow C$ with the same source, we want to be able to construct a product structure of type $A\times B$ from a value of C, by putting the result of f in the left component and the result of g in the right component. We will denote this function, called the *split* of f and g, by $f\vartriangle g$. For the split and the projections we have the following equivalence,

$$h = f\vartriangle g \quad\equiv\quad \mathsf{outl}\cdot h = f \;\wedge\; \mathsf{outr}\cdot h = g$$

Using this definition of the product, we can also define a polymorphic map function on product objects which takes a pair of arrows $f:A\leftarrow C$ and $g:B\leftarrow D$ and maps this pair on a value of $C\times D$ to obtain a value of $A\times B$. The left component of this map function consists of first projecting the value of type A out of the product, and then applying the function f on it. The right component can be found similarly using g, and the result can be combined using the split operation, and thus we have,

$$f\times g \;:=\; (f\cdot\mathsf{outl})\vartriangle(g\cdot\mathsf{outr})$$

Now we have that $\times$ maps objects of the category $\mathcal{C}^2$ to objects of the category $\mathcal{C}$, and similarly, it takes arrows of the category $\mathcal{C}^2$ to arrows of the category $\mathcal{C}$. Furthermore, it is easy to verify that $\mathsf{id}_A\times\mathsf{id}_B=\mathsf{id}_{A\times B}$ and that $(f\times g)\cdot(h\times k)=(f\cdot h)\times(g\cdot k)$, so that $\times$ is a functor.

To extend the product functor to work on relations we need to be a bit more careful in the definition of the projection arrows and the split operation since relations need not be simple or total. For the projections, we obviously want them to be functions, and furthermore, to every pair of values of A and B we want there to correspond a unique value of $A\times B$. These conditions are equivalent to saying that $(\mathsf{outl}_{A,B},\mathsf{outr}_{A,B})$ is a tabulation of the relation $\top\!\top_{A,B}$. Since the requirements of the projections are much stricter in the allegorical settings it is possible to directly define the relational extension of the split operation in terms of the projections. Reasoning from Rel we want that $x\langle R \vartriangle S\rangle y$ if the left component of x is related by R to y and the right component of x is related by S to y; lifting this definition to an allegory we have,

$$
\begin{array}{ll}
 & \forall(a : a\langle \mathsf{outl}\rangle x : a\langle R\rangle y) \ \wedge\ \forall(b : b\langle \mathsf{outr}\rangle x : b\langle S\rangle y) \\
\equiv & \qquad \{ \text{ factors } \} \\
 & x\langle \mathsf{outl}\backslash R\rangle y \wedge x\langle \mathsf{outr}\backslash S\rangle y \\
\equiv & \qquad \{ \text{ meet } \} \\
 & x\langle \mathsf{outl}\backslash R \cap \mathsf{outr}\backslash S\rangle y
\end{array}
$$

Thus we can define $R \vartriangle S \ := \ \mathsf{outl}\backslash R \cap \mathsf{outr}\backslash S$. Using this definition of a split we can define the product of two relations in the same fashion as we did in a category, namely,

$$R \times S \ := \ (R \cdot \mathsf{outl}) \vartriangle (S \cdot \mathsf{outr})$$

The proof that $\times$ is a relator is simple, apart from the requirement that $\times$ distributes over composition. The reason for this is that composition of arbitrary relations does not distribute over meet and one has to make use of the modular law. The proof that $\times$ is the canonical relational extension of the product functor is not difficult, and both can be found in [6].

3.2.4 The Coproduct Relator

The coproduct functor is the datatype former and operation for the disjoint union of two datatypes. To model the disjoint union or sum of two datatypes in a category $\mathcal{C}$ we require the existence of a binary operation $+ : \mathcal{C} \leftarrow \mathcal{C}^2$ which associates an object $A+B$ to each pair of objects A and B. A value from $A+B$ then represents either a value from A or a value from B, and we want to be able to see whether the value originated in A or B. To do this we require the existence of two injection arrows $\mathsf{inl}_{A,B} : A+B \leftarrow A$ and $\mathsf{inr}_{A,B} : A+B \leftarrow B$ which add a tag to the original value to indicate its source. Futhermore, for each pair of arrows $f : A \leftarrow B$ and $g : A \leftarrow C$ with the same target we want it to be possible to construct an arrow which takes a value of type $B+C$ and applies f if the value originated in B and otherwise applies g if the value originated in C. We will call this function the *junc* of f and g and denote it by $f \triangledown g$. For the junc operation and the injections we have the equivalence,

$$h = f \triangledown g \ \equiv \ h \cdot \mathsf{inl} = f \ \wedge \ h \cdot \mathsf{inr} = g$$

Using this definition of the coproduct we can define a polymorphic map function on coproduct objects which takes a pair of arrows $f : A \leftarrow C$ and $g : B \leftarrow D$ and maps this pair on a value of $C+D$ to obtain a value of $A+B$. The definition is as follows,

$$f + g \ := \ (\mathsf{inl} \cdot f) \triangledown (\mathsf{inr} \cdot g)$$

Just as with the product, we now have that $+$ maps pairs of objects and pairs of arrows of $\mathcal{C}^2$ to objects and arrows of $\mathcal{C}$ respectively, and it is easy to show that $+$ is a functor.

To extend the coproduct functor to relations we must again be more careful with the definition of the injections and the junc operation. We obviously want the injections to be functions, and we want them to be injective since we want to be able to decompose a disjoint sum and recover the original values. Futhermore, we want the sum to be disjoint, that is, a value of $A+B$ may not originate from both A and B. This can be formulated as $\mathsf{inl}^\circ \cdot \mathsf{inr} = \perp\!\!\perp$, which is easy to verify using the interpretations in Rel. Finally, we want the disjoint sum to contain all the values of both A and B, which can be formulated as $\mathsf{inl} \cdot \mathsf{inl}^\circ \cup \mathsf{inr} \cdot \mathsf{inr}^\circ \ = \ \mathsf{id}$.

As we did with the product, we can now use these stricter definitions of the injections to define the junc operation. Given relations $R : A \leftarrow B$ and $S : A \leftarrow C$ we have that $x\langle R \triangledown S\rangle y$ if y originates from B and this value is related to x via R or y originates from C and this value is related to x via S. Again we can lift this condition from Rel to an arbitrary allegory as follows,

$$
\begin{aligned}
& \forall(b : y\langle \mathsf{inl}\rangle b : x\langle R\rangle b) \;\wedge\; \forall(c : y\langle \mathsf{inr}\rangle c : x\langle S\rangle c) \\
\equiv \quad & \qquad \{ \text{ factors } \} \\
& x\langle R/\mathsf{inl}\rangle y \wedge x\langle S/\mathsf{inr}\rangle y \\
\equiv \quad & \qquad \{ \text{ meet } \} \\
& x\langle R/\mathsf{inl} \cap S/\mathsf{inr}\rangle y
\end{aligned}
$$

Thus we can define $R \triangledown S \;:=\; R/\mathsf{inl} \cap S/\mathsf{inr}$. Using this definition of junc we can define the coproduct of two relations in the same way that we did in a category, namely as follows,

$$R{+}S \;:=\; (\mathsf{inl} \cdot R) \triangledown (\mathsf{inr} \cdot S)$$

To prove that this defines a relator is considerably easier than in the case of the product relator since composition distributes over arbitrary unions. This proof and the proof that the coproduct relator is the canonical extension of the coproduct functor are left to the reader.

3.2.5 Polynomial Relators

Using the identity, constant, product and coproduct functors repeatedly (by composing them together) one can construct a complex class of datatypes called the polynomial datatypes. Polynomial datatypes account for all the primitive types and constants, arbitrary Cartesian products of datatypes, arbitrary disjoint unions, and any complex combination of these.

Two important classes of datatypes that the polynomial relators do not include are powersets and recursive datatypes. Powersets allow one to model sets over a certain type and recursive datatypes allow one to model all kinds of types which contain another element of the same type somewhere inside of it. Lists and binary trees are two good examples, a list being either the empty list or an element followed by another list, and a binary tree being either the empty tree or a value with a left and a right subtree.

3.2.6 The Power Relator

To model sets we require the existence of a mapping $\mathsf{P} : \mathcal{C} \leftarrow \mathcal{C}$ which maps every object A in a category $\mathcal{C}$ to a power object $\mathsf{P}A$, where $\mathsf{P}A$ represents the datatype of sets over the type A. Futhermore, given a value of $\mathsf{P}A$ we want to be able to recover the values of A contained in the set it represents, and to do this we require the existence of a membership relation $\in_A \,:\, A \leftarrow \mathsf{P}A$. Note that membership is a relation and not a function, so we will only consider sets in a relational setting. Finally, given a relation $R : A \leftarrow B$ we want to be able to create a set valued function which for each value in A returns the set of values in $\mathsf{P}B$ related to it via R. We call this function the power transpose of R and we have that $\Lambda R : \mathsf{P}B \leftarrow A$. For the membership relation and the power transpose we require that the following property holds, for all f,

$$f = \Lambda R \;\equiv\; \in \cdot f = R$$

To make a relator out of the mapping P we still need the polymorphic map function on sets. Given a relation R, we want $a\langle \mathsf{P}R\rangle b$ if and only if every single element of the set a is related via R to a value in the set b, and that for every value in the set b there exists at least one element in the set a related to it via R. We can write this in Rel as follows,

$$\forall(x : x\langle\in\rangle a : \exists(y : y\langle\in\rangle b : x\langle R\rangle y)) \;\wedge\; \forall(y : y\langle\in\rangle b : \exists(x : x\langle\in\rangle a : x\langle R\rangle y))$$

$$\equiv \quad \{ \text{ calculus } \}$$

$$\forall(x : x\langle\in\rangle a : \exists(y :: x\langle R\rangle y \wedge y\langle\in\rangle b)) \;\wedge\; \forall(y : y\langle\in\rangle b : \exists(x :: x\langle\in\rangle a \wedge x\langle R\rangle y))$$

$$\equiv \quad \{ \text{ converse, composition } \}$$

$$\forall(x : x\langle\in\rangle a : x\langle R\cdot \in\rangle y) \;\wedge\; \forall(y : y\langle\in\rangle b : x\langle \in^\circ \cdot R\rangle y)$$

$$\equiv \quad \{ \text{ converse, factors } \}$$

$$x\langle \in \backslash (R\cdot \in)\rangle y \;\wedge\; x\langle (\in^\circ \cdot R) \,/\, \in^\circ\rangle y$$

$$\equiv \quad \{ \text{ meet } \}$$

$$x\langle \in \backslash (R\cdot \in) \,\cap\, (\in^\circ \cdot R) \,/\, \in^\circ\rangle y$$

Thus we will define the mapping P on relations as follows,

$$\mathsf{P}R \;:=\; \in \backslash (R\cdot \in) \,\cap\, (\in^\circ \cdot R) \,/\, \in^\circ$$

To prove that P defines a relator is very tricky and requires the assumption of tabulations. The proof is not included here, but the interested reader can see [3] or [6].

3.3 Recursive Datatypes

The final class of datatypes that we will introduce in terms of functors and relators is the class of recursive datatypes.

3.3.1 F-Algebras and F-Homomorphisms

A recursive datatype is a datatype which contains itself somewhere inside of it. Examples of recursive datatypes are lists and trees. A list is either the empty list, or it consists of a value and another list. Similarly, a binary tree is either the empty tree, or it consists of a value and two other binary trees, namely the left and right subtrees. Just as the $\mathsf{inl}_{A,B}$ creates a coproduct value of type $A+B$ out of a value of type A, for recursive datatypes we expect to have an arrow which creates a recursive datatype A from some other datatype formed by a functor F which contains the datatype A inside of it. In other words, for a recursive datatype, we expect there to exist a constructor arrow of the type $A \leftarrow \mathsf{F}A$.

Let us consider the type of lists over a type A, which we will denote by *ListA*. First we consider the case of the empty list. The empty list is a constant , and thus we can represent it by the unique element in 1. Since the element of 1 is not a list, we have to make it one, which we do with an arrow nil of type $\mathit{ListA} \leftarrow \mathsf{K}_1\mathit{ListA}$. Furthermore, lists which are not empty can be represented by a pair of values where the first element of the pair is a value of type A and the second element of the pair is the rest of the list, and thus a value of type *ListA*. Thus we can use a value of the type $A\times\mathit{ListA}$ to represent non-empty lists, and since values of this type are not actually lists, we have to make it one using an arrow cons of type $\mathit{ListA} \leftarrow A\times\mathit{ListA}$. Now we have that type *ListA* is actually the disjoint sum of the empty list and non-empty lists so that we can define a constructor

$$\mathsf{nil}\triangledown\mathsf{cons} \;:\; \mathit{ListA} \;\leftarrow\; \mathsf{K}_1\mathit{ListA} + (A\times\mathit{ListA})$$

By defining the relator F by $\mathsf{F}X \;:=\; \mathsf{K}_1X + A\times X$ we see that the above constructor arrow indeed has the type $\mathit{ListA} \leftarrow \mathsf{F}\mathit{ListA}$.

An arrow which has the type $A \leftarrow \mathsf{F}A$ is called an F-algebra where the object A is called the carrier of the type. Connected with the notion of an F-algebra is that of an F-homomorphism. An F-homomorphism is an arrow $h : A \leftarrow B$ which converts an F-algebra $f : A \leftarrow \mathsf{F}A$ with carrier A to an F-algebra $g : B \leftarrow \mathsf{F}B$ with a carier B. For h we require that,

$$h \cdot g \;=\; f \;\cdot\; \mathsf{F}h$$

Note that there are exactly two ways to go from a value of type $\mathsf{F}B$ to a value of A using the arrow h, and h is an F-homomorphism if it does not matter which way one chooses.

As an example of an F-homomorphism consider a function $h : B \leftarrow \mathit{ListA}$ which takes a list and converts it to a value of the type B. Normally, a recursive definition of h would look as follows: first we define h on the empty list,

$$h.[] = c$$

and then we would define the value of h on a non-empty list $\mathsf{cons}.(a,x)$ in terms of the value of h applied to the rest of the list x,

$$h.(\mathsf{cons}.(a,x)) = f.(a\;,\;h.x)$$

Here c is a constant function of type $B \leftarrow \mathsf{K}_1 B$ mapping the sole inhabitant of $\mathsf{1}$ to a constant value of B, and f is a binary function of type $B \leftarrow A \times B$. With a bit of calculus, the point-wise definitions above can be rewritten in a point-free fashion as follows,

$$h \cdot (\mathsf{nil} \triangledown \mathsf{cons}) \;=\; (c \triangledown f) \;\cdot\; \mathsf{K}_1 h + (\mathsf{id}_A \times h)$$

Since $c \triangledown f : B \leftarrow \mathsf{K}_1 B + (A \times B)$ we have that $c \triangledown f$ is an F-algebra with carrier B. Futhermore, the above equation says that h is an F-homomorphism. A nice example of h is obtained when c is replaced by the constant function which returns the natural number 0, and f is the binary operator $+$ on natural numbers. In this case we have that

$$h.(\mathsf{cons}.(3\;,\;\mathsf{cons}.(2,\mathsf{nil}))) = \;+.(3\;,\;+.(2,0)) = 3{+}2{+}0 = 5$$

Here we clearly see what an F-homomorphism does: it replaces the constructor arrow (in this case $\mathsf{nil} \triangledown \mathsf{cons}$) with a new arrow (in this case $c \triangledown f$), throughout the entire structure. Thus in this case all the cons operations in the list are replaced by the $+$ operator and the nil operation is replaced by the constant 0. In this example h is simply the function which calculates the sum of the values in a list of integers.

3.3.2 Initial Algebras

In the previous section we saw that we could construct a datatype using an F-algebra and that we could define functions on an F-algebra by following the recursion structure using an F-homomorphism. In the above example we saw that h was a solution of the equation,

$$h \cdot (\mathsf{nil} \triangledown \mathsf{cons}) \;=\; (c \triangledown f) \;\cdot\; \mathsf{K}_1 h + (\mathsf{id}_A \times h)$$

Furthermore, we claimed that we had defined h by this equation. The claim that h is well defined is the same as the claim that the above equation has exactly one solution. That is, h is the unique homomorphism to an arbitrary F-algebra $c \triangledown f$ from the F-algebra $\mathsf{nil} \triangledown \mathsf{cons}$.

In general, if we have some F-algebra $\mathsf{in} : \mathsf{T} \leftarrow \mathsf{F}\mathsf{T}$ such that for every F-algebra $f : A \leftarrow \mathsf{F}A$ there exists a unique homomorphism $h : A \leftarrow \mathsf{T}$ from in to f, we say that in is an initial algebra. We call this homomorphism the *catamorphism* of f which we will denote by $(\!|\mathsf{F};\ f|\!)$ or $(\!|f|\!)$ if the functor is clear from the context. For $(\!|\mathsf{F};\ f|\!)$ we have the defining property,

$$h = (\!|F;\ f|\!) \quad \equiv \quad h \cdot \mathsf{in} \ = \ f \ \cdot \ Fh$$

It is simple to prove that $\mathsf{id}_T = (\!|\mathsf{in}|\!)$, that $(\!|f|\!) \cdot \mathsf{in} = f \ \cdot \ F(\!|f|\!)$, and that the following fusion law holds,

$$h \cdot (\!|f|\!) = (\!|g|\!) \quad \Leftarrow \quad h \cdot f \ = \ g \ \cdot \ Fh$$

Finally, it can be proved that in is an isomorphism, that is, there exists an arrow α such that $\mathsf{in} \cdot \alpha = \mathsf{id}_{FT}$ and $\alpha \cdot \mathsf{in} = \mathsf{id}_T$. For the proofs of these statements, see Paul Hoogendijk's Ph.D. thesis [6].

For lists we have that the initial algebra is $\mathsf{nil} \triangledown \mathsf{cons}$ and using the above notation for the function h which calculates the sum of a list of natural numbers we have that $h = (\!|0 \triangledown (+)|\!)$.

Since allegories are also categories and relators are also functors, we can take the same definitions for F-algebras, F-homomorphisms, and initial algebras in an allegory. In a power allegory (that is, an allegory with power objects and the power functor), it can be shown that the relational inital algebra coincides with the initial algebra of the subcategory of functions. For a proof see [3].

3.3.3 Parameterized Datatypes

Above we saw how to define a type *ListA* which represented lists over the type A. As we would like to be able to make lists over an arbitrary datatype, and not just the type A, we would like to define a functor List which takes the object A as a parameter. To do this we need an object *ListA* for every object A and thus we need an initial algebra $(\mathsf{nil} \triangledown \mathsf{cons})_A$ for every object A. If we define $A \otimes X \ := \ K_1 X + A \times X$ then we expect there to be an initial algebra $\mathsf{in}_A : \mathit{ListA} \leftarrow (A\otimes)\mathit{ListA}$ for each object A.

To define the functor List we want to be able to map a function $f : A \leftarrow B$ on a $\mathsf{List}B$ structure to obtain a $\mathsf{List}A$ structure. In a pointwise fashion, we can define List as follows; for the empty list we have,

$$\mathsf{List}f\ .\ []_B = []_A$$

and for the non-empty list we have,

$$\mathsf{List}f\ .\ (\mathsf{cons}_B.(b,x)) = \mathsf{cons}_A.(f.b\ ,\ \mathsf{List}f\,.\,x) = (\mathsf{cons}_A\ \cdot\ f \times \mathsf{id}_{\mathsf{List}A}).(b\ ,\ \mathsf{List}f\,.\,x)$$

Writing this in a point-free way, we get,

$$\mathsf{List}f\ \cdot\ (\mathsf{nil} \triangledown \mathsf{cons})_B = (\mathsf{nil}_A \triangledown (\mathsf{cons}_A\ \cdot\ f \times \mathsf{id}_{\mathsf{List}A})) \cdot (1 + (\mathsf{id}_A \times \mathsf{List}f))$$

With a little bit of calculation, we can rewrite this as follows,

$$\begin{array}{ll}
 & \mathsf{nil}_A \triangledown (\mathsf{cons}_A\ \cdot\ f \times \mathsf{id}_{\mathsf{List}A}) \\
= & \qquad \{\ \text{identity}\ \} \\
 & (\mathsf{nil}_A \cdot \mathsf{id}_1) \triangledown (\mathsf{cons}_A\ \cdot\ f \times \mathsf{id}_{\mathsf{List}A}) \\
= & \qquad \{\ \text{junc-coproduct fusion}\ \} \\
 & (\mathsf{nil} \triangledown \mathsf{cons})_A \cdot (\mathsf{id}_1 + (f \times \mathsf{id}_{\mathsf{List}A})) \\
= & \qquad \{\ \text{definition } \mathsf{in}_A \text{ and } \otimes\ \} \\
 & \mathsf{in}_A \cdot (f \otimes \mathsf{id}_{\mathsf{List}A})
\end{array}$$

Thus we have that $\mathsf{List}f = (\!|B\otimes;\ \mathsf{in}_A \cdot (f \otimes \mathsf{id}_{\mathsf{List}A})|\!)$.

Generalizing this to an arbitrary recursive datatype, we want to define a functor T such that for every type A we have an initial algebra $\mathsf{in}_A : TA \leftarrow (A\otimes)TA$ of the functor $(A\otimes)$. Replacing List in the above example with T we define,

$$\mathsf{T}f \;:=\; (\!|B\otimes;\ \mathsf{in}_A \cdot (f\otimes \mathsf{id}_{TA})|\!)$$

T will be called the *tree type functor induced by* $\otimes$. That T is indeed a functor will not be proved here. The interested reader can see [6] for the proof. Furthermore, we can generalize the above result to relators by keeping the same definition for T and using the relational catamorphism and initial algebras. Again the proofs can be found in [6].

Chapter 4

Generic Programs

In this chapter we give a formal definition of a polymorphic program in terms of a "healthiness" condition called a natural transformation. Using this formalization gives a precise, mathematical way of verifying whether or not a program is truely generic.

Furthermore, we will define a generic membership relation, which is the largest natural transformation of a certain type. We can use this generic membership to extend the notion of a generic datatype and at the same time, to give a motivation for the interpretation of a natural transformation.

4.1 Natural Transformations

Natural transformations are a mathematical formalization of polymorphic programs which convert one datatype to another without changing the values of the elements contained in the data structures themselves. Thus a natural transformation might convert a tree of integers to a list of integers by flattening it, and a pair of values can be converted to a single value with the projection arrows. The important part of a natural transformation is that it behaves uniformly over all types, and that it does no arithmetic on the values contained in a structure but only copies them.

First we will define natural transformations for categories and functors. Then we will extend this concept to allegories and relators, using tabulations to show that the chosen extension is canonical in some way.

4.1.1 Definition

Given two functors $\mathsf{F} : \mathcal{C} \leftarrow \mathcal{D}$ and $\mathsf{G} : \mathcal{C} \leftarrow \mathcal{D}$, then a mapping α to arrows in $\mathcal{C}$ from objects in $\mathcal{D}$ is a natural transformation of type $\mathsf{F} \leftarrow \mathsf{G}$ if and only if for all arrows $f : A \leftarrow B$ in C we have,

$$\mathsf{F}f \cdot \alpha_B = \alpha_A \cdot \mathsf{G}f$$

The equation says that it should not make a difference whether we first map an arrow f on our original $\mathsf{G}B$-structure and then transform the resulting $\mathsf{G}A$-structure to an $\mathsf{F}A$-structure using the arrow α_A, or first transform the $\mathsf{G}B$-structure to an $\mathsf{F}B$-structure using the arrow α_B and then map f on the resulting $\mathsf{F}B$-structure. In other words, the different component arrows α must behave uniformly over types. Thus this condition is a healthiness condition for generic programs.

To extend a natural transformation to relations we can simply use the same defining property for relators and relations, i.e., we can say that a mapping α to relations in allegory $\mathcal{C}$ from objects in an allegory $\mathcal{D}$ is a natural transformation of type $\mathsf{F} \leftarrow \mathsf{G}$ for relators $\mathsf{F} : \mathcal{C} \leftarrow \mathcal{D}$ and $\mathsf{G} : \mathcal{C} \leftarrow \mathcal{D}$ if for all relations $R : A \leftarrow B$ in C we have,

$$\mathsf{F}R \cdot \alpha_B = \alpha_A \cdot \mathsf{G}R$$

This is not the natural definition of a natural transformation on relations however, since the equality is too strong for relations. Consider for example the natural transformation on functions $\mathsf{id}\vartriangle\mathsf{id} : \mathsf{F} \leftarrow \mathsf{Id}$, where $\mathsf{F}X = X \times X$. To verify that it is a natural transformation on functions one need only verify the trivial property that,

$$f \times f \cdot (\mathsf{id}\vartriangle\mathsf{id}) = (\mathsf{id}\vartriangle\mathsf{id}) \cdot f$$

However, if we replace the function f by a relation which is not simple, i.e., a non-deterministic relation R, the equality no longer holds. Note that $\mathsf{id}\vartriangle\mathsf{id}$ takes a value a and doubles it to create the pair (a,a). If we first map R on the value a to obtain a value b or c, and then apply the natural transformation $\mathsf{id}\vartriangle\mathsf{id}$ we get either the pair (b,b) or (c,c). However, if we first double the value to get (a,a) and then map $R \times R$ on the resulting pair, we get the pairs (b,c) and (c,b) as well. Thus the left hand side of the above expression is larger.

Now consider the natural transformation $\mathsf{outl} : \mathsf{Id} \leftarrow \mathsf{F}$, where again we define $\mathsf{F}X = X \times X$. In this case the naturality property of functions says that,

$$f \cdot \mathsf{outl} = \mathsf{outl} \cdot f \times f$$

which is easily verified. However, if we now replace the function f by a partial relation R which maps a to c but does not map b to anything, then the equality fails to hold also. Since outl takes a pair (a,b) to the value a we have that if we first apply $R \times R$ to the pair (a,b) we get no result, since R is not defined on b, and applying outl to no result leaves us with no result. However, if we first apply outl to the pair (a,b) we get the value a, and since R *is* defined on a we get the value c as a result. Thus the left hand side of the above expression is again larger than the right hand side.

It is the fact that a natural transformation can lose or duplicate information which causes the inequality, and thus we define the notion of a lax natural transformation. A mapping α is a lax natural transformation to F from G, denoted $\alpha : \mathsf{F} \hookleftarrow \mathsf{G}$, if for all relations $R : A \leftarrow B$ we have,

$$\mathsf{F}R \cdot \alpha_B \supseteq \alpha_A \cdot \mathsf{G}R$$

In a similar manner we can define $\alpha : \mathsf{F} \hookrightarrow \mathsf{G}$ if,

$$\mathsf{F}R \cdot \alpha_B \subseteq \alpha_A \cdot \mathsf{G}R$$

which is called a conatural transformation. Note that we have the equivalences $\alpha : \mathsf{F} \hookleftarrow \mathsf{G} \equiv \alpha^\circ : \mathsf{G} \hookrightarrow \mathsf{F}$ and $\alpha : \mathsf{F} \leftarrow \mathsf{G} \equiv \alpha : \mathsf{F} \hookleftarrow \mathsf{G} \wedge \alpha^\circ : \mathsf{F} \hookrightarrow \mathsf{G}$, the proof of which will be left to the reader. Although it is easy to reconstruct the defining property of a natural transformation from this notation, it is not really a good notation since $\alpha : \mathsf{F} \hookleftarrow \mathsf{G}$ is not equivalent to $\alpha : \mathsf{G} \hookrightarrow \mathsf{F}$.

4.1.2 The Link Between Categories and Allegories

As we have seen, if α is a natural transformation on functions, then α is in general not a natural transformation with equality on relations, and this suggests that we take a lax natural transformation to be the relational extension of a natural transformation on functions. To prove that a lax natural transformation is an extension of a functional natural transformation, we first have to show that for functions equality holds. Thus let f be a function, and $\alpha : \mathsf{F} \hookleftarrow \mathsf{G}$, then we only have to prove the following inclusion,

$$
\begin{aligned}
& \mathsf{Ff} \cdot \alpha_B \subseteq \alpha_A \cdot \mathsf{Gf} \\
\equiv \quad & \quad \{ \text{ shunting of functions } \} \\
& \alpha_B \cdot \mathsf{Gf}^\circ \subseteq \mathsf{Ff}^\circ \cdot \alpha_A \\
\equiv \quad & \quad \{ \ \alpha : \mathsf{F} \hookleftarrow \mathsf{G} \ \} \\
& \mathit{true}
\end{aligned}
$$

If on the other hand, we know that α is a natural transformation with equality on functions, then assuming tabulations we can also prove that it is a lax natural transformation on relations. To do this, let (f,g) be a tabulation of R, then,

$$
\begin{aligned}
& \mathsf{FR} \cdot \alpha_B \supseteq \alpha_A \cdot \mathsf{GR} \\
\equiv \quad & \quad \{ \text{ tabulations } \} \\
& \mathsf{Ff} \cdot \mathsf{Fg}^\circ \cdot \alpha_B \supseteq \alpha_A \cdot \mathsf{Gf} \cdot \mathsf{Gg}^\circ \\
\equiv \quad & \quad \{ \ \alpha : \mathsf{F} \leftarrow \mathsf{G} \text{ for functions } \} \\
& \mathsf{Ff} \cdot \mathsf{Fg}^\circ \cdot \alpha_B \supseteq \mathsf{Ff} \cdot \alpha_A \cdot \mathsf{Gg}^\circ \\
\Leftarrow \quad & \quad \{ \text{ monotonicity } \} \\
& \mathsf{Fg}^\circ \cdot \alpha_B \supseteq \alpha_A \cdot \mathsf{Gg}^\circ \\
\equiv \quad & \quad \{ \text{ shunting of functions } \} \\
& \alpha_B \cdot \mathsf{Gg} \supseteq \mathsf{Fg} \cdot \alpha_A \\
\Leftarrow \quad & \quad \{ \ \alpha : \mathsf{F} \leftarrow \mathsf{G} \text{ for functions } \} \\
& \mathit{true}
\end{aligned}
$$

Since the relators F and G are functors on functions, we have that $\alpha : \mathsf{F} \hookleftarrow \mathsf{G}$ in an allegory $\mathcal{C}$ if and only if $\alpha : \mathsf{F} \leftarrow \mathsf{G}$ in the subcategory of functions $\mathit{Map}(\mathcal{C})$. Thus the concept of a lax transformation is the canonical extension of a natural transformation in a category.

4.1.3 Interpretation

The interpretation of a natural transformation $\alpha : \mathsf{F} \hookleftarrow \mathsf{G}$ is that it transforms a G-structure into an F-structure without doing any computation on the values of the elements. That is, α can throw away or duplicate values, but it cannot invent new values. This claim is motivated using the membership relation in section 4.2. However, this does not fully characterize a natural transformation. As we will see in chapter 5 on positions, there exist mappings from an object to arrows which satisfy the condition that they do not invent values, but that are not natural transformations.

In fact, a natural transformation does no computation on the values at all, and it can not examine the elements to alter its behavior. A natural transformation may only select elements from the F based on where they are located and nothing else. This characterization of natural transformations can be formulated using positions, which we will discuss in chapter 5.

4.1.4 The Natural Transformations $\mathsf{H}\alpha$ and α_{H}

We will discuss two important mappings from natural transformations to natural transformations. Given a natural transformation $\alpha : \mathsf{F} \leftarrow \mathsf{G}$ and a functor H we can define $\mathsf{H}\alpha$ to be a mapping from arrows to objects as follows, for each object A,

$$(\mathsf{H}\alpha)_A := \mathsf{H}(\alpha_A)$$

It is easy to verify that $\mathsf{H}\alpha$ defined in this way is a natural transformation of type $\mathsf{HF} \leftarrow \mathsf{HG}$. Similarly we can define the mapping α_H from arrows to objects as follows, for each object A,

$$(\alpha_\mathsf{H})_A := \alpha_{\mathsf{H}A}$$

Again this defines a natural transformation α_H, and this one has the type $\mathsf{FH} \leftarrow \mathsf{GH}$.

The interpretation of $\mathsf{H}\alpha$ is that it takes an HG structure and applies α on every G-structure it contains to obtain an HF structure. Similarly, α_H represents the natural transformation which takes a GH-structure and simply applies α on the outer structure to get an FH-structure.

4.1.5 The Category and Allegory of Natural Transformations

Given a natural transformation $\alpha : \mathsf{F} \leftarrow \mathsf{G}$ and a natural transformation $\beta : \mathsf{G} \leftarrow \mathsf{H}$ for relators F, G and H, we can define a natural transformation $\alpha \cdot \beta : \mathsf{F} \leftarrow \mathsf{H}$ by $(\alpha \cdot \beta)_A := \alpha_A \cdot \beta_A$. Furthermore, we have that $\mathsf{id}_\mathsf{F} : \mathsf{F} \leftarrow \mathsf{F}$ for any relator F. Using these definitions, natural transformations form a category, allowing us to combine natural transformations to make new natural transformations.

Just as we can lift the definition of composition from the base category of a natural transformation, we can lift the partial order, and the reverse structure as well. That is, we define,

$$\alpha \subseteq \beta \equiv \forall(A :: \alpha_A \subseteq \beta_A)$$

and all the other operations on a pointwise basis. All these basic allegorical operations are well defined on natural transformations, i.e. they preserve naturality, except for the meet operation $\cap$. For example, consider the two natural transformations $\mathsf{swap}_A : A \times A \leftarrow A \times A$ which swaps elements in a pair, ie., $\mathsf{swap}_A.(a,b) = (b,a)$, and $(\mathsf{id} \times \mathsf{id})_A : A \times A \leftarrow A \times A$. Then $\mathsf{id} \times \mathsf{id} \cap \mathsf{swap}$ is the partial identity on $A \times A$ which returns a pair if and only if the two elements it contains are equal. It is easy to verify that this relation is not a natural transformation, since there exists no image of $(0 \times 0 \cdot (\mathsf{id} \times \mathsf{id} \cap \mathsf{swap})).(1,2)$ as $1 \neq 2$, but there does exist an image of $((\mathsf{id} \times \mathsf{id} \cap \mathsf{swap}) \cdot 0 \times 0).(1,2)$, namely the pair $(0,0)$. Thus we do not have the inclusion

$$f \times f \cdot (\mathsf{id} \times \mathsf{id} \cap \mathsf{swap}) \supseteq (\mathsf{id} \times \mathsf{id} \cap \mathsf{swap}) \cdot f \times f$$

and thus $(\mathsf{id} \times \mathsf{id} \cap \mathsf{swap})$ is not natural.

4.2 Membership

In chapter 3 we introduced the notion of a relator to characterize datatypes. A relator consisted of a datatype former and a polymorphic map function, but it gave us no way to examine the elements contained in a data structure. To that end we now define a generic notion of membership, as described by Paul Hogendijk in [6].

4.2.1 Definition

Given a partial identity $X : A \leftarrow A$ which represents a subset of the elements of A and a relator F, then $\mathsf{F}X$ is the partial identity representing the subset of $\mathsf{F}A$-structures which contain elements from X. Thus, for a generic membership relation $\mathsf{mem}_A : A \leftarrow \mathsf{F}A$, we want the range of $\mathsf{mem}_A \cdot \mathsf{F}X$ to be contained in X. This can be expressed as follows,

$$(\mathsf{mem}_A \cdot \mathsf{F}X)< \subseteq X$$

Furthermore, if $\mathsf{Y} : \mathsf{FA} \leftarrow \mathsf{FA}$ is a partial identity representing a subset of the collection of FA-structures for which the membership relation only returns elements of X, then we should have that Y is no larger than FX. Thus we have,

$$(\mathsf{mem}_\mathsf{A} \cdot \mathsf{Y})_< \subseteq \mathsf{X} \Rightarrow \mathsf{Y} \subseteq \mathsf{FX}$$

Using monotonicity of composition and converse, the first equation can be rewritten as

$$(\mathsf{mem}_\mathsf{A} \cdot \mathsf{Y})_< \subseteq \mathsf{X} \Leftarrow \mathsf{Y} \subseteq \mathsf{FX}$$

and thus we have the following equivalence,

$$(\mathsf{mem}_\mathsf{A} \cdot \mathsf{Y})_< \subseteq \mathsf{X} \equiv \mathsf{Y} \subseteq \mathsf{FX}$$

As Paul Hoogendijk shows in his Ph. D. thesis [6], this expression can be generalized to the more useful property that for every relation R we have,

$$\mathsf{FR} \cdot \mathsf{mem}_\mathsf{A} \backslash \mathsf{id}_\mathsf{A} = \mathsf{mem}_\mathsf{B} \backslash \mathsf{R}$$

This property is taken as the defining property of membership, and it implies the previous equivalence.

4.2.2 Properties

In his thesis [6], Paul Hoogendijk shows the the following two important properties of a collection of arrows mem defined as above. First of all, the above equation defines the collection of arrows mem in a unique way and thus it is a good definition of membership, and secondly, the collection of arrows mem is a natural transformation of type $\mathsf{Id} \hookleftarrow \mathsf{F}$, and it is in fact the largest natural transformation of its type. To prove that membership is a natural transformation we reason as follows,

$$\begin{array}{ll}
 & \mathsf{R} \cdot \mathsf{mem}_\mathsf{A} \supseteq \mathsf{mem}_\mathsf{B} \cdot \mathsf{FR} \\
\equiv & \qquad \{ \text{ factors } \} \\
 & \mathsf{mem}_\mathsf{B} \backslash (\mathsf{R} \cdot \mathsf{mem}_\mathsf{A}) \supseteq \mathsf{FR} \\
\equiv & \qquad \{ \text{ membership } \} \\
 & \mathsf{FR} \cdot \mathsf{mem}_\mathsf{A} \backslash \mathsf{mem}_\mathsf{A} \supseteq \mathsf{FR} \\
\equiv & \qquad \{ \text{ factors } \} \\
 & \mathit{true}
\end{array}$$

To prove that membership is the largest natural transformation of its type we need the axiom that $\mathsf{id} : \mathsf{Id} \hookleftarrow \mathsf{Id}$ is the largest natural transformation of its type (this "identification axiom" follows from extensionality and holds in Rel; see [6]). First we show that $\mathsf{mem} \backslash \mathsf{id}$ is a natural transformation of type $\mathsf{F} \hookleftarrow \mathsf{Id}$,

$$\begin{array}{ll}
 & \mathsf{mem}_\mathsf{A} \backslash \mathsf{id}_\mathsf{A} \cdot \mathsf{R} \\
\subseteq & \qquad \{ \text{ factors } \} \\
 & \mathsf{mem}_\mathsf{B} \backslash \mathsf{R} \\
= & \qquad \{ \text{ membership } \} \\
 & \mathsf{FR} \cdot \mathsf{mem}_\mathsf{A} \backslash \mathsf{id}_\mathsf{A}
\end{array}$$

Now we can show that membership is the largest natural transformation of its type, for let $\alpha : \mathsf{Id} \hookleftarrow \mathsf{F}$, then

$$
\begin{array}{ll}
& \alpha \\
\subseteq & \quad \{ \text{ factors } \} \\
& \alpha \cdot \mathsf{mem} \backslash \mathsf{mem} \\
= & \quad \{ \text{ membership } \} \\
& \alpha \cdot \mathsf{Fmem} \cdot (\mathsf{mem} \backslash \mathsf{id})_{\mathsf{F}} \\
\subseteq & \quad \{ \ \alpha : \mathsf{Id} \hookleftarrow \mathsf{F} \ \} \\
& \mathsf{mem} \cdot \alpha_{\mathsf{F}} \cdot (\mathsf{mem} \backslash \mathsf{id})_{\mathsf{F}} \\
\subseteq & \quad \{ \ \alpha \cdot \mathsf{mem} \backslash \mathsf{id} : \mathsf{Id} \hookleftarrow \mathsf{Id}, \text{ thus } \alpha \cdot \mathsf{mem} \backslash \mathsf{id} \subseteq \mathsf{id} \text{ (identification axiom) } \} \\
& \mathsf{mem}
\end{array}
$$

And thus every natural transformation of type $\mathsf{Id} \hookleftarrow \mathsf{F}$ is contained in mem and mem is the unique solution of the defining equation of membership.

There are two other important properties of membership that we will mention but will not prove here. The reader can refer to [6] for a full discussion. First of all, the membership of the composition FG of two relators F and G is given by

$$\mathsf{mem}(\mathsf{FG}) \ = \ \mathsf{mem}(\mathsf{G}) \cdot (\mathsf{mem}(\mathsf{F}))_{\mathsf{G}}$$

Thus one takes a member of an FG-structure by first taking a G-structure out of the outer F-structure, and then taking an element of the remaining G-structure. Secondly, using this property one can prove that $\mathsf{mem}(\mathsf{F}) \backslash \mathsf{mem}(\mathsf{G})$ is the largest natural transformation of the type $\mathsf{F} \hookleftarrow \mathsf{G}$, which motivates the interpretation of membership given in section 4.1.3.

Futhermore, for a relator which takes more than one argument, we define the membership as a vector with the same length as the arity of the relator, where each component of the vector is the membership relation of the corresponding argument. For example, for the product relator $\times$ we define

$$\mathsf{mem}(\times) = (\mathsf{mem}(\mathsf{id}\times) \ , \mathsf{mem}(\times\mathsf{id}))$$

4.2.3 The Standard Datatypes

In this section we will list the membership relators for all the standard datatypes. For the proofs that they do indeed satisfy the defining property of membership see [6].

For the identity relator Id we have the axiom that $\mathsf{id} : \mathsf{Id} \hookleftarrow \mathsf{Id}$ is the largest natural transformation of its type, and thus its membership relation is id. Since the constant relator K_{A} represents the datatype with no elements, we expect $\perp\!\!\!\perp$ to be the membership relation, which is indeed the case. For the product relator, which is of arity 2, we should have a vector of length two, the first of which returns the members of the first argument and the second of which returns the members of the second argument. Thus we expect the membership relation to be the vector $(\mathsf{outl},\mathsf{outr})$ which is also the case. With a similar line of reasoning one gets that the membership relation of the coproduct relator is $(\mathsf{inl}^\circ \ , \mathsf{inr}^\circ)$. For the power relator we get the expected property that its membership relation is the relation $\in$.

Finally, for a recursive datatype T with initial algebra in induced by a binary relator $\otimes$ we can define the membership relation as $\mathsf{mem} \ := \ root \cdot branch^*$ given that the membership relation of $\otimes$ is the vector $(\mathsf{meml},\mathsf{memr})$ and $root \ := \ \mathsf{meml} \cdot \mathsf{in}^\circ$ and $branch \ := \ \mathsf{memr} \cdot \mathsf{in}^\circ$. Note that we have,

$$root \cdot branch^* = root \cup (root \cdot branch^*) \cdot branch = root \cup \mathsf{mem} \cdot branch$$

The interpretation of this is clear; to get a member of a tree type we either take a root element or we select a member from a branch.

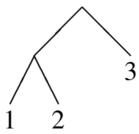
Chapter 5

Positions

In the description of generic datatypes, we have started with the most generic definition using functors in a category, then extended this definition to relations in an allegory using relators, and finally, we required a datatype to consist of a relator with membership. The final definition, which is the least generic but also the most useful, makes it possible to take a data structure apart and look at the individual elements contained in it.

Even the notion of a relator combined with membership is still not enough to completely describe the standard datatypes such as pairs, lists, trees, and sets. There is another generic notion which belongs to datatypes, strongly related to membership, namely the notion of positions. While membership makes it possible to examine individual elements in a data structure, one does not know where in the data structure the element came from. Positions make this possible.

Consider the pair $(3,\text{'a'})$ of an integer and a character. It is very natural to say that the 3 is contained in the first or left position of the pair, and the 'a' is in the second or right position. In fact, the names of the two projection funtions on pairs outl and outr which return the left and the right elements of the pair already give as much away. Similarly, for a list of arbitrary values of an arbitrary type, say $[a,b,c]$, we can say that element a is in the first position, element b is in the second position, and element c is in the third position. For binary leaf trees, it becomes a bit harder to give a position a name, but in the following tree



we clearly have, for example, that 2 is contained in "the right leaf of the left branch". Finally, for a set it is more difficult to define positions, since the sets $\{1,2\}$ and $\{2,1\}$ are identical, but as we will see, it will make sense to define a single position for sets which contains multiple elements, in this case both 1 and 2.

Just as membership returns a value, non-deterministically chosen from an entire data structure, we will identify a position with the relation that returns the elements at that position in a data structure.

5.1 Motivation

Relators are a very weak characterization of datatypes in general. While they do allow us to *form* datatypes, they do not allow us to take them apart again. For all the standard datatypes, the methods of taking them apart are defined in an ad hoc manner. For example, for the product and coproduct relators, we require the existence of the projections and the injections. The power relator is defined in terms of its membership relation. In the case of tree types, the decomposition aspect resides in the binary relator $\otimes$.

Relators with membership are still a weak characterization of datatypes in general. Although we can examine the elements in a data structure using membership, we still cannot tell where the elements came from. However, all standard datatypes do have a means to do this. For the product relator, for example, we have the projections outl and outr to inspect the left and right value of a pair respectively. Similarly, using inl° and inr° we can see if the element in a disjoint sum comes from the first type or the second type. Finally, using the membership relation of sets, we already know where the element came from, since there is only one place it can hide.

Positions embody a logical extension to the characterization of generic datatypes. Not only do they allow us to take apart a data structure to investigate the elements, they also give us a way of seeing where each element came from. Futhermore, we expect the union of all the positions to return all the values in a datatype, so that we get membership for free. Having a datatype with membership but without positions is not logical because there would be no operational way to distinguish two different data structures with the same elements.

Positions allow us to do even more. The shape of a data structure and the elements it contains at each position uniquely identify it. Thus we can separate a data structure into the two concepts of shape and position, as C. B. Jay does in [7] and [8]. C. B. Jay makes use of lists as an intermediate type to store the members of a data structure and a natural transformation which fills a shape given a list of values. Using positions one has a generic theory of contents and shape, without the intermediate datatype of lists, and furthermore, which can be defined for a larger class of datatypes.

Another advantage of positions is that they give a conceptual unification of many properties of datatypes. For example, the definitions of split and junc and the existential image functor defined by $\mathsf{E}R \;:=\; \Lambda(R \cdot \in)$, are in fact instances of one generic concept for datatypes with positions. Similarly, the product-split fusion rule given by,

$$R{\times}S \;\cdot\; (T{\vartriangle}U) \;=\; (R \cdot T){\vartriangle}(S \cdot U)$$

and the following fusion rule of the powerset relator P, which is actually the defining property of its membership relation,

$$\mathsf{P}R \;\cdot\; \in\backslash S \;=\; \in\backslash(R \cdot S)$$

are actually instances of a generalized rule which can be expressed in terms of positions.

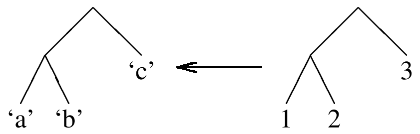
We know that $\mathsf{mem}(\mathsf{F}) \setminus \mathsf{mem}(\mathsf{G})$ is the largest natural transformation of type $\mathsf{F}\hookleftarrow\mathsf{G}$, which shows that a natural transformation cannot invent new values. That is, given a G-structure x and an F-structure y such that $y\langle\alpha\rangle x$ for some natural transformation $\alpha : \mathsf{F}\hookleftarrow\mathsf{G}$ we know that the members in y must all be contained in x. However, there are other relations that satisfy this condition that are not natural transformations. For example, consider the *filter* function which takes a list of values and filters out all occurrences of the first element (if it exists). Thus is, $\mathit{filter}.[1,1,6] = [6]$ and $\mathit{filter}.[\text{‘a’},\text{‘a’},\text{‘a’}] = []$. Now consider the function *littleA* which maps lists of integers to lists of little letters ‘a’ of the same length, then we have $(\mathit{littleA} \cdot \mathit{filter}).[1,1,6] = [\text{‘a’}]$ and $(\mathit{filter} \cdot \mathit{littleA}).[1,1,6] = []$. Thus $\mathit{littleA} \cdot \mathit{filter}$ and $\mathit{filter} \cdot \mathit{littleA}$ are not equal and thus *filter* is not a natural transformation, even though it does not invent values. In other words, the property that natural transformations do not invent values does not characterize them completely. Using positions however, it should be possible to give a more complete characterization. In fact, a natural transformations can be defined on a shape-wise basis as a relation which non-deterministically selects elements from a group of positions in the source structure for each position in the target structure.

5.2 Definition

We will consider the nature of positions, based on an intuitive feeling for how they should behave, and try to derive a useful definition. The definition should not be too specific, nor should it be too general, and it should lend itself well to calculations. Furthermore, from the set of requirements that we will formulate, we must choose a small and consistent subset to characterize positions, and we should be able to derive the other requirements and all other desired properties of positions from this basis.

5.2.1 Naturality

Since we identify a position with the relation which returns the elements at that position, a position is a relation of type $A \leftarrow \mathsf{F}A$ for a given relator F and an object A, which is the type of the values contained in the F-structure. Just as the shape of a data structure is preserved when mapping a function on a data structure using the arrow map of the relator, we expect positions to be preserved under the arrow map of the relator. For example, if we map a function which maps n to the nth letter of the alphabet on the above tree, we expect the following result,



Here we see that the character 'a' is in the same position as the 1 originally was. Thus, for any function f, it should not matter if we first map f and then take a position, or first take a position and then map f. In a formula, let ρ be a position, then

$$f \cdot \rho \;=\; \rho \cdot \mathsf{F}f$$

This is the condition for a natural transformation. Note that if we take a relation R instead of a function f the equality must be replaced by the inclusion $\supseteq$ and we have that positions are lax natural transformations of the type $\mathsf{Id} \hookleftarrow \mathsf{F}$.

The requirement that a position is a lax natural transformation of type $\mathsf{Id} \hookleftarrow \mathsf{F}$ is quite obvious. Positions convert an F-structure to an Id-structure, by returning an element of the F-structure but without changing it. Futhermore, since the Id-structure contains only one value, and an F-structure, in general, contains many values, positions cause a loss of information.

Since membership is the largest natural transformation of type $\mathsf{Id} \hookleftarrow \mathsf{F}$ we have that all positions are contained in membership, that is, for all positions ρ we have that $\rho \subseteq \mathsf{mem}$.

5.2.2 Minimality

The requirement that positions are natural transformations of type $\mathsf{Id} \hookleftarrow \mathsf{F}$ is not enough to characterize them. For instance, membership satisfies this requirement, and so does any union of a collection of what we intuitively see as positions. For example, the union of the first and second positions of a list, which non-deterministically returns either the first or second element, satisfies the naturality requirement, but clearly we do not want to consider it as a position. In fact we want positions to be 'atomic' or 'minimal' in some sense.

A possibility would be to actually require atomicity. That is, we could require a position to be an atomic natural transformation of type $\mathsf{Id} \hookleftarrow \mathsf{F}$. Formally, a natural transformation $\alpha : \mathsf{F} \hookleftarrow \mathsf{G}$ is *atomic* if $\alpha \neq \perp\!\!\perp$ and for all natural transformations $\beta : \mathsf{F} \hookleftarrow \mathsf{G}$ we have,

$$\beta \subseteq \alpha \;\equiv\; \beta = \perp\!\!\perp \vee \beta = \alpha$$

That is, the only natural transformations contained in an atomic natural transformation are itself and $\perp\!\!\perp$. Note that $\perp\!\!\perp$ is not a position in this case. If it turns out to be more useful to include bottom, we can always remove the requirement $\alpha \neq \perp\!\!\perp$, but for now we assume that all positions do actually describe a place in a data structure where data can be stored.

As it turns out, the requirement that positions are atomic results in a theory which does not lend itself well with calculations. Atoms are not easy to work with in an allegory, let alone atomic natural transformations. Furthermore, the requirement may be too strong, since a minimal natural transformation only works on a single shape. This implies that we cannot have a position 'one' on all nonempty lists, which returns the element contained in the first position.

Without using atomicity it is very hard to formulate the notion of minimality directly, since we do not wish to restrict the size of positions arbitrarily. We will first look at the other requirements, and see if we can use a combination of those to define the notion of minimality that we want to have.

5.2.3 Completeness

While for a datatype defined by a relator F with membership the membership relation is unique, we expect most datatypes to have many positions. Thus not only will we need to deal with requirements of single positions, we must also place restrictions on what can be considered a collection of positions. For a given relator F, let us denote a collection of positions for F by $\mathit{Pos}(\mathsf{F})$, or if the relator is clear from the context, by Pos.

One basic requirement of the collection Pos of position, is that it completely covers every F-structure, that is, for every element in a data structure there should be a position in this collection which returns it. For example, if we have a list [2,3,5] we expect there to exist a position which returns the 2, one which returns the 3, and one that returns the 5. Using Rel as a model, we can formulate this *completeness* requirement in a pointwise form and rewrite it in a point-free form as follows; for all objects A,

$$
\begin{array}{ll}
& \forall(a, f :: \; a\langle \mathsf{mem}_A \rangle f \Rightarrow \exists(\rho : \; \rho \in \mathit{Pos} : \; a\langle \rho_A \rangle f)) \\
\equiv & \qquad \{ \text{ join } \} \\
& \forall(a, f :: \; a\langle \mathsf{mem}_A \rangle f \Rightarrow a\langle \cup(\rho : \; \rho \in \mathit{Pos} : \; \rho_A)\rangle f) \\
\equiv & \qquad \{ \text{ inclusion } \} \\
& \mathsf{mem}_A \subseteq \cup(\rho : \; \rho \in \mathit{Pos} : \; \rho_A) \\
\equiv & \qquad \{ \; \mathsf{mem} \text{ is the largest natural transformation of its type } \} \\
& \mathsf{mem}_A = \cup(\rho : \; \rho \in \mathit{Pos} : \; \rho_A)
\end{array}
$$

Thus the union of all positions in the collection Pos must be the membership relation. This also implies that it satisfies the equation which defines the membership relation, and we have,

$$\mathsf{F}R \cdot \cup(\rho : \; \rho \in \mathit{Pos} : \; \rho) \backslash S \;=\; \cup(\rho : \; \rho \in \mathit{Pos} : \; \rho) \backslash (R \cdot S)$$

Using the fact that $(\backslash S)$ is the lower adjoint of $(S/)$ with the reverse ordering, we can use the distribution property of adjoints over joins to get,

$$\mathsf{F}R \cdot \cap(\rho : \rho \in Pos : \rho\backslash S) = \cap(\rho : \rho \in Pos : \rho\backslash(R \cdot S))$$

Now we see that we have a division operation $\rho\backslash S$ for each ρ, whereas for membership we only had one division operation. For positions we should be able to do better than for membership, and we should be able to strengthen the above equation by allowing S to vary for each position ρ. Thus we can take a vector of relations S indexed by the set of positions, and strengthen the above equation as follows,

$$\mathsf{F}R \cdot \cap(\rho : \rho \in Pos : \rho\backslash S_\rho) = \cap(\rho : \rho \in Pos : \rho\backslash(R \cdot S_\rho))$$

The interpretation of the relation $\cap(\rho : \rho \in Pos : \rho\backslash S_\rho)$ is that it creates an F-structure x of an arbitrary shape from a value a, so that at each position ρ the values in x originate via S_ρ from a. The above requirement says that mapping R on a structure created in this way from a value a is the same as creating a structure x such that at every position ρ the values in x originate via $R \cdot S_\rho$ from a. In fact, the relation $\cap(\rho : \rho \in Pos : \rho\backslash S_\rho)$ is a generalization of the split and junc operations of products and coproducts respectively, and the above condition is a generalization of the product-split and coproduct-junc fusion rules.

Instead of the completeness requirement that the union of all the positions in *Pos* is the membership relation, we will make this stronger equation a requirement for positions. The fact that the union of all positions is membership follows from this stronger requirement by simply instantiating S_ρ to a fixed S for every ρ.

Note that the completeness requirement is not enough since the collection of only the membership relation or a collection of arbitrary unions of what we intuitively see as positions satisfies this requirement.

5.2.4 Disjointness

Another requirement of the entire collection of positions *Pos*, is that positions are pairwise disjoint, that is, we do not want a single element to be returned by more than one position. For example, we do not want two different positions to return the element 2 of the list [3,2,5]. Assume we have two positions α and β, then if every F-structure which has an element in position α shares one of those elements with position β, then the positions are not disjoint. The motivation behind this line of reasoning is that one should always be able to fill different positions in a data structure with different values. Since we want α and β to be disjoint unless they represent the same position, we can formulate the disjointness requirement in Rel as follows, and reason,

$$\begin{array}{ll}
 & \forall(f : f\langle\alpha{>}\rangle f : \exists(e :: e\langle\alpha\rangle f \wedge e\langle\beta\rangle f)) \Rightarrow \alpha{=}\beta \\
\equiv & \quad \{ \text{ meet, composition } \} \\
 & \forall(f : f\langle\alpha{>}\rangle f : f\langle\alpha^\circ \cdot \beta\rangle f) \Rightarrow \alpha{=}\beta \\
\equiv & \quad \{ \text{ domains, calculus } \} \\
 & \forall(f_1, f_2 : f_1\langle\alpha{>}\rangle f_2 : f_1\langle\alpha^\circ \cdot \beta\rangle f_2) \Rightarrow \alpha{=}\beta \\
\equiv & \quad \{ \text{ inclusion } \} \\
 & \alpha{>} \subseteq \alpha^\circ \cdot \beta \Rightarrow \alpha{=}\beta \\
\equiv & \quad \{ \text{ domains } \}
\end{array}$$

$$
\begin{array}{ll}
 & \alpha{>} \ \subseteq\ \alpha^{\circ} \cdot \beta \cap \mathsf{id} \ \Rightarrow\ \alpha{=}\beta \\
\equiv & \qquad \{\ \text{domains}\ \} \\
 & \alpha{>} \subseteq (\alpha{\cap}\beta){>} \Rightarrow \alpha{=}\beta
\end{array}
$$

The inclusion can be strengthened to an equality on account of the monotonicity of the domain operator, and the implication can be strengthened to an equivalence due to the idempotence of meet. Thus we formulate the disjointness requirement as follows,

$$\alpha{=}\beta \ \equiv\ \alpha{>} = (\alpha{\cap}\beta){>}$$

For the collection of positions *Pos* we require that all pairs of positions satisfy this condition. Note that this requirement excludes $\perp\!\!\perp$ from any collection of positions which contains more than one position, for if $\perp\!\!\perp$ and β are positions then,

$$\beta{=}\perp\!\!\perp \ \equiv\ \perp\!\!\perp{>} = (\perp\!\!\perp{\cap}\beta){>} \ \equiv\ \perp\!\!\perp{>} = \perp\!\!\perp{>} \ \equiv\ \mathit{true}$$

5.2.5 Equality of F-structures

We will now consider under what circumstances two F-structures are equal and express this in terms of positions. Intuitively, two F-structures are equal if they have the same shape and at every position they contain exactly the same elements. Note the shape requirement is absolutely necessary. For example, if a datatype has two different constants, that is, data structures without any elements contained in them, then these two constants contain the same information at every position, since they have no positions, but they have two different shapes. Furthermore, if we have a composite datatype, such as a list over a datatype with two constants, the same problem occurs and only by requiring shape as well as the contents of two structures to be the same can we guarantee they are equal. In other words, we want to formalize the idea that an F-structure is completely determined if we know its shape and its contents, i.e. what elements it contains at every position.

As we have already seen, two F-structures have the same shape if they are related via $\mathsf{F}\top\!\!\top$. Futhermore, two F-structures f_1 and f_2 have the same elements at every position if all elements at each position in f_1 are also located in that position in f_2 and vice versa. Looking at the first conjunct only and reasoning from Rel we have,

$$
\begin{array}{ll}
 & \forall(\rho :\ \rho{\in}\mathit{Pos} :\ \forall(e :\ e\langle\rho\rangle f_2 :\ e\langle\rho\rangle f_1)) \\
\equiv & \qquad \{\ \text{factors}\ \} \\
 & \forall(\rho :\ \rho{\in}\mathit{Pos} :\ f_2\langle\rho\backslash\rho\rangle f_1) \\
\equiv & \qquad \{\ \text{meet}\ \} \\
 & f_2\langle\cap(\rho :\ \rho{\in}\mathit{Pos} :\ \rho\backslash\rho)\rangle f_1
\end{array}
$$

Now we can define,

$$\mathit{psub} \ :=\ \cap(\rho :\ \rho{\in}\mathit{Pos} :\ \rho\backslash\rho) \cap \mathsf{F}\top\!\!\top$$

Here *psub* is a position-wise subset relation, that is $f_2\langle\mathit{psub}\rangle f_1$ if f_1 and f_2 have the same shape, and for all positions the set of elements contained in f_2 at that position is a subset of the elements contained in f_1 at that position. What we would like is that *psub* is a partial order on F-structures. That is, *psub* should be a reflexive, anti-symmetric, and transitive relation. That *psub* is reflexive follows from the fact that $\mathsf{id} \subseteq R\backslash R$ for all relations R and that $\mathsf{Fid} \subseteq \mathsf{F}\top\!\!\top$. We can show that *psub* is transitive by proving $\mathit{psub} \cdot \mathit{psub} \subseteq \mathit{psub}$, which can be done as follows,

$$
\begin{array}{ll}
 & psub \cdot psub \\
= & \qquad \{ \text{ definition of } psub \ \} \\
 & (\cap(\rho :\ \rho \in Pos :\ \rho \backslash \rho) \cap \mathsf{F}\top\!\top) \cdot (\cap(\rho :\ \rho \in Pos :\ \rho \backslash \rho) \cap \mathsf{F}\top\!\top) \\
\subseteq & \qquad \{ \text{ distribution of } \cdot \text{ over } \cap \ \} \\
 & \cap(\rho_1, \rho_2 :\ \rho_1 \in Pos \wedge \rho_2 \in Pos :\ \rho_1 \backslash \rho_1 \cdot \rho_2 \backslash \rho_2) \cap \mathsf{F}\top\!\top \\
\subseteq & \qquad \{ \text{ meet, take only conjuncts where } \rho_1 = \rho_2 \ \} \\
 & \cap(\rho :\ \rho \in Pos :\ \rho \backslash \rho \cdot \rho \backslash \rho) \cap \mathsf{F}\top\!\top \\
\subseteq & \qquad \{ \text{ factors } \} \\
 & \cap(\rho :\ \rho \in Pos :\ \rho \backslash \rho) \cap \mathsf{F}\top\!\top \\
= & \qquad \{ \text{ definition of } psub \ \} \\
 & psub
\end{array}
$$

The final requirement that *psub* is anti-symmetric can be formulated as follows,

$$psub \cap psub^{\circ} \ \subseteq\ \mathsf{id}_{\mathsf{F}}$$

Since $\mathsf{id}_{\mathsf{F}} \subseteq psub$ we can replace the inclusion with an equality, and then we have precisely the requirement that two structures are equal if and only if they have the same shape and at every position they contain the same elements. This does not hold for an arbitrary collection of natural transformations of type $\mathsf{Id} \hookleftarrow \mathsf{F}$, but we make it a requirement for a collection of positions *Pos*.

This requirement should imply the requirement that the union of all positions is the membership relation, for if there existed an F-structure which contained an element that was not returned by a position, then by changing the value of that element one could create different F-structures that are related by $psub \cap psub^{\circ}$, violating the above condition.

Note that the equality requirement does *not* imply the minimality or disjointness requirements. Consider for example two lists $[a,b,c]$ and $[d,e,f]$ of length 3, and a collection of three positions on lists of length 3, one of which returns either the first or the second element in the list, one of which returns the second or the third element, and one of which returns either the first or the third element. Then we can reason as follows,

$$
\begin{array}{ll}
 & \{a,b\}=\{d,e\} \wedge \{a,c\}=\{d,f\} \wedge \{b,c\}=\{e,f\} \\
\Rightarrow & \qquad \{ \text{ calculus } \} \\
 & \{a,b\}\cap\{a,c\}=\{d,e\}\cap\{d,f\} \wedge \{a,b\}\cap\{b,c\}=\{d,e\}\cap\{e,f\} \wedge \{a,c\}\cap\{b,c\}=\{d,f\}\cap\{e,f\} \\
\equiv & \qquad \{ \text{ calculus } \} \\
 & \{a\}=\{d\} \wedge \{b\}=\{e\} \wedge \{c\}=\{f\} \\
\equiv & \qquad \{ \text{ sets, lists } \} \\
 & [a,b,c]=[d,e,f]
\end{array}
$$

and thus the equality requirement is satisfied even though the positions are neither minimal nor disjoint.

5.2.6 Relators

The equality requirement can be strengthened. We can express the behavior of a relator in terms of positions and shape. Namely, two F-structures are related to each other via $\mathsf{F}R$ if and only if they have the same shape, and if at every position the values are related to each other by the relation R. Note that every value in the target structure needs to originate via R from an element in the same position in the source structure, and that every value in the source structure needs to have at least one value that originates from it via R in the target structure. These two requirements can be expressed by generalizing the *psub* relation. If we define,

$$\mathit{psub}(R) \;:=\; \cap(\rho : \rho{\in}\mathit{Pos} : \rho\backslash(R\cdot\rho)) \cap \mathsf{F}\top\!\!\top$$

then $\mathit{psub}(R)$ is a position-wise subset relation via R, that is, $f_2\langle \mathit{psub}(R)\rangle f_1$ if f_1 and f_2 have the same shape, and for all positions the set of elements contained in f_2 at that position is a subset of R mapped on the set of elements contained at that position in f_1.

Using the above interpretation and generalizing from the fact that *psub* is a partial order we get that $\mathsf{F}R \subseteq \mathit{psub}(R)$, which can easily be proved using the fact that $\mathsf{id}_\mathsf{F}{\subseteq}\mathit{psub}$:

$$
\begin{array}{ll}
 & \mathsf{F}R \\
\subseteq & \quad \{\ \mathsf{id}_\mathsf{F}{\subseteq}\mathit{psub}\ \} \\
 & \mathit{psub}\cdot\mathsf{F}R \\
= & \quad \{\ \text{definition of } \mathit{psub}\ \} \\
 & (\cap(\rho : \rho{\in}\mathit{Pos} : \rho\backslash\rho)\cap\mathsf{F}\top\!\!\top)\cdot\mathsf{F}R \\
\subseteq & \quad \{\ \text{distribution of } \cdot \text{ over } \cap\ \} \\
 & \cap(\rho : \rho{\in}\mathit{Pos} : \rho\backslash\rho\cdot\mathsf{F}R)\cap\mathsf{F}\top\!\!\top\cdot\mathsf{F}R \\
\subseteq & \quad \{\ \text{factors, relators, top}\ \} \\
 & \cap(\rho : \rho{\in}\mathit{Pos} : \rho\backslash(\rho\cdot\mathsf{F}R))\cap\mathsf{F}\top\!\!\top \\
\subseteq & \quad \{\ \rho{:}\mathsf{Id}{\hookleftarrow}\mathsf{F}\ \} \\
 & \cap(\rho : \rho{\in}\mathit{Pos} : \rho\backslash(R\cdot\rho))\cap\mathsf{F}\top\!\!\top \\
= & \quad \{\ \text{defintion of } \mathit{psub}(R)\ \} \\
 & \mathit{psub}(R)
\end{array}
$$

Futhermore, we also get that $\mathit{psub}(R)\cdot\mathit{psub}(S)\subseteq\mathit{psub}(R\cdot S)$, the proof being a simple generalization of the proof of $\mathit{psub}\cdot\mathit{psub}\subseteq\mathit{psub}$:

$$
\begin{array}{ll}
 & \mathit{psub}(R)\cdot\mathit{psub}(S) \\
= & \quad \{\ \text{definition of } \mathit{psub}\ \} \\
 & (\cap(\rho : \rho{\in}\mathit{Pos} : \rho\backslash(R\cdot\rho))\cap\mathsf{F}\top\!\!\top)\cdot(\cap(\rho : \rho{\in}\mathit{Pos} : \rho\backslash(S\cdot\rho))\cap\mathsf{F}\top\!\!\top) \\
\subseteq & \quad \{\ \text{distribution of } \cdot \text{ over } \cap\ \} \\
 & \cap(\rho_1,\rho_2 : \rho_1{\in}\mathit{Pos}\wedge\rho_2{\in}\mathit{Pos} : \rho_1\backslash(R\cdot\rho_1)\cdot\rho_2\backslash(S\cdot\rho_2))\cap\mathsf{F}\top\!\!\top \\
\subseteq & \quad \{\ \text{meet, take only conjuncts where } \rho_1{=}\rho_2\ \} \\
 & \cap(\rho : \rho{\in}\mathit{Pos} : \rho\backslash(R\cdot\rho)\cdot\rho\backslash(S\cdot\rho))\cap\mathsf{F}\top\!\!\top
\end{array}
$$

$$\begin{array}{ll}
\subseteq & \{ \text{ factors } \} \\
& \cap(\rho : \rho \in Pos : \rho \backslash (\mathsf{R} \cdot \mathsf{S} \cdot \rho)) \cap \mathsf{F}\top\!\top \\
= & \{ \text{ definition of } psub \ \} \\
& psub(\mathsf{R} \cdot \mathsf{S})
\end{array}$$

Finally, generalizing from the requirement that $psub \cap psub^{\circ} \subseteq \mathsf{id}_{\mathsf{F}}$, we get that the condition that $psub(\mathsf{R}) \cap psub(\mathsf{R}^{\circ})^{\circ} \subseteq \mathsf{FR}$. This last condition follows from the fact that $psub \cap psub^{\circ} \subseteq \mathsf{id}_{\mathsf{F}}$ for functions, but not for arbitrary relations (although it should if one assumes tabularity, using a generalized proof of the fact that the product relator distributes over composition with the proof of the fact that the power relator distributes over composition). Thus we will make this a requirement for positions instead of the equality requirement, since by instantiating R with the identity relation we get the equality requirement.

Note that for a partial order one can strengthen the inclusions in the anti-symmetry condition and the transitivity condition to equality. The generalized anti-symmetry condition for $psub(\mathsf{R})$ can be strengthened to an equality using the fact that $\mathsf{FR} \subseteq psub(\mathsf{R})$ and that $\mathsf{FR} = (\mathsf{F}(\mathsf{R}^{\circ})^{\circ})$. With these conditions we immediately have that $\mathsf{FR} \subseteq psub(\mathsf{R}^{\circ})^{\circ}$, and using the universal property of meet we get the required result, namely that $\mathsf{FR} \subseteq psub(\mathsf{R}) \cap psub(\mathsf{R}^{\circ})^{\circ}$. Thus we will take the relator requirement for a collection of positions to be,

$$\mathsf{FR} \ = \ psub(\mathsf{R}) \ \cap \ psub(\mathsf{R}^{\circ})^{\circ}$$

The generalized transitivity condition for $psub(\mathsf{R})$ can be strengthened to an equality as well, by making use of the completeness requirement,

$$\mathsf{FR} \cdot \cap(\rho : \rho \in Pos : \rho \backslash \mathsf{S}_{\rho}) = \cap(\rho : \rho \in Pos : \rho \backslash (\mathsf{R} \cdot \mathsf{S}_{\rho}))$$

Using this requirement we can prove that.,

$$\mathsf{FR} \cdot psub \ = \ psub(\mathsf{R})$$

This can be proved in a straightforward way as follows,

$$\begin{array}{ll}
& \mathsf{FR} \cdot (\cap(\rho :: \rho \backslash \rho) \cap \mathsf{F}\top\!\top) \\
\subseteq & \{ \text{ distribution of } \cdot \text{ over } \cap \ \} \\
& \mathsf{FR} \cdot \cap(\rho :: \rho \backslash \rho) \ \cap \ \mathsf{FR} \cdot \mathsf{F}\top\!\top \\
\subseteq & \{ \text{ completeness with } \mathsf{S}_{\rho} := \rho, \mathsf{R} \subseteq \top\!\top, \text{relators } \} \\
& \cap(\rho :: \rho \backslash (\mathsf{R} \cdot \rho)) \cap \mathsf{F}\top\!\top \\
= & \{ \text{ completeness with } \mathsf{S}_{\rho} := \rho \ \} \\
& \mathsf{FR} \cdot \cap(\rho :: \rho \backslash \rho) \cap \mathsf{F}\top\!\top \\
\subseteq & \{ \text{ modular law } \} \\
& \mathsf{FR} \cdot (\cap(\rho :: \rho \backslash \rho) \ \cap \ \mathsf{FR}^{\circ} \cdot \mathsf{F}\top\!\top) \\
\subseteq & \{ \ \mathsf{R}^{\circ} \subseteq \top\!\top, \text{relators } \} \\
& \mathsf{FR} \cdot (\cap(\rho :: \rho \backslash \rho) \cap \mathsf{F}\top\!\top)
\end{array}$$

Using the fact that F is a relator we can generalize this property as follows,

$$\mathsf{FR} \cdot psub(\mathsf{S}) \ = \ \mathsf{FR} \cdot \mathsf{FS} \cdot psub \ = \ \mathsf{F}(\mathsf{R} \cdot \mathsf{S}) \cdot psub \ = \ psub(\mathsf{R} \cdot \mathsf{S})$$

Now using this generalized property we have that,

$$\begin{array}{ll} & psub(\mathsf{R}\cdot\mathsf{S}) \\ = & \quad \{ \text{ above } \} \\ & \mathsf{FR}\ \cdot\ psub(\mathsf{S}) \\ \subseteq & \quad \{ \ \mathsf{FR} \subseteq psub(\mathsf{R}) \ \} \\ & psub(\mathsf{R})\ \cdot\ psub(\mathsf{S}) \end{array}$$

and thus we have that $psub(\mathsf{R}\cdot\mathsf{S}) = psub(\mathsf{R})\cdot psub(\mathsf{S})$. This property states that *psub* distributes over composition, which is a very useful property to have especially considering that *psub* is an intersection.

Using the property that $\mathsf{FR}\cdot psub = psub(\mathsf{R})$ we can also prove that *psub* is a lax natural transformation to F from F, i.e $psub : \mathsf{F} \hookleftarrow \mathsf{F}$. This can be done as follows,

$$\begin{array}{ll} & psub\ \cdot\ \mathsf{FR} \\ \subseteq & \quad \{ \text{ definition of } psub,\ \mathsf{FR} \subseteq psub(\mathsf{R}) \ \} \\ & psub(\mathsf{id})\ \cdot\ psub(\mathsf{R}) \\ \subseteq & \quad \{ \text{ transitivity of } psub(\mathsf{R}),\text{ identity } \} \\ & psub(\mathsf{R}) \\ = & \quad \{ \text{ above } \} \\ & \mathsf{FR}\ \cdot\ psub \end{array}$$

5.2.7 Compositionality

Another property that one might like positions to have is compositionality, that is, given two relators F and G with their respective collections of positions $Pos(\mathsf{F})$ and $Pos(\mathsf{G})$, we would like to be able to construct the collection of positions $Pos(\mathsf{FG})$ as follows,

$$Pos(\mathsf{FG}) = \cup(\alpha,\beta :\ \alpha \in Pos(\mathsf{F})\ \wedge\ \beta \in Pos(\mathsf{G}) :\ \{\beta\cdot\alpha_{\mathsf{G}}\})$$

For example, given a list of trees, we would like to be able to take positions of this datatype by simply taking a position of the list, and then taking a position of the resulting tree. From a calculational viewpoint, compositionality is a very nice property to have.

Note that compositionality and atomicity are inconsistent with each other, and one cannot assume both at the same time. The reason for this is the restriction that atomic positions only accept a single shape. For example, the shapes of [[3],[2,5]] and [[3],[2]] are clearly different, so no atomic position could return the first element of the first list of both of these structures. However, compositionality would allow precisely that, since both structures are lists of length 2 with a list of length 1 in the first position.

5.2.8 Choosing a Definition

For a collection of positions we definitely want the naturality, completeness, disjointness and relator conditions to hold. We have seen that disjointness does not follow from completeness or the relator conditions. However, for membership naturality follows from its completeness property, and we must still investigate if this is also the case for arbitrary positions. Furthermore, we must also investigate whether the relator condition implies completeness.

The one difficulty in defining positions lies in choosing how to group shapes together. By requiring positions to be atomic one restricts every position to a single shape. However, it is easy to verify that if we take the union of several positions which work on different shapes, then the new collection still satisfies the other requirements of positions. This means that we can make collections of positions that behave rather strangely, for example, we could make a position that returns the first element of lists of length 1 and the sixth element of lists of length 10. Atomicity does guarantee nice behavior in this sense. Another benefit of atomicity is that it gives at most one collection of positions for a given relator, whereas without atomicity, any number of collections satisfy the requirements. Atomicity has one serious defect, however, namely it makes positions very hard to do calculations and proofs with.

The compositionality requirement, on the other hand, makes positions very useful in calculations. For example, if we can prove that the class of polynomial relators is compositional for a given definition of the positions of Id, K_A, $\times$ and $+$, then we immediately have the positions of all relators in this class. In section 5.3 we will suggest a collection of compositional positions for all standard datatypes, including the polynomial relators, the power relator and the tree type relator.

5.3 Positions of the Standard Datatypes

In this section we will suggest a collection of positions for identity, constant, product, coproduct, power and tree type relators. We also conjecture that these positions are compositional, so that we can use these positions as a basis to construct positions for all standard datatypes.

5.3.1 The Identity Relator

For the identity relator Id it seems obvious that there should be one position, namely the natural transformation id, since an $\mathsf{Id}A$-structure only contains one value of type A. Thus we will take $Pos(\mathsf{Id}) = \{\mathsf{id}\}$. Since this collection consists of only one single position we have that the positions are pairwise disjoint. For the completeness requirement we have that,

$$
\begin{array}{ll}
 & \mathsf{Id}R \cdot \cap(\rho : \rho \in Pos(\mathsf{Id}) : \rho\backslash S_\rho) \\
= & \qquad \{ \text{ one point rule, } \mathsf{Id} \text{ relator } \} \\
 & R \cdot \mathsf{id}\backslash S_{\mathsf{id}} \\
= & \qquad \{ \text{ factors } \} \\
 & \mathsf{id}\backslash(R \cdot S_{\mathsf{id}}) \\
= & \qquad \{ \text{ one point rule, } \mathsf{Id} \text{ relator } \} \\
 & \cap(\rho : \rho \in Pos(\mathsf{Id}) : \rho\backslash(R \cdot S_\rho))
\end{array}
$$

The relator requirement can be verified as follows,

$$
\begin{array}{ll}
 & \cap(\rho : \rho \in Pos(\mathsf{Id}) : \rho\backslash(R \cdot \rho)) \cap \cap(\rho : \rho \in Pos(\mathsf{Id}) : (\rho^\circ \cdot R) / \rho^\circ) \cap \mathsf{Id}\top\!\!\top \\
= & \qquad \{ \text{ one point rule, } \mathsf{Id} \text{ relator } \} \\
 & \mathsf{id}\backslash(R \cdot \mathsf{id}) \;\cap\; (\mathsf{id}^\circ \cdot R) / \mathsf{id}^\circ \;\cap\; \top\!\!\top \\
= & \qquad \{ \text{ converse, factors, meet } \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{R} \\
= & \quad \{\ \mathsf{Id} \text{ relator } \} \\
 & \mathsf{Id}\mathsf{R}
\end{array}
$$

Note that if we had required the atomicity of positions, we would not be able to prove that id is a position of the identity relator Id. A counter example would be the identity relator in $\mathcal{C}^2$, because the identity natural transformation $(\mathsf{id},\mathsf{id})$ is not atomic as it properly contains both natural transformations $(\mathsf{id},\perp\!\!\perp)$ and $(\perp\!\!\perp,\mathsf{id})$.

5.3.2 The Constant Relators

For the constant relator $\mathsf{K_A}$ we expect that there are no positions at all since a constant data structure contains no data. Thus we expect $Pos(\mathsf{K_A}) = \emptyset$. Vacuously we have that all pairs of positions are disjoint. For the completeness requirement we get that,

$$
\begin{array}{ll}
 & \mathsf{K_A R} \cdot \cap(\rho :\ \rho \in Pos(\mathsf{K_A}) :\ \rho\backslash \mathsf{S}_\rho) \\
= & \quad \{\ \text{empty domain, } \mathsf{K_A} \text{ relator } \} \\
 & \mathsf{id} \cdot \top\!\!\top \\
= & \quad \{\ \text{identity, empty domain } \} \\
 & \cap(\rho :\ \rho \in Pos(\mathsf{K_A}) :\ \rho\backslash(\mathsf{R}\cdot \mathsf{S}_\rho))
\end{array}
$$

For the relator requirement we have,

$$
\begin{array}{ll}
 & \cap(\rho :\ \rho \in Pos(\mathsf{K_A}) :\ \rho\backslash(\mathsf{R}\cdot\rho)) \cap \cap(\rho :\ \rho \in Pos(\mathsf{K_A}) :\ (\rho^\circ \cdot \mathsf{R}) / \rho^\circ) \cap \mathsf{K_A} \top\!\!\top \\
= & \quad \{\ \text{empty domain, } \mathsf{K_A} \text{ relator } \} \\
 & \top\!\!\top \cap \mathsf{id} \\
= & \quad \{\ \text{meet, } \mathsf{K_A} \text{ relator } \} \\
 & \mathsf{K_A R}
\end{array}
$$

5.3.3 The Product Relator

For the binary product relator $\times$ we have that the membership relation is the vector $(\mathsf{outl},\mathsf{outr})$. Since the membership relation for a single-valued non-endo relator consists of a vector of membership relations for each argument, it seems logical to do the same for positions. Thus, we will take $Pos(\times) = \{(\mathsf{outl},\mathsf{outr})\}$. The disjointness condition is again vacuously satisfied. Since we have that $(\mathsf{outl},\mathsf{outr}) = \mathsf{mem}$ and since we only have one position, completeness is also trivially satisfied. For the relator requirement we have,

$$
\begin{array}{ll}
 & \cap(\rho :\ \rho \in Pos(\times) :\ \cap(\rho\backslash((\mathsf{R},\mathsf{S})\cdot\rho))) \cap \cap(\rho :\ \rho \in Pos(\times) :\ \cap((\rho^\circ \cdot (\mathsf{R},\mathsf{S}))/\rho^\circ)) \cap \top\!\!\top \times \top\!\!\top \\
= & \quad \{\ \text{one point rule } \} \\
 & \cap((\mathsf{outl},\mathsf{outr})\backslash((\mathsf{R},\mathsf{S})\cdot(\mathsf{outl},\mathsf{outr}))) \cap \cap(((\mathsf{outl},\mathsf{outr})^\circ \cdot (\mathsf{R},\mathsf{S}))/(\mathsf{outl},\mathsf{outr})^\circ) \cap \top\!\!\top \times \top\!\!\top \\
= & \quad \{\ \text{vectors } \} \\
 & \mathsf{outl}\backslash(\mathsf{R}\cdot\mathsf{outl}) \cap \mathsf{outr}\backslash(\mathsf{S}\cdot\mathsf{outr}) \cap (\mathsf{outl}^\circ \cdot \mathsf{R}) \ /\ \mathsf{outl}^\circ \cap (\mathsf{outl}^\circ \cdot \mathsf{S}) \ /\ \mathsf{outr}^\circ \cap \top\!\!\top \times \top\!\!\top \\
= & \quad \{\ \text{product relator } \}
\end{array}
$$

$$\mathsf{R}{\times}\mathsf{S} \cap (\mathsf{R}^\circ \times \mathsf{S}^\circ)^\circ \cap \top\!\!\top{\times}\top\!\!\top$$

$$= \qquad \{ \text{ product relator } \}$$

$$\mathsf{R}{\times}\mathsf{S}$$

5.3.4 The Coproduct Relator

For the binary coproduct relator $+$ we have that the membership relation is the vector $(\mathsf{inl}^\circ\,, \mathsf{inr}^\circ)$. For the same reason as with the product relator we will take $Pos(+) = \{(\mathsf{inl}^\circ\,, \mathsf{inr}^\circ)\}$, and the disjointness and completeness conditions are satisfied. For the relator requirement we reason,

$$\cap(\rho:\ \rho{\in}Pos(+):\ \cap(\rho\backslash((\mathsf{R,S})\cdot\rho)))\cap\cap(\rho:\ \rho{\in}Pos(+):\ \cap((\rho^\circ\cdot(\mathsf{R,S}))/\rho^\circ))\cap\top\!\!\top{+}\top\!\!\top$$

$$= \qquad \{ \text{ one point rule } \}$$

$$\cap((\mathsf{inl}^\circ\ , \mathsf{inr}^\circ)\backslash((\mathsf{R,S})\cdot(\mathsf{inl}^\circ\ , \mathsf{inr}^\circ)))\cap\cap(((\mathsf{inl}^\circ\ , \mathsf{inr}^\circ)^\circ\cdot(\mathsf{R,S}))/(\mathsf{inl}^\circ\ , \mathsf{inr}^\circ)^\circ)\cap\top\!\!\top{+}\top\!\!\top$$

$$= \qquad \{ \text{ vectors } \}$$

$$\mathsf{inl}^\circ\backslash(\mathsf{R}\cdot\mathsf{inl}^\circ)\ \cap\ \mathsf{inr}^\circ\backslash(\mathsf{S}\cdot\mathsf{inr}^\circ)\ \cap\ (\mathsf{inl}\cdot\mathsf{R})/\mathsf{inl}\ \cap\ (\mathsf{inr}\cdot\mathsf{S})/\mathsf{inr}\ \cap\ \top\!\!\top{+}\top\!\!\top$$

$$= \qquad \{ \text{ coproduct relator } \}$$

$$\mathsf{R}{+}\mathsf{S} \cap (\mathsf{R}^\circ + \mathsf{S}^\circ)^\circ \cap \top\!\!\top{+}\top\!\!\top$$

$$= \qquad \{ \text{ coproduct relator } \}$$

$$\mathsf{R}{+}\mathsf{S}$$

5.3.5 The Power Relator

For the power relator P we have the membership relation $\in$. Since the order of the elements do not matter in a set, however, we expect to have only one position, namely membership itself. Thus we take $Pos(\mathsf{P}) = \{\in\}$. Again, since we have only one position and it is equal to membership, we have that disjointness and completeness are trivially satisfied. For the relator requirement we have,

$$\cap(\rho:\ \rho \in Pos(\mathsf{P}):\ \rho\backslash(\mathsf{R}\cdot\rho)) \cap \cap(\rho:\ \rho \in Pos(\mathsf{P}):\ (\rho^\circ\ \cdot\ \mathsf{R})\ /\ \rho^\circ) \cap \mathsf{P}\top\!\!\top$$

$$= \qquad \{ \text{ one point rule } \}$$

$$\in\backslash(\mathsf{R}\cdot \in)\ \cap\ (\in^\circ\ \cdot \mathsf{R})\ /\ \in^\circ\ \ \cap\ \mathsf{P}\top\!\!\top$$

$$= \qquad \{ \text{ power relator, meet } \}$$

$$\mathsf{PR}$$

5.3.6 The Tree Type Relator

For the tree type relator it is more difficult to derive a collection of positions. First of all, let us consider the List relator. To get the first position of a list we would simply take the standard *head* function defined on lists. Similarly, to get the element at the second position (if it exists), we would apply $head \cdot tail$ to the list. Doing this for all positions would give us $Pos(\mathsf{List}) = \{head\}\cdot\{tail\}^*$, where composition on collections A and B, is defined by $\mathsf{A}\cdot\mathsf{B}\ :=\ \{\mathsf{R}\cdot\mathsf{S}\ :\ \mathsf{R}{\in}\mathsf{A}\wedge\mathsf{S}{\in}\mathsf{T}\}$ and A^* is the reflexive, transitive closure of composition of the collection A. If we consider binary trees defined by a BTree relator, we can apply the same procedure; the "first" position is the *root* function which returns the element in the root of the tree. Next there are two second level positions, namely the root of the left branch or subtree and the root of the right branch or subtree, given by $root \cdot left$ and $root \cdot right$

respectively. Again repeating this procedure we would get that $Pos(\mathsf{BTree}) = \{root\} \cdot \{left,right\}^*$. This result can be generalized to arbitrary tree type relators.

Let $\otimes$ be a binary relator and (in,T) be its corresponding tree type. Given that $\mathsf{mem}=(\mathsf{meml},\mathsf{memr})$ is the membership relation of the binary relator $\otimes$, we know that,

$$\mathsf{mem}(\mathsf{T}) = \mathsf{root} \cdot \mathsf{branch}^*$$

where

$$\mathsf{root} \ := \ \mathsf{meml} \cdot \mathsf{in}^\circ$$

and

$$\mathsf{branch} \ := \ \mathsf{memr} \cdot \mathsf{in}^\circ$$

If we replace the two membership relations by collections of positions, that is, let *Pos* be the collection of positions of the binary relator $\otimes$ and let $Posl = \{\alpha \ : \ (\alpha,\beta) \in Pos\}$ and $Posr = \{\beta \ : \ (\alpha,\beta) \in Pos\}$, then we can define,

$$Pos(\mathsf{T}) = Root \cdot Branch^*$$

where

$$Root \ := \ Posl \cdot \mathsf{in}^\circ$$

and

$$Branch \ := \ Posr \cdot \mathsf{in}^\circ$$

Thus a position of a tree type consists of a position in the root element after taking a specific path along the branches of the tree. This corresponds to the intuitive notion of positions in tree types. To prove that $Pos(\mathsf{T})$ satisifies the requirements of positions is not so trivial in this case however. Naturality of positions is not too difficult to prove, since in° is a natural transformation and thus an element of $Root \cdot Branch^*$ consists only of the composition positions from *Posl* and *Posr* and in°, and must be a natural transformation as well.

Next let us look at the disjointness condition. We must prove that for all α and β in $Root \cdot Branch^*$ we have that

$$\alpha = \beta \ \equiv \ \alpha> = (\alpha \cap \beta)>$$

Notice that it should be possible to prove this condition using induction. Let

$$Pos_n(\mathsf{T}) \ := \ \cup(i : i \leq n : Root \cdot Branch^i)$$

then we have that

$$Root \cdot Branch^* = \cup\{Pos_n(\mathsf{T}) \ : \ n \in \mathbb{N}\}$$

Thus one needs to show that for any pair of positions in $Pos_0(\mathsf{T})$ the disjointness condition holds (which is not too difficult due to the isomorphism between *Root* and *Posl* and the fact that disjointess holds for *Posl*), and that if disjointness holds for $Pos_k(\mathsf{T})$ then it also holds for $Pos_{k+1}(\mathsf{T})$. The proof depends on the isomorphism in between *Posl* and *Root* and *Posr* and *Branch*, the fact that *Posl* and *Posr* are collection of positions, and that there is only one path to a certain position in a tree, that is, the collection of postions contained in different branches are disjoint.

The proof and of the completeness requirement and the relator requirement are extremely difficult. Specifically, the relator requirement is the generalization of two difficult proofs, namely the fact that the product relator distributes over composition and the fact that the power relator distributes over composition. Furthermore, since the proof the distribution of the power relator over composition depends on tabulations, one would expect that in the general case, this proof does too.

For all the induction proofs it will be necessary that the base relator of the tree type $\otimes$ preserves intersection, since otherwise the inductive argument is not possible as it depends on being able to fuse the base case $R \otimes T\top\!\top$ with the induction hypothesis $\top\!\top \otimes TS$, and thus we want,

$$R \otimes T\top\!\top \;\cap\; \top\!\top \otimes TS \quad = \quad (R \cap \top\!\top) \otimes (T\top\!\top \cap TS) \;=\; R \otimes TS$$

This is only the case if $\otimes$ preserves intersection, a property which holds for the polynomial datatypes and membership, but not for relators in general.

An interesting consequence of this definition is that the *head* and *tail* operations on lists, or in general, the *root* and different *branch* operations on tree types are simply the relations in the collections *Root* and *Branch*. Furthermore, *Root* and *Branch* are isomorphic to the collections of left positions *Posl* and right positions *Posr* of the binary relator $\otimes$, respectively, and thus these so-called selectors are a generic notion induced for an arbitrary tree type by the positions of $\otimes$.

5.4 Conjectures About Positions

In this section we give two conjectures about the combination of positions and shapes. First of all, we conjecture that it is possible to define a generic split in terms of a collection of positions of a relator F, and then using this split we can define a relator with the arity of the number of positions. Secondly we conjecture that it is possible to use positions and shapes to define an arbitrary natural transformation, i.e., that all natural transformations can be defined in terms of positions and shapes.

5.4.1 Relators

Assume we have a relator $\mathsf{F}A = A \times A$, which has positions outl and outr. Then we can define a split operation for F of arity two, denoted by $\vartriangle_{\mathsf{F}}$, by,

$$\vartriangle_{\mathsf{F}}(R,S) = \mathsf{outl}\backslash R \cap \mathsf{outr}\backslash S$$

and a binary functor $\times_{\mathsf{F}}$ by,

$$\times_{\mathsf{F}}(R,S) = \mathsf{outl}\backslash(R \cdot \mathsf{outl}) \cap \mathsf{outr}\backslash(S \cdot \mathsf{outr})$$

In this case $\vartriangle_{\mathsf{F}}$ is simply the binary split operation and $\times_{\mathsf{F}}$ is the binary product relator, and we have $\mathsf{F}A = \times_{\mathsf{F}}(A,A)$ and $\mathsf{F}R = \times_{\mathsf{F}}(R,R)$.

It should be possible to apply this trick for any collection of positions of a relator F. We can define a generic split operation as follows,

$$\vartriangle_{\mathsf{F}}(R) = \cap(\rho :: \; \rho\backslash R_\rho)$$

where R is a mapping from positions to relations. Note that this is the same relation that we have in the completeness requirement and we see that the completeness requirement is a generalization of the so-called product-split fusion rule,

$$R \times S \;\cdot\; (T \vartriangle U) \;=\; (R \cdot T) \vartriangle (S \cdot U)$$

Using the split relation and our positions we conjecture that we can define a relator as follows, where R is again a mapping from positions to relations

$$\mathsf{F}'\mathsf{R} \;=\; (\triangle_{\mathsf{F}}(\lambda\rho\,.\,(\mathsf{R}_{\rho}\cdot\rho)) \cap (\triangle_{\mathsf{F}}(\lambda\rho\,.\,(\mathsf{R}_{\rho}\cdot\rho)))^{\circ}) \cap \mathsf{F}\top\!\!\top$$

If we define the constant mapping R' such that $\mathsf{R}'_{\rho}{=}\mathsf{R}$ for a given relation R, then we also have that,

$$\mathsf{F}'\mathsf{R}' = \mathsf{F}\mathsf{R}$$

5.4.2 Natural Transformations

The interpretation of a lax natural transformation of type $\mathsf{F} \hookleftarrow \mathsf{G}$ is that it converts a G-structure into an F-structure without looking at the elements or changing them. Futhermore, if one restricts a natural transformation's domain or range by a set of shapes, naturality is preserved, and if a natural transformation has a certain shape in its domain, then it must contain all structures of that shape in its domain.

Using this information about natural transformations we should be able to prove that a natural transformation can be split up into a set of smaller natural transformations, each with at most one shape in its range and domain. Obviously the pieces that have no shape in the domain or range are $\perp\!\!\perp$ and these are not interesting and can be discarded. Thus the original transformation is then simply the union of a collection of natural transformations with precisely one shape in the domain and one shape in the range.

If we consider one of the little pieces which contains only one shape in its domain and one shape in its range, we should be able to define the behavior of the natural transformation entirely in terms of positions. To represent an F-shape and a G-shape we consider two functions, namely $p : \mathsf{F1} \leftarrow \mathsf{1}$ and $q : \mathsf{G1} \leftarrow \mathsf{1}$. Since the unit $\mathsf{1}$ has only one element, we have that p represents the unique F-shape in its range, and the set of all F-structures with that shape can then be represented by the partial identity $(\mathsf{F}! \cdot p)<$. Given these two shapes, we conjecture that every natural transformation with only the shape p in its range and the shape q in its domain, has the following form,

$$(\mathsf{F}!\;\cdot\;p)<\;\cdot\;\cap(\alpha :\;\alpha \in \mathit{Pos}(\mathsf{F}) :\;\alpha \setminus f.\alpha)\;\cdot\;(\mathsf{F}!\;\cdot\;q)<$$

where $f.\alpha$ is the union of an arbitrary collection of G-positions. This says that for a given F-shape and G-shape we can fill each F-position in the F-shape by non-deterministically choosing values from an arbitrary collection of G-positions. Thus we conjecture that for every such mapping f we have one of these components of a natural transformation, and that each component can be put in this form.

Since a natural transformation of type $\mathsf{F} \hookrightarrow \mathsf{G}$ is the converse of a natural transformation of type $\mathsf{F} \hookleftarrow \mathsf{G}$, and a natural transformation of type $\mathsf{F} \leftarrow \mathsf{G}$ is a natural transformation of type $\mathsf{F} \hookrightarrow \mathsf{G}$ and $\mathsf{F} \hookleftarrow \mathsf{G}$, we can characterize all natural transformations using lax natural transformations.

Chapter 6

Conclusions and Further Research

6.1 Conclusions

Using allegories as a theoretic model for relational programming it is possible to model both generic datatypes and generic programs. For generic theory of datatypes we must abstract away from all the unnecessary details of specific datatypes and consider only the important properties that any datatype should have. We must also be careful not to make the theory too generic, however, since then it will not be possible to derive anything useful about datatypes other than the painfully obvious. For a generic programs we have seen that the concept of a natural transformation gives a precise mathematical condition to test if a program can be considered to be generic or not. Furthermore, using the notion of a membership relation, we can describe the nature of the natural transformations.

Our contribution to the subject is the theory of positions, which can be used for several goals. First of all, instead of taking the concept of a "relator with membership" as the definition of a datatype, one can define a datatype as a "relator with positions". All the standard datatypes have positions and these positions allow us to fully decompose a data structure so that we know where each element came from. Thus a datatype is a relator or datatype former, together with a collection of positions, or datatype destructors. Furthermore, positions cleanly separate the concepts of "shape" and the "contents" of a data structure; the theory of positions states that the shape of a data structure and the elements contained at each position, uniquely identify the data structure.

Another goal of positions is to characterize natural transformations. The conjecture presented in this thesis is that a natural transformation to F-structures from G-structures can be completely characterized using shapes and positions. This is done by defining the natural transformation on a shape-wise basis, that is, for every F-shape and G-shape one specifies for every position in the F-shape the positions from which the elements may originate in the G-shape. This shows that natural transformations are still very wild beasts, which do not necessarily have to behave consistently over a collection of related shapes. In fact, if we consider natural numbers to be the recursive datatype induced by the functor $(\mathsf{1}+)$ then every relation between natural numbers is a natural transformation.

6.2 Futher Research

There is still a lot of work to be done on positions. The definition is still rather bulky, since there are at least four different properties that must be verified to prove that a collection of relations is a collection of postions. It should be possible to give a shorter, more concise definition. Futhermore, the definition still leaves a lot of ambiguity unless atomicity is assumed, and there are in general many possible collections of positions for a given relator. The question is whether it is possible to give a useful definition of positions which, for an arbitrary relator, has only one solution. This might not be necessary however.

Futhermore, the conjecture that the collections of positions of the standard datatypes that we gave are compositional still needs to be proved. This should be a relatively straight-forward though lengthy exercise, and with a better definition of positions the proof is probably a lot more elegant. The question also arises if there exist other definitions of the positions of the standard data types which are also compositional, or if the compositionality requirement does somehow force positions to behave nicely and as one expects. It would also be nice to see if these positions have any practical value as a generic property of datatypes.

Another important result would be to prove the conjecture of the relationship between positions and natural transformations, namely that positions together with shapes can be used to characterize natural transformations. This would nicely tie all the theory together, since datatype formers and positions as datatype destructors would complement each other in the theory of datatypes, and the positions and shapes can be used to define generic programs on datatypes, including the polymorphic map operation and natural transformations.

Finally, it still needs to be proved that one can use the generic definition of a split to define a relator with the same arity as the number of positions that a datatype has. Perhaps it is possible to use this relator in a better definition of positions.

Bibliography

[1] C.J. Aarts. Galois connections presented calculationally. Afstudeer verslag (Graduating Dissertation), Department of Computing Science, Eindhoven University of Technology, 1992.

[2] Roland Backhouse. Mathematics of program construction. In preparation. Copies of draft available on request. Last revision: October 1997, 370 pages.

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

[4] Oege de Moor and Ganesh Sittampalam. Generic program transformation. Lecture notes for the 3rd International Summer School on Advanced Functional Programming, Braga, Portugal, 1998.

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

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

[7] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25(251–283), 1995.

[8] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *ESOP '94: 5th European Symposium on Programming*, pages 302–316. Springer Verlag Lecture Notes in Computer Science, April 1994.

[9] O. de Moor. *Categories, Relations and Dynamic Programming*. PhD thesis, Oxford University Laboratory, Programming Research Group, April 1992.