Generic Programming
– An Introduction –

Roland Backhouse
Patrik Jansson
Johan Jeuring
Lambert Meertens

Roland Backhouse
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven
The Netherlands
email: rolandb@win.tue.nl
url: http://www.win.tue.nl/~rolandb/

Patrik Jansson
Department of Computing Science
Chalmers University of Technology
S-412 96 Göteborg
Sweden
email: patrikj@cs.chalmers.se
url: http://www.cs.chalmers.se/~patrikj/

Johan Jeuring
Department of Computer Science
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
email: johanj@cs.uu.nl
url: http://www.cs.uu.nl/~johanj/

Lambert Meertens
CWI & Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
email: lambert@cwi.nl
url: http://www.cwi.nl/~lambert/

Contents

Chapter 1

Introduction

1.1 The Abstraction-Specialisation Cycle

The development of science proceeds in a cycle of activities, the so-called abstraction-specialisation cycle. *Abstraction* is the process of seeking patterns or commonalities, which are then classified, often in a formal mathematical framework. In the process of abstraction, we gain greater understanding by eliminating irrelevant detail in order to identify what is essential. The result is a collection of general laws which are then put to use in the second phase of the cycle, the *specialisation* phase. In the specialisation phase the general laws are instantiated to specific cases which, if the abstraction is a good one, leads to novel applications, yet greater understanding, and input for another round of abstraction followed by specialisation.

The abstraction-specialisation cycle is particularly relevant to the development of the science of computing because the modern digital computer is, above all else, a *general-purpose* device that is used for a dazzling range of tasks. Harnessing this versatility is the core task of software design.

Good, commercially viable, software products evolve in a cycle of abstraction and *customisation*. Abstraction, in this context, is the process of identifying a single, general-purpose product out of a number of independently arising requirements. Customisation is the process of optimizing a general-purpose product to meet the special requirements of particular customers. Software manufacturers are involved in a continuous process of abstraction followed by customisation.

1.2 Genericity in Programming Languages

The abstraction-specialisation/customisation cycle occurs at all levels of software design. Programming languages play an important role in facilitating its implementation. Indeed, the desire to be able to name and reuse "programming patterns" —capturing them in the form of parametrisable abstractions— has been a driving force in the evolution of high-level programming languages to the extent that the level of "genericity" of a programming language has become a vital criterion for usability.

To determine the level of genericity there are three questions we can ask:

- What kind of entities can be named in a definition and then referred to by that given name?
- What kind of entities can be supplied as a parameter?
- What kind of entities can be used "anonymously", in the form of an expression, as a parameter? (For example, in $y = \mathsf{sin}(2 \times x)$, the number resulting from $2 \times x$ is not given a name. In a language allowing numeric parameters, but not anonymously, we would have to write something like $y = \mathsf{sin}(z)$ **where** $z = 2 \times x$.)

A kind of entities for which all three are possible is called a *first-class citizen* of that language.

In one of the first high-level programming languages, FORTRAN (1957), procedures could be named, but not used as parameters. In ALGOL 60 procedures (including functions) were made almost first-class citizens: they were allowed as parameters, but only by name. In neither language could types be named, nor passed as parameter. In ALGOL 68 procedures were made true first-class citizens, making higher-order functions possible (but not practical, because of an awkward scope restriction[1]). Further, types could be named, but not used as parameters.

Functional programming languages stand out in the evolution of programming languages because of the high-level of abstraction that is achieved by the combination of higher-order functions and parametric polymorphism. In, for example, Haskell higher-order functions are possible *and practical*. But the level of genericity still has its limitations. Types can be defined *and* used as parameters, but ... types can only be given as parameters in "type expressions". They cannot be passed to functions.

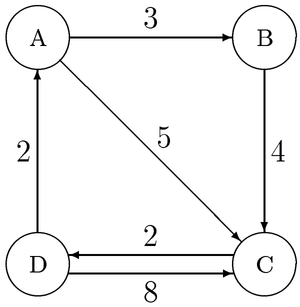
In these lecture notes we introduce another dimension to the level of abstraction in programming languages, namely parameterisation with respect to classes of algebras of variable signature. This first chapter is intended to introduce the key elements of the lectures in broad terms and to motivate what is to follow. We begin by giving a concrete example of a generic algorithm. (The genericity of this algorithm is at a level that can be implemented in conventional functional programming languages, since the parameter is a class of algebras with a fixed signature.) This is followed by a plan of the later chapters.

[1]Anything that —in implementation terms— would have required what is now known as a "closure", was forbidden.

1.3 Path Problems

A good example of parameterising programs by a class of algebras is provided by the problem of finding "extremal" paths in a graph.

Extremal path problems have as input a finite, labelled graph such as the one shown below.



Formally, a *graph* consists of a finite set of *nodes*, V, a finite set of *edges*, E and two functions source and target, each with domain E and range V. If source e is the node x and target e is the node y, we say that e is *from* x *to* y. (In the figure the nodes are circled and an edge e is depicted by an arrow that begins at source e and points to target e.) A *path* through the graph from node s to node t of *edge length* n is a finite sequence of nodes x_0, x_1, ..., x_n such that $s = x_0$ and $t = x_n$ and, for each i, $0 \leq i < n$, there is an edge in the graph from x_i to x_{i+1}. A graph is *labelled* if it is supplied with a function label whose domain is the set of edges, E.

In an extremal path problem the edge labels are used to weight paths, and the problem is to find the extreme (i.e. best or least, in some sense) weight of paths between given pairs of nodes. We discuss three examples: the reachability problem, the least-cost path problem and the bottleneck problem.

Reachability The *reachability* problem is the problem of determining for each pair of nodes x and y whether there is a path in the graph from x to y. It is solved by a very elegant (and now well-known) algorithm discovered by Roy [40] and Warshall [44]. The algorithm assumes that the nodes are numbered from 1 to N (say) and that the existence of edges in the graph is given by an $N \times N$ matrix a where a_{ij} is true if there is an edge from node numbered i to the node numbered j, and false otherwise. The matrix is updated by the following code. On termination a_{ij} is true if there is a path from node i to node j of edge length at least one; otherwise a_{ij} is false.

for each k, $1 \leq k \leq N$

do for each pair (i,j), $1 \leq i,j \leq N$

```
        do a_ij  :=  a_ij ∨ (a_ik ∧ a_kj)
        end_for
    end_for
```

(The order in which the nodes are numbered, and the order in which pairs of nodes (i,j) are chosen in the inner loop, is immaterial.)

The reachability problem is an extremal path problem in which all edges have the same label and all paths have the same weight, namely true.

Least-Cost Paths About the same time as Warshall's discovery of the reachability algorithm, Floyd [16] discovered a very similar algorithm that computes the cost of a least cost path between each pair of nodes in the graph. The algorithm assumes that the matrix a is a matrix of numbers such that a_{ij} represents the least *cost* of traversing an edge from node i to node j. If there is no edge from i to j then a_{ij} is ∞. The *cost* of a path is the sum of the costs of the individual edges on the path. Floyd's algorithm for computing the cost of a least cost path from each node to each other node is identical to the Roy-Warshall algorithm above except for the assignment statement which instead is:

$$a_{ij} \quad := \quad a_{ij} \downarrow (a_{ik} + a_{kj})$$

where $x \downarrow y$ denotes the minimum of x and y.

Bottleneck Problem A third problem that can be solved with an algorithm of identical shape to the Roy-Warshall algorithm is called the *bottleneck* problem. It is most easily explained as determining the best route to negotiate a high load under a series of low bridges. Suppose an edge in a graph represents a road between two cities and the label is the height of the lowest underpass on the road. The *height* of a path between two nodes is defined to be the minimum of the heights of the individual edges that make up the path. The problem is to determine, for each pair of nodes i and j, the maximum of the heights of the paths from node i to node j (thus the maximum of the minimum height underpass on a path from i to j).

The bridge height problem is solved by an algorithm identical to the Roy-Warshall algorithm above except for the assignment statement which in this case is:

$$a_{ij} \quad := \quad a_{ij} \uparrow (a_{ik} \downarrow a_{kj})$$

where $x \downarrow y$ denotes the minimum of x and y and $x \uparrow y$ denotes their maximum. (In the case that there is no edge from i to j then the initial value of a_{ij} is 0.)

A Generic Path Algorithm If we abstract from the general shape of these three algorithms we obtain a single algorithm of the form

for each k, $1 \leq k \leq N$
do for each pair (i,j), $1 \leq i,j \leq N$
    do $a_{ij} \;:=\; a_{ij} \oplus (a_{ik} \otimes a_{kj})$
    end_for
end_for

where $\oplus$ and $\otimes$ are binary operators. The initial value of a_{ij} is the label of the edge from i to j if such an edge exists, and is a constant **0** otherwise. (For the purposes of exposition we assume that there is at most one edge from i to j for each pair of nodes i and j.) The algorithm is thus parameterised by an algebra. In the case of the Roy-Warshall algorithm the carrier of the algebra is the two-element set containing true and false, the constant **0** is false, the operator $\oplus$ is disjunction and the operator $\otimes$ is conjunction. In the case of the least-cost path problem the carrier is the set of positive real numbers, the constant **0** is ∞, the operator $\oplus$ is the binary minimum operator, and the operator $\otimes$ is addition. Finally, in the case of the bridge height problem the carrier is also the set of positive real numbers, the operator $\oplus$ is the binary maximum operator, and the operator $\otimes$ is minimum.

Correctness The above generic algorithm will compute "something" whatever actual parameters we supply for the formal parameters $\oplus$, $\otimes$ and **0**, the only proviso being that the parameters have compatible types. But, that "something" is only guaranteed to be meaningful if the operators obey certain algebraic properties. The more general *transitive closure* algorithm shown below

for each k, $1 \leq k \leq N$
do for each pair (i,j), $1 \leq i,j \leq N$
    do $a_{ij} \;:=\; a_{ij} \oplus (a_{ik} \otimes (a_{kk})^* \otimes a_{kj})$
    end_for
end_for

is guaranteed to be correct if the algebra is *regular* [5, 7][2]. By correctness is meant that if initially

$$a_{ij} \;=\; \Sigma\langle e\text{: } e \text{ is an edge from } i \text{ to } j\text{: } \mathsf{label}\ e\rangle \quad ,$$

[2]Without going into complete details, an algebra is regular if it has two constants **0** and **1**, two binary operators $\oplus$ and $\otimes$, and one unary operator *. The constants **0** and **1** and operators $\oplus$ and $\otimes$ should behave like 0, 1, + and $\times$ in real arithmetic except that $\times$ is not required to be commutative, and + is required to be idempotent. The * operator is a least fixed point operator. The three algebras mentioned above are all regular, after suitably defining the constant **1** and defining a^* to be **1** for all a.

where Σ is the generalisation of the binary operator $\oplus$ to arbitrary bags, then on termination

$$a_{ij} \;=\; \Sigma\langle p\text{: } p \text{ is a path of positive edge length from } i \text{ to } j\text{: } \mathsf{weight}\ p\rangle$$

where $\mathsf{weight}\ p$ is defined recursively by

$$\mathsf{weight}\ p = \mathbf{1}$$

for all paths p of edge length 0, and for paths $e : p$ (the edge e followed by path p)

$$\mathsf{weight}\ (e : p) = (\mathsf{label}\ e) \otimes (\mathsf{weight}\ p) \quad .$$

Exercise 1.1 Suppose that the edges of a graph are coloured. (So there are blue edges, red edges, etc.) We say that a path has *colour c* if all the edges on the path have colour c. Suggest how to use the above algorithm to determine for each pair of nodes x and y the set of colours c such that there is a path of colour c from x to y in the graph.

□

1.4 The Plan

The difference between executability and correctness is an important one that shows up time and time again, and it is important to stress it once more. The transitive closure algorithm presented above can be *executed* provided only that instantiations are given for the unary operator *, the two binary operators $\oplus$ and $\otimes$, the number N and the matrix a. An implementation of the algorithm thus requires just the specification of these five parameters. Moreover, if we bundle the first three parameters together into an algebra, all that is required for the implementation is the *signature* of the algebra: the knowledge that there is one unary operator and two binary operators. For the *correctness* of the algorithm much, much more is needed. We have to supply a specification relative to which correctness is asserted, and establishing correctness demands that we require the algebra to be in a certain *class* of algebras (in this case the class of regular algebras).

As for conventional programs, the specification is absent from a generic program's implementation. Nevertheless, it is the complete process of *program construction* —from program specification to a systematic derivation of the final implementation— that will dominate the discussion in the coming pages. Our aim is not to show how to derive functional programs but to show how to derive functional programs that are correct by construction. To this end we borrow a number of concepts from category theory, emphasising the calculation properties that these concepts entail.

Algebras, Functors and Datatypes The emphasis on calculational properties begins right at the outset in chapter 2 where we introduce the notion of a *functor* and an *initial algebra* and relate these notions to *datatypes*.

An *algebra* (in its simplest form) is a set, called the *carrier* of the algebra, together with a number of operations on that set. A Boolean algebra, for example, has as carrier a set with two elements, commonly named true and false and binary operations $\wedge$ (conjunction) and $\vee$ (disjunction) and unary operation $\neg$ (negation). The *signature* of the algebra specifies the arities of the basic operations in the algebra.

In order to implement a generic algorithm we need to provide the compiler with information on the signature of the operators in the algebra on which the algorithm is parameterised. In order to calculate and reason about generic algorithms we also need a *compact* mechanism for defining signatures. The use of functors provides such a mechanism, compactness being achieved by avoiding naming the operators of the algebra. The use of functors entails much more however than just defining the signature of an algebra. As we shall see, a datatype is a functor and inductively defined datatypes are (the carriers of) *initial* algebras. The concepts of functor, datatype and algebra are thus inextricably intertwined.

PolyP Following the discussion of algebras and datatypes, we introduce *PolyP*, an extension of the Haskell programming language in which generic functions can be implemented.

The name of PolyP is derived from "polytypic programming", polytypic programs being generic programs defined on a particular class of datatypes, the so-called regular datatypes. Writing programs in PolyP means that one can get hands-on experience of generic programming thus reinforcing one's understanding and, hopefully, leading to further insights.

A Unification Algorithm Chapter 4 presents a more substantial example of generic programming — a generic unification algorithm. The basis for the algorithm is a generic construction of a type representing terms with variables, and substitution of terms for variables. The algorithm is implemented using type classes in a style similar to object-oriented programming.

Relations The discussion in chapters 2 and 3 is on *functional* programs. In chapter 5 we outline how the concepts introduced in chapter 2 are extended to *relations*, and we show how the extension is used in establishing one element of the correctness of the generic unification algorithm.

There are several reasons for wanting to take the step from functions to relations. The most pressing is that specifications are *relations* between the input and the output, and our concern is with both specifications and implementations. Related to this is that termination properties of programs are typically established by appeal to a well-founded *relation* on the state space. We will not go into termination properties in these lecture notes but the use of well-founded relations will play an integral part in our discussion of one element of the correctness of a generic unification algorithm in chapter 4.

Another reason for wanting to extend the discussion to relations lies in the theoretical basis of generic programming. In chapter 5 we demonstrate how every parametrically polymorphic function satisfies a so-called *logical relation*.

The final reason is *why not?* As we shall see, extending the theory to relations does not significantly increase the complexity whilst the benefits are substantial.

1.5 Why Generic Programming?

The form of genericity that we present in the coming pages is novel and has not yet proved its worth. Our goal is to stimulate your interest in exploring it further, and to provide evidence of its potential value.

Generic programming has indeed, potentially, major advantages over "one-shot" programming, since genericity makes it possible to write programs that solve a class of problems once and for all, instead of writing new code over and over again for each different instance. The two advantages that we stress here are the greater potential for reuse, since generic programs are natural candidates for incorporation in library form, and the increased reliability, due to the fact that generic programs are stripped of irrelevant detail which often makes them easier to construct. But what we want to stress most of all is that generic programming is fun. Finding the right generic formulation that captures a class of related problems can be a significant challenge, whose achievement is very satisfying.

Acknowledgements The work that is presented here emerged out of the Dutch STOP (Specification and Transformation of Programs) project which ran formally from 1988 to 1992. The project was particularly successful because of the real spirit of cooperation among those participating. Project members (both official and unofficial) included, in alphabetical order, Roland Backhouse, Richard Bird, Henk Doornbos, Maarten Fokkinga, Paul Hoogendijk, Johan Jeuring, Grant Malcolm, Lambert Meertens, Erik Meijer, Oege de Moor, Frans Rietman, Doaitse Swierstra, Jaap van der Woude, Nico Verwer, Ed Voermans. Our thanks go to all who made participation in the project such an enjoyable and stimulating experience.

Development of both practical applications of generic programming and the underlying theory is continuing: see the bibliography for a selection of recent (formally-published and web-published) papers.

Chapter 2

Algebras, Functors and Datatypes

This chapter introduces the concepts fundamental to generic programming. The first section (section 2.1) introduces algebras and homomorphisms between algebras. In this section we see that datatypes (like the natural numbers) are also algebras, but of a special kind. The presentation in section 2.1 is informal. In section 2.4 we make precise in what way datatypes are special: we introduce the all-important notion of an "initial" algebra and the notion of a "catamorphism" (a special sort of homomorphism). The link between the two sections is provided by the intermediate sections on functors. The first of these (section 2.2) provides the formal definition of a functor, motivating it by examples from functional programming. Then section 2.3 introduces further examples of functors forming a class called the "polynomial functors". Section 2.4 augments the class of polynomial functors with so-called type functors; the resulting class is called the class of "regular functors", and generic programs defined over the regular functors are called "polytypic" programs. The final section (section 2.5) presents an elementary example of a polytypic program.

2.1 Algebras and Homomorphisms

In this section we review the notion of an algebra. The main purpose is to introduce several examples that we can refer to later. The examples central to the discussion are datatypes. At the end of the section we consider how we might formalise the notion of an algebra. We recall a formulation typical of ones in texts on Universal Algebra and remark why this is inadequate for our purposes. We then present the definition of an algebra in category theory based on the notion of a "functor" and outline how the latter expresses the content of the traditional definitions much more succinctly and in a much more structured way.

Algebras An algebra is a set, together with a number of operations (functions) that return values in that set. The set is called the *carrier* of the algebra. Here are some concrete examples of algebras:

$$
\begin{array}{llll}
(\mathbb{N}, (+), 0), & \text{with} & (+) :: \mathbb{N} \leftarrow \mathbb{N} \times \mathbb{N}, & 0 :: \mathbb{N} \leftarrow 1 \\
(\mathbb{N}, (\uparrow), 0), & \text{with} & (\uparrow) :: \mathbb{N} \leftarrow \mathbb{N} \times \mathbb{N}, & 0 :: \mathbb{N} \leftarrow 1 \\
(\mathbb{R}, (\times), 1), & \text{with} & (\times) :: \mathbb{R} \leftarrow \mathbb{R} \times \mathbb{R}, & 1 :: \mathbb{R} \leftarrow 1 \\
(\mathbb{B}, (\equiv), \mathsf{true}), & \text{with} & (\equiv) :: \mathbb{B} \leftarrow \mathbb{B} \times \mathbb{B}, & \mathsf{true} :: \mathbb{B} \leftarrow 1 \\
(\mathbb{B}, (\vee), \mathsf{false}), & \text{with} & (\vee) :: \mathbb{B} \leftarrow \mathbb{B} \times \mathbb{B}, & \mathsf{false} :: \mathbb{B} \leftarrow 1 \\
(\mathbb{B}, (\wedge), \mathsf{true}), & \text{with} & (\wedge) :: \mathbb{B} \leftarrow \mathbb{B} \times \mathbb{B}, & \mathsf{true} :: \mathbb{B} \leftarrow 1 \\
(A^\star, (+\!\!+), \varepsilon), & \text{with} & (+\!\!+) :: A^\star \leftarrow A^\star \times A^\star, & \varepsilon :: A^\star \leftarrow 1
\end{array}
$$

In the last line $A^\star$ stands for the words over some alphabet A, with "$+\!\!+$" denoting word concatenation, and "ε" the empty word. This is, of course, basically the same algebra as $(List\ A, (+\!\!+), [\,])$, the (finite) lists of A-elements with list concatenation. Note that in the typing of the operations we use the notation "target-type $\leftarrow$ source-type". In an algebra all operations have the same target type[1]: its carrier. Note further that we use the "uncurried" view in which a binary operation takes a pair (2-tuple) of arguments and so has some type like $A \leftarrow B \times C$. To make fixed elements, like $0 \in Nat$, fit in, they are treated here as nullary operations: operations with a 0-tuple of arguments. This is indicated by the source type $\mathsf{1}$, which in Haskell would be denoted as "$(\,)$". Sometimes we will instantiate a generic program to a specific Haskell program, and in doing so we will switch back to the conventional curried view for binary operations in these languages, having some type $(A \leftarrow C) \leftarrow B$, and to the view of nullary operations as plain elements, having type A rather than $A \leftarrow \mathsf{1}$. Conversely, going from a Haskell program to an algebraic view, we will uncurry n-ary functions, $n \geq 2$, and treat constants as nullary functions.

The concrete algebras above were chosen in such a way that they all have the same number of operations with the same typing pattern. They can be unified generically into the following abstract algebra:

$$(A, \oplus, e), \quad \text{with} \quad \oplus :: A \leftarrow A \times A, \quad e :: A \leftarrow \mathsf{1}$$

So they all belong to the same *class* of algebras. An example of another class of algebras is:

$$
\begin{array}{llll}
(\mathbb{N}, (+), (+1)), & \text{with} & (+) :: \mathbb{N} \leftarrow \mathbb{N} \times \mathbb{N}, & (+1) :: \mathbb{N} \leftarrow \mathbb{N} \\
(\mathbb{R}, (\times), (\times 2)), & \text{with} & (\times) :: \mathbb{R} \leftarrow \mathbb{R} \times \mathbb{R}, & (\times 2) :: \mathbb{R} \leftarrow \mathbb{R} \\
(A, \oplus, f), & \text{with} & \oplus :: A \leftarrow A \times A, & f :: A \leftarrow A
\end{array}
$$

Here, the first two are concrete, while the last is the generic algebra.

By just looking at an algebra, it is not possible (in general) to tell what class of algebras it

[1] We freely identify types and sets whenever convenient.

belongs to: a given algebra can belong to several different classes. So the class information has to be supplied additionally. Take for example the following class:

$$\begin{array}{llll}
(\mathbb{N}, (+), 0), & \text{with} & (+) :: \mathbb{N} \leftarrow \mathbb{N} \times \mathbb{N}, & 0 :: \mathbb{N} \leftarrow 1 \\
(\mathbb{N}, (\uparrow), 0), & \text{with} & (\uparrow) :: \mathbb{N} \leftarrow \mathbb{N} \times \mathbb{N}, & 0 :: \mathbb{N} \leftarrow 1 \\
(\mathit{List}\ \mathbb{N}, (:), [\,]), & \text{with} & (:) :: \mathit{List}\ \mathbb{N} \leftarrow \mathbb{N} \times \mathit{List}\ \mathbb{N}, & [\,] :: \mathit{List}\ \mathbb{N} \leftarrow 1 \\
(A, \oplus, e), & \text{with} & \oplus :: A \leftarrow \mathbb{N} \times A, & e :: A \leftarrow 1
\end{array}$$

The first two concrete algebras also occur in the first class treated above, but the generic algebra reveals that this is a different class.

To give a concluding example of an algebra class:

$$\begin{array}{llll}
(\mathbb{N}, (+1), 0), & \text{with} & (+1) :: \mathbb{N} \leftarrow \mathbb{N}, & 0 :: \mathbb{N} \leftarrow 1 \\
(\mathbb{R}, (\times 2), 1), & \text{with} & (\times 2) :: \mathbb{R} \leftarrow \mathbb{R}, & 1 :: \mathbb{R} \leftarrow 1 \\
(\mathbb{B}, (\neg), \mathsf{true}), & \text{with} & (\neg) :: \mathbb{B} \leftarrow \mathbb{B}, & \mathsf{true} :: \mathbb{B} \leftarrow 1 \\
(\mathbb{B}, (\neg), \mathsf{false}), & \text{with} & (\neg) :: \mathbb{B} \leftarrow \mathbb{B}, & \mathsf{false} :: \mathbb{B} \leftarrow 1 \\
(A, f, e), & \text{with} & f :: A \leftarrow A, & e :: A \leftarrow 1
\end{array}$$

A recursively defined datatype determines, in a natural way, an algebra. A simple example is the datatype Nat defined by:

$$\textbf{data}\ Nat \ = \ \mathsf{succ}\ Nat \mid \mathsf{zero}$$

The corresponding algebra is:

$$(Nat, \mathsf{succ}, \mathsf{zero}), \quad \text{with} \quad \mathsf{succ} :: Nat \leftarrow Nat, \quad \mathsf{zero} :: Nat \leftarrow 1$$

This belongs to the last class mentioned; in fact, if we ignore the possibility of infinite data structures —made possible by lazy evaluation— this is essentially the same algebra as ($\mathbb{N}$, $(+1)$, 0). Another example is:

$$\textbf{data}\ Natlist \ = \ \mathsf{cons}\ \mathbb{N}\ Natlist \mid \mathsf{nil}$$

The corresponding algebra is:

$$(Natlist, \mathsf{cons}, \mathsf{nil}), \quad \text{with} \quad \mathsf{cons} :: Natlist \leftarrow \mathbb{N} \times Natlist, \quad \mathsf{nil} :: Natlist \leftarrow 1$$

This is basically the same as ($\mathit{List}\ \mathbb{N}$, $(:)$, $[\,]$). Both of these examples illustrate the general phenomenon that a recursively defined datatype determines an algebra in which the carrier of the algebra is the datatype itself, and the constructors of the datatype are the operations of the algebra.

Homomorphisms A homomorphism between two algebras, which must be *from the same class*, is a function between their carrier sets that "respects the structure" of the class. For example, the function $\mathsf{exp} :: \mathbb{R} \leftarrow \mathbb{N}$ is a homomorphism with as *target algebra* $(\mathbb{R},\ (\times),\ 1)$ and as *source algebra* $(\mathbb{N},\ (+),\ 0)$. In this case, respecting the structure of this algebra class means that it satisfies the following two properties:

$$\begin{array}{lcl} \mathsf{exp}(x+y) & = & (\mathsf{exp}\ x) \times (\mathsf{exp}\ y) \\ \mathsf{exp}\ 0 & = & 1 \end{array}$$

Another example in the same class is $\mathsf{length} :: (\mathbb{N},\ (+),\ 0) \leftarrow (A^\star,\ (+\!\!+),\ \varepsilon)$. (This notation is shorthand for the statement that the function $\mathsf{length} :: \mathbb{N} \leftarrow A^\star$ is a homomorphism to target algebra $(\mathbb{N},\ (+),\ 0)$ from source algebra $(A^\star,\ (+\!\!+),\ \varepsilon)$.) In this case, respecting the structure means:

$$\begin{array}{lcl} \mathsf{length}(x +\!\!+ y) & = & (\mathsf{length}\ x) + (\mathsf{length}\ y) \\ \mathsf{length}\ \varepsilon & = & 0 \end{array}$$

In general (for this class of algebras), $h :: (A,\ \oplus,\ e) \leftarrow (B,\ \otimes,\ u)$ means:

$$\begin{array}{lcl} h :: A \leftarrow B & & \\ h(x \otimes y) & = & (h\ x) \oplus (h\ y) \\ h\ u & = & e \end{array}$$

So to apply h to a value in B that resulted from a $\otimes$-operation, we may equally well apply h to the operands (which gives two A-values) and combine these with the operation $\oplus$. Similarly, to apply h to a value in B that resulted from a u-operation (and there is only one such value), we may equally apply h to the operands (of which there are none) and apply e to the resulting 0-tuple. Here are some more examples of homomorphisms in this class:

$$\begin{array}{rl} (\downarrow 1) & :: (\mathbb{N},\ (\uparrow),\ 0\ ) \leftarrow (\mathbb{N},\ (+),\ 0\ ) \\ \mathsf{even} & :: (\mathbb{B},\ (\equiv),\ \mathsf{true}) \leftarrow (\mathbb{N},\ (+),\ 0\ ) \\ (>0) & :: (\mathbb{B},\ (\vee),\ \mathsf{false}) \leftarrow (\mathbb{N},\ (\uparrow),\ 0\ ) \\ (\neg) & :: (\mathbb{B},\ (\vee),\ \mathsf{false}) \leftarrow (\mathbb{B},\ (\wedge),\ \mathsf{true}) \\ (\neg) & :: (\mathbb{B},\ (\wedge),\ \mathsf{true}) \leftarrow (\mathbb{B},\ (\vee),\ \mathsf{false}) \end{array}$$

If we have two homomorphisms $h :: (A,\ \oplus,\ e) \leftarrow (B,\ \otimes,\ u)$ and $k :: (B,\ \otimes,\ u) \leftarrow (C,\ \odot,\ z)$, in which the source algebra of the first is the target algebra of the second, then their composition is also a homomorphism $h \bullet k :: (A,\ \oplus,\ e) \leftarrow (C,\ \odot,\ z)$. For example,

$$\begin{array}{rl} (>0) \bullet (\downarrow 1) & :: (\mathbb{B},\ (\vee),\ \mathsf{false}) \leftarrow (\mathbb{N},\ (+),\ 0\ ) \\ (\neg) \bullet (\neg) & :: (\mathbb{B},\ (\vee),\ \mathsf{false}) \leftarrow (\mathbb{B},\ (\vee),\ \mathsf{false}) \end{array}$$

Now $(>0) \bullet (\downarrow 1)\ =\ (>0)$ on $\mathbb{N}$, and $(\neg) \bullet (\neg)\ =\ \mathsf{id}_{\mathbb{B}}$ (the identity function on $\mathbb{B}$), so we have

$$\begin{array}{rl} (>0) & :: (\mathbb{B},\ (\vee),\ \mathsf{false}) \leftarrow (\mathbb{N},\ (+),\ 0\ ) \\ \mathsf{id} & :: (\mathbb{B},\ (\vee),\ \mathsf{false}) \leftarrow (\mathbb{B},\ (\vee),\ \mathsf{false}) \end{array}$$

The identity function id_A is of course a homomorphism between *any* algebra with carrier A and itself.

For the class of algebras whose generic algebra is

$$(A, \oplus, e), \quad \text{with} \quad \oplus :: A \leftarrow \mathbb{N} \times A, \quad e :: A \leftarrow \mathbf{1}$$

we have that $h :: (A,\ \oplus,\ e) \leftarrow (B,\ \otimes,\ u)$ means:

$$\begin{array}{lcl} h :: A \leftarrow B & & \\ h(x \otimes y) & = & x \oplus (h\, y) \\ h\ u & = & e \end{array}$$

So why is h for this class not applied to the occurrence of x in the righthand side of the first equality? The answer is that that would not make sense, since h has source type B, but x is of type $\mathbb{N}$. (Later, after we have introduced functors, we shall see how to define the notion of homomorphism generically, independent of the specific algebra class.) We have:

$$\begin{array}{rl} \mathsf{sum} & :: (\mathbb{N}, (+), 0) \leftarrow (\mathit{List}\ \mathbb{N}, (:), [\,]) \\ \mathsf{foldr}\ \oplus\ e & :: (A,\ \oplus\ , e) \leftarrow (\mathit{List}\ \mathbb{N}, (:), [\,]) \end{array}$$

In fact, $\mathsf{sum}\ =\ \mathsf{foldr}\ (+)\ 0$.

Uniqueness We have given several examples of algebra classes and their homomorphisms. The first class had generic algebra

$$(A,\ \oplus,\ e) \quad \text{with} \quad \oplus\ ::\ A \leftarrow A \times A, \quad e\ ::\ A \leftarrow \mathbf{1}\ .$$

Note that the fact that a function is a homomorphism of algebras in this class does not uniquely define the function. For example, we observed above that length is a homomorphism with target $(\mathbb{N},\ (+),\ 0)$ and source $(A^*,\ (+\!\!+),\ \varepsilon)$. But the function that is constantly 0 for all lists is also a homomorphism with exactly the same target and source algebras. Indeed, in the case of all the examples we gave of homomorphisms between algebras in this class the constant function returning the value e of the target algebra has the same homomorphism type as the given function.

Contrast this with the third class of algebras. The generic algebra has the form

$$(A,\ \oplus,\ e) \quad \text{with} \quad \oplus\ ::\ A \leftarrow \mathbb{N} \times A, \quad e\ ::\ A \leftarrow \mathbf{1}$$

Again, the fact that a function is a homomorphism of algebras in this class does not uniquely define the function. But there is something rather special about the algebra $(\mathit{List}\ \mathbb{N},\ (:),\ [])$ in this class of algebras. Specifically, $\mathsf{foldr} \oplus e$ is the *unique* homomorphism with target algebra $(A,\ \oplus,\ e)$ and source algebra $(\mathit{List}\ \mathbb{N},\ (:),\ [\,])$. For example, sum is the unique homomomorphism with target $(\mathbb{N}, (+), 0)$ and source $(List\ \mathbb{N}, (:), [\,])$ That is, function h satisfies the equations

$$\begin{aligned} h &:: \mathbb{N} \leftarrow List\ \mathbb{N} \\ h(x{:}y) &= x + (h\ y) \\ h\ [\,] &= 0 \end{aligned}$$

if and only if $h = \mathsf{sum}$.

This is an important property that will be a focus of later discussion.

Isomorphisms Above, we said several times that two algebras were "basically" or "essentially" the same. We want to make this notion precise. The technical term for this is that these algebras are *isomorphic*. In set theory, two sets A and B are called isomorphic whenever there exists a bijection between A and B. Equivalently, A and B are isomorphic whenever there exist functions $f :: A \leftarrow B$ and $g :: B \leftarrow A$ that cancel each other, that is:

$$\begin{aligned} f \bullet g &= \mathsf{id}_A \\ g \bullet f &= \mathsf{id}_B \end{aligned}$$

The generalisation for algebras is now that we require these functions to be *homomorphisms* between the algebras involved. A homomorphism that has a cancelling homomorphism is called an *isomorphism*. From the examples above we see that the algebras $(\mathbb{B}, (\wedge), \mathsf{true})$ and $(\mathbb{B}, (\vee), \mathsf{false})$ are isomorphic.

Algebras with laws Although we will hardly use this, no account of the notion of algebra is complete without mentioning the following. A class of algebras can be further determined by a set of laws. In a "lawful" class of algebras, all algebras satisfy the same set of (possibly conditional) equational laws. Monoids form the best-known example of a lawful algebra class. The generic monoid is $(A, \oplus, e)$, with $\oplus :: A \leftarrow A \times A$, $e :: A \leftarrow \mathbf{1}$, and the monoid laws are the following two:

$\oplus$ is *associative*: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
e is *neutral* for $\oplus$: $e \oplus x = x = x \oplus e$

If an operation $\oplus$ has a neutral element, it is unique, and we denote it as $\nu_\oplus$. For example, $\nu_+ = 0$ and $\nu_\times = 1$. The examples of concrete algebras from the first class treated in this chapter are actually all monoids. For lawful algebras the definition of homomorphism is the same as before.

Graphs The notion of homomorphism is more general than that of a "structure-respecting" function between algebras. Homomorphisms can generally be defined for anything having structure. As an example, we consider homomorphisms between directed graphs. Recall that a directed graph is a structure

$$(V,\ E,\ \mathsf{target},\ \mathsf{source}), \quad \text{with} \ \ \mathsf{target} :: V \leftarrow E,\ \ \mathsf{source} :: V \leftarrow E$$

in which the elements of V are called "vertices" or "nodes", and the elements of E are called "edges" or "arcs". If edge e is an edge to node m from node n, we have: $\mathsf{target}\, e = m$ and $\mathsf{source}\, e = n$. Directed graphs are just like an algebra class, except that we have *two* "carrier sets": V and E. (There is a term for algebras with more carrier sets: *heterogeneous* or *multi-sorted* algebras.) A homomorphism to graph (V_0, E_0, target_0, source_0) from graph (V_1, E_1, target_1, source_1) is a *pair* of functions, one with the typing $V_0 \leftarrow V_1$ and one with the typing $E_0 \leftarrow E_1$, and if we overload the identifier h to denote both functions, they satisfy:

$$\begin{aligned} h(\mathsf{target}\; a) &= \mathsf{target}(h\, a) \\ h(\mathsf{source}\; a) &= \mathsf{source}(h\, a) \end{aligned}$$

As before for algebras, two graphs are isomorphic whenever there are cancelling homomorphisms between them. Informally, this means that one graph can be obtained from the other by systematic renaming. In standard Graph Theory, for unlabelled graphs like the ones we are considering here, two isomorphic graphs are usually considered *identical*. Still, there can be non-trivial *automorphisms*, that is, isomorphisms between a graph and itself that are not the identity isomorphism.

Summarising and looking ahead In this section we have introduced the notion of a class of algebras and homomorphisms between algebras in the same class. We have observed that datatype definitions in a functional programming language define an algebra, the carrier of the algebra being the datatype itself and the operations being the constructors of the datatype. We have also made the important observation that in some cases a function is uniquely characterised by its homomorphism type (the fact that it is a homomorphism combined with knowledge about its target and source algebras).

In the remaining sections of this chapter our goal is to formalise all these ideas in a way that facilitates the calculational construction of programs. Let us give an outline of what is in store.

The notion of an algebra is formalised in many textbooks on Universal Algebra. Here is an example of such a definition. This is *not* the definition we intend to use so you don't need to understand it in detail.

Σ-algebra A Σ-algebra with respect to a signature with list of operators $\Sigma = (S,\Gamma)$ is a pair (V,F) such that

- V is an S-sorted set, and
- $F = \{\gamma\colon\ \gamma \in\ \cup\Gamma\colon\ f_\gamma\}$ is a set of functions such that

$$\gamma \in \Gamma_{\langle\langle s_0,\ldots,s_{n-1}\rangle,r\rangle} \quad\Rightarrow\quad f_\gamma \;\in\; V_{s_0}\times\ldots\times V_{s_{n-1}} \to V_r$$

$$\gamma \in \Gamma_{\langle s,r\rangle} \quad\Rightarrow\quad f_\gamma \in V_s{\to}V_r$$

V is called the *carrier set* of the Σ-algebra and set F is its *operator set*.

Contrast this with the definition we are going to explain in the coming sections.

F-algebra Suppose F is a functor. Then an F-algebra is a pair (A,α) such that $\alpha \in A \leftarrow FA$.

Neither definition is complete since in the first definition the notion of a signature has not been defined, and in the second the notion of a functor hasn't been defined. In the first definition, however, it's possible to guess what the definition of a signature is and, after struggling some time with the subscripts of subscripts, it is possible to conclude that the definition corresponds to the "intuitive" notion of an algebra. The disadvantage is that the definition is grossly unwieldy. If the definitions of one's basic concepts are as complicated as this then one should give up altogether any hope that one can calculate with them.

The second definition is very compact and, as we shall see, gives an excellent basis for program construction. Its disadvantage, however, is that it is impossible to guess what the definition of a functor might be, and it is difficult to see how it corresponds to the familiar notion of an algebra. How is it possible to express the idea that an algebra consists of a set of operations? On the face of it, it would appear that an F-algebra has just one operation α. Also, how does one express the fact that the operations in an algebra have various arities?

The answer to these questions is hidden in the definition of a "functor". And, of course, if its definition is long and complicated then all the advantages of the compactness of the definition of an algebra are lost. We shall see, however, that the definition of a functor is also very compact. We shall also see that functors can be constructed from primitive functors in a systematic way. The "disjoint sum" of two functors enables one to express the idea that an algebra has a set of operations; the "cartesian product" of functors allows one to express the arity of the various operations; "constant functors" enable the expression of the existence of designated constants in an algebra. An additional major bonus is that the categorical notion of an "initial algebra" leads to a very compact and workable definition of inductively defined datatypes in a programming language. The remaining sections of this chapter thus provide a veritable arsenal of fundamental concepts whose mastery is tremendously worthwhile.

Exercise 2.1 Check the claim that $\mathsf{even} :: (\mathbb{B}, (\equiv), \mathsf{true}) \leftarrow (\mathbb{N}, (+), 0)$ is a homomorphism. Do the same for $\mathsf{foldr}\ \oplus\ e :: (A, \oplus, e) \leftarrow (List\ \mathbb{N}, (:), [\,])$.

Exercise 2.2 Give the composition of the following two homomorphisms:

$$\begin{array}{rl} (\neg) & :: (\mathbb{B}, (\wedge), \mathsf{true}) \leftarrow (\mathbb{B}, (\vee), \mathsf{false}) \\ (>0) & :: (\mathbb{B}, (\vee), \mathsf{false}) \leftarrow (\mathbb{N}, (+), \quad 0 \quad) \end{array}$$

Exercise 2.3 Show that the only automorphism on the algebra $(\mathbb{B}, (\equiv), \mathsf{true})$ is the trivial automorphism id.

Exercise 2.4 Give an example of a non-trivial automorphism on the algebra $(\mathbb{R}, (\times), 0)$.

2.2 Functors

To a first approximation, datatypes are just sets. A second approximation, which we have just seen, is that a datatype is the carrier of an algebra. In this section we identify *parameterised* datatypes with the categorical notion of functor, giving us a third approximation to what it is to be a datatype. It is in this section that we take the first steps towards a generic theory of datatypes.

Examples The best way to introduce the notion of a functor is by abstraction from a number of examples. Here are a few datatype definitions:

$$\textbf{data}\ List\ a \ = \ \mathsf{cons}\ a\ (List\ a) \ \mid\ \mathsf{nil}$$

$$\textbf{data}\ Maybe\ a \ = \ \mathsf{none} \ \mid\ \mathsf{one}\ a$$

$$\textbf{data}\ Bin\ a \ = \ \mathsf{tip}\ a \ \mid\ \mathsf{join}\ (Bin\ a)\ (Bin\ a)$$

$$\textbf{data}\ Rose\ a \ = \ \mathsf{fork}\ a\ (List(Rose\ a))$$

Each of these types can be viewed as a structured repository of information, the type of information being specified by the parameter a in the definition. Each of these types has its own map combinator. "Mapping" a function over an instance of one of these datatypes means applying the function to all the values stored in the structure without changing the structure itself. The typings of the individual map combinators is thus as follows.

$$\begin{array}{ll}
\mathsf{map}_{List} & :: (List\ a \leftarrow List\ b) \leftarrow (a \leftarrow b) \\
\mathsf{map}_{Maybe} & :: (Maybe\ a \leftarrow Maybe\ b) \leftarrow (a \leftarrow b) \\
\mathsf{map}_{Bin} & :: (Bin\ a \leftarrow Bin\ b) \leftarrow (a \leftarrow b) \\
\mathsf{map}_{Rose} & :: (Rose\ a \leftarrow Rose\ b) \leftarrow (a \leftarrow b)
\end{array}$$

Datatypes having more than one type parameter also have a map combinator, but with more arguments. For instance, defining the type of trees with leaves of type a and interior nodes of type b by

$$\textbf{data}\ Tree\ a\ b \ = \ \mathsf{leaf}\ a \ \mid\ \mathsf{node}\ (Tree\ a\ b)\ b\ (Tree\ a\ b)$$

the corresponding map combinator has type

$$\mathsf{map}_{Tree} \quad :: \quad (Tree\ a\ b \leftarrow Tree\ c\ d) \leftarrow (a{\leftarrow}c) \leftarrow (b{\leftarrow}d)$$

Given a tree of type $Tree\ c\ d$, the combinator applies a function of type $a{\leftarrow}c$ to all the leaves of the tree, and a function of type $b{\leftarrow}d$ to all the nodes, thus creating a tree of type $Tree\ a\ b$.

In general, the map combinator for an n-ary datatype maps n functions over the values stored in the datatype. (This also holds for the case that n is zero. Datatypes having *no* type parameter also have a map combinator, but with *no* functional arguments! The map in this case is the identity function on the elements of the datatype.)

Functors Defined The idea that parameterised datatypes are structured repositories of information over which arbitrary functions can be mapped is captured by the concept of a *functor*. We first explain the concept informally for unary functors. Consider the world of typed functions. Functors are the structure-respecting functions for that world. So what is the structure involved? First, that world can be viewed as a directed graph, in which the nodes are types and the arcs are functions. So, as for graphs, we require that a functor is a *pair* of mappings, one acting on types and one acting on functions, and if we overload the identifier F to denote both functions, they satisfy the typing rule:

$$\frac{f :: \quad a \leftarrow \quad b}{F f :: F a \leftarrow F b}$$

Further, functions can be composed with the operation "$\bullet$", which is associative and has neutral element the identity function, id, so this world forms a monoid algebra. Functors also respect the monoid structure:

$$\begin{array}{lcl} F\ \ (f \bullet g) & = & (F\, f) \bullet (F\, g) \\ F\ \ \mathsf{id}_a & = & \mathsf{id}_{Fa} \end{array}$$

The first of these laws says that there is no difference between mapping the composition of two functions over an F structure in one go and mapping the functions over the structure one by one. The second law says that mapping the identity function over an F structure of a's has no effect on the structure.

To be completely precise, the world of functions is not quite a monoid, since the algebra is *partial*: the meaning of $f{\bullet}g$ is only defined when this composition is well-typed, that is, when the source type of f is the target type of g. The first equality above should therefore only be applied to cases for which $f{\bullet}g$ is defined, and from now on we assume this as a tacit condition on such equations. It follows from the typing rule that then also the composition $(F\, f) \bullet (F\, g)$ is well-typed, so that is not needed as a condition.

Now, in general, an n-ary *functor* F is a pair of mappings that maps an n-tuple of types

$a_0, \ldots, a_{n-1}$ to a type $F\ a_0\ \cdots\ a_{n-1}$ and an n-tuple of functions $f_0, \ldots, f_{n-1}$ to a function $F\ f_0\ \cdots\ f_{n-1}$ in such a way that typing, composition and identity are respected:

$$\frac{f_i :: a_i \leftarrow b_i \quad \text{for } i = 0, \ldots, n-1}{F f_0 \cdots f_{n-1} :: F a_0 \cdots a_{n-1} \leftarrow F b_0 \cdots b_{n-1}}$$

$$\begin{array}{lcccl} F & (f_0 \bullet g_0) & \cdots & (f_{n-1} \bullet g_{n-1}) & = (F\, f_0 \cdots f_{n-1}) \bullet (F\, g_0 \cdots g_{n-1}) \\ F & \mathsf{id} & \cdots & \mathsf{id} & = \mathsf{id} \end{array}$$

Examples Revisited As anticipated in the introduction to this section, the pairs of mappings F (on types) and map_F (on functions) for $F = List, Maybe$, etcetera, are all functors since they satisfy the typing rule

$$\frac{f :: a \leftarrow b}{\mathsf{map}_F\, f :: F a \leftarrow F b}$$

and the functional equalities

$$\begin{array}{lcl} \mathsf{map}_F\ (f \bullet g) & = & (\mathsf{map}_F\, f) \bullet (\mathsf{map}_F\, g) \\ \mathsf{map}_F\ \mathsf{id} & = & \mathsf{id} \quad . \end{array}$$

So they are unary functors.

An example of a binary functor is the pair of mappings $Tree$ and map_{Tree} since the pair satisfies the typing rule

$$\frac{\begin{array}{c} f :: a \leftarrow c \\ g :: b \leftarrow d \end{array}}{\mathsf{map}_{Tree}\, f\, g :: F\, a\, b \leftarrow F\, c\, d}$$

and the functional equalities

$$\begin{array}{lcl} \mathsf{map}_{Tree}\ (f \bullet g)\ (h \bullet k) & = & (\mathsf{map}_{Tree}\, f\, h) \bullet (\mathsf{map}_F\, g\, k) \\ \mathsf{map}_{Tree}\ \mathsf{id}\ \mathsf{id} & = & \mathsf{id} \quad . \end{array}$$

Notational convention Conventionally, the *same* notation is used for the type mapping and the function mapping of a functor, and we follow that convention here. Moreover, when applicable, we use the name of the type mapping. So, from here on, for function f, we write $List\ f$ rather than $\mathsf{map}_{List}\ f$.

Exercise 2.5 Consider the following datatype declarations. Each defines a mapping from types to types. For example, $Error$ maps the type a to the type $Error\ a$. Extend the definition of each so that it becomes a functor.

$$\textbf{data}\ Error\ a\ =\ \mathsf{error}\ String \mid \mathsf{ok}\ a$$

$$\begin{array}{lcl} \textbf{data}\ Drawing\ a & = & \mathsf{above}\ (Drawing\ a)\ (Drawing\ a) \\ & \mid & \mathsf{beside}\ (Drawing\ a)\ (Drawing\ a) \\ & \mid & \mathsf{atom}\ a \end{array}$$

□

2.3 Polynomial Functors

Now that we have defined the notion of a functor and have seen some non-trivial examples it is time to consider more basic examples. Vital to the usefulness of the notion is that non-trivial functors can be constructed by composing more basic functors. In this section we consider the *polynomial* functors. As the name suggests, these are the functors that can be obtained by "addition" and "multiplication" possibly combined with the use of a number of "constants".

The technical terms for addition and multiplication are "disjoint sum" and "cartesian product". The use of disjoint sum enables one to capture in a *single* functor the fact that an algebra has a *set* of operations. The use of cartesian product enables one to express the fact that an operator in an algebra has an arity greater than one. We also introduce constant functors and the identity functor; these are used to express the designated constants (functions of arity zero) and unary functions in an algebra, respectively. For technical reasons, we also introduce a couple of auxiliary functors in order to complete the class of polynomial functors. We begin with the simpler cases.

The identity functor The simplest example of a functor is the *identity functor* which is the trivial combination of two identity functions, the function that maps every type to itself and the function that maps every function to itself. Although trivial, this example is important and shouldn't be forgotten. We denote the identity functor by Id.

Constant functors For the constant mapping that maps any n-tuple of arguments to the same result x we use the notation x^{K}. As is easily verified, the pair of mappings a^{K} and $\mathsf{id}_a{}^{\mathsf{K}}$, where a is some type, is also a functor. It is n-ary for all n.

Following the naming convention introduced above, we write a^{K} to denote both the mapping on types and the mapping on functions. That is, we write a^{K} where strictly we should write $\mathsf{id}_a{}^{\mathsf{K}}$. So, for functions $f_0 \ \ldots \ f_{n-1}$, $a^{\mathsf{K}}\, f_0 \ \ldots \ f_{n-1} = \mathsf{id}_a$.

A constant functor that we will use frequently is the constant functor associated with the *unit type*, $\mathsf{1}$. The unit type is the type that is denoted `()` in Haskell. It is a type having exactly one element (which element is also denoted `()` in Haskell). This functor will be denoted by $\mathsf{1}$ rather than $\mathsf{1}^{\mathsf{K}}$.

Extraction Each extraction combinator

$$\mathsf{Ex}_i^n\ z_0\ \cdots\ z_{n-1} \ = \ z_i, \quad \text{for } i \ = \ 0,\ldots,n-1$$

is an n-ary functor. The extractions that we have particular use for are the identity functor Id, which is the same as Ex_0^1, and the binary functors Ex_0^2 and Ex_1^2, for which we use the more convenient notations Par and Rec. (The reason for this choice of identifiers will become evident in chapter 3. When defining recursive datatypes like *List*, we identify a binary "pattern functor". The first parameter of the pattern functor is the parameter of the recursive datatype —and is thus called the Par parameter— and the second parameter is used as the argument for recursion —and is thus called the Rec parameter.)

The sum functor The binary sum functor + gives the "disjoint union" of two types. We write it as an infix operator. It is defined by:

$$\textbf{data}\ a+b \ = \ \mathsf{inl}\ a \ \mid \ \mathsf{inr}\ b$$

$$\begin{array}{lll} f+g & = & h \ \ \textbf{where} \\ & & h(\mathsf{inl}\ u) \ = \ \mathsf{inl}(f\ u) \\ & & h(\mathsf{inr}\ v) \ = \ \mathsf{inr}(g\ v) \end{array}$$

$$\begin{array}{lll} f \triangledown g & = & h \ \ \textbf{where} \\ & & h(\mathsf{inl}\ u) \ = \ f\ u \\ & & h(\mathsf{inr}\ v) \ = \ g\ v \end{array}$$

The datatype definition introduces both the type $a{+}b$, called the *disjoint sum* of a and b, and the two constructor functions $\mathsf{inl} :: a+b \leftarrow a$ and $\mathsf{inr} :: a+b \leftarrow b$. The name "disjoint sum" is used because $a{+}b$ is like the set union of a and b except that each element of the sets a and b is, in effect, tagged with either the label inl, to indicate that it originated in set a, or inr, to indicate that it originated in set b. In this way $a{+}a$ is different from a since it effectively contains two copies of every element in a, one with label inl and one with label inr. In particular $\mathsf{1}{+}\mathsf{1}$ has two elements. The constructors inl and inr are called *injections* and are said to *inject* elements of a and b into the respective components of $a{+}b$.

In order to extend the sum mapping on types to a functor we have to define the sum of two functions. This is done in the definition of $f{+}g$ above. Its definition is obtained by type considerations — if + is to be a functor, we require that if $f :: a{\leftarrow}b$ and $g :: c{\leftarrow}d$ then $f{+}g :: a{+}c \leftarrow b{+}d$. It is easily checked that the above definition of $f{+}g$ meets this requirement; indeed, there is no other way to do so.

In addition to defining $f{+}g$ we have defined another way of combining f and g, namely $f{\triangledown}g$, which we pronounce f "junk" g. ("Junk" is short for "junction".) As we see shortly, $f{\triangledown}g$ is more basic than $f{+}g$. The meaning of $f{\triangledown}g$ is only defined when f and g have the same target type; its source type is a disjoint sum of two types. Operationally, it inspects the label on its argument to see whether the argument originates from the left or right component of the disjoint sum. Depending on which component it is, either the function f or the function g is applied to the argument after first stripping off the label. In other words, $f{\triangledown}g$ acts like a case statement, applying f or g depending on which component of the disjoint sum the argument comes from.

The *typing rule* for $\triangledown$ is a good way of memorising its functionality:

$$\frac{\begin{array}{l} f :: c \leftarrow a \\ \quad g :: c \leftarrow \quad b \end{array}}{f \triangledown g :: c \leftarrow a+b}$$

(Haskell's prelude contains a definition of disjoint sum:

$$\textbf{data}\ Either\ a\ b\ =\ \mathsf{Left}\ a \mid \mathsf{Right}\ b$$

with *either* playing the role of $\triangledown$.)

Now that we have defined $+$ on types and on functions in such a way as to fulfill the typing requirements on a (binary) functor it remains to verify that it respects identities and composition. We do this now. In doing so, we establish a number of calculational properties that will prove to be very useful for other purposes.

Note first that the definitions of $+$ (on functions) and of $\triangledown$ can be rewritten in point-free style as the following *characterisations*:

$$\boxed{\begin{array}{lllllllll} f{+}g & = & h & \equiv & \mathsf{inl} \bullet f & = & h \bullet \mathsf{inl} & \wedge & \mathsf{inr} \bullet g = h \bullet \mathsf{inr} \\ f \triangledown g & = & h & \equiv & f & = & h \bullet \mathsf{inl} & \wedge & g = h \bullet \mathsf{inr} \end{array}}$$

This style is convenient for reasoning. For example, we can prove the *identity rule*:

$$\boxed{\mathsf{inl} \triangledown \mathsf{inr} = \mathsf{id}}$$

by calculating as follows:

$$\begin{array}{ll} & \mathsf{inl} \triangledown \mathsf{inr} \ = \ \mathsf{id} \\ \equiv & \{ \quad \text{characterisation of } \triangledown \quad \} \\ & \mathsf{id}{\bullet}\mathsf{inl} = \mathsf{inl} \ \wedge\ \mathsf{id}{\bullet}\mathsf{inr} = \mathsf{inr} \\ \equiv & \{ \quad \mathsf{id} \text{ is the identity of composition} \quad \} \\ & \mathsf{true}\ \ . \end{array}$$

If we substitute $f + g$ or $f \triangledown g$ for h in the corresponding characterisation, the left-hand sides of the equivalences become trivially true. The right-hand sides are then also true, giving the *computation rules*:

$$\boxed{\begin{array}{lllllll} (f{+}g) \bullet \mathsf{inl} & = & \mathsf{inl} \bullet f & \quad & (f{+}g) \bullet \mathsf{inr} & = & \mathsf{inr} \bullet g \\ (f \triangledown g) \bullet \mathsf{inl} & = & f & & (f \triangledown g) \bullet \mathsf{inr} & = & g \end{array}}$$

The validity of the so-called $\triangledown$-*fusion* rule:

$$\boxed{h \bullet (f \triangledown g) = (h{\bullet}f) \triangledown (h{\bullet}g)}$$

is shown by the following calculation[2]:

[2] We adopt the convention that composition has lower precedence than all other operators. Thus $h \bullet f \triangledown g$ should be read as $h \bullet (f \triangledown g)$. In the statement of the basic rules, however, we always parenthesise fully.

$$
\begin{array}{ll}
& h \bullet f \triangledown g \;=\; \alpha \triangledown \beta \\
\equiv & \{ \quad \text{characterisation of } \triangledown \quad \} \\
& h \bullet f \triangledown g \bullet \mathsf{inl} \;=\; \alpha \;\wedge\; h \bullet f \triangledown g \bullet \mathsf{inr} \;=\; \beta \\
\equiv & \{ \quad \text{computation rules for } \triangledown \quad \} \\
& h \bullet f \;=\; \alpha \;\wedge\; h \bullet g \;=\; \beta \;.
\end{array}
$$

This last calculation is a simple illustration of the way we often *derive* programs. In this case the goal is to express $h \bullet f \triangledown g$ in terms of $\triangledown$. We therefore introduce the unknowns α and β, and calculate expressions for α and β that satisfy the goal.

It is also possible to express $+$ in terms of $\triangledown$, namely by:

$$\boxed{f + g = (\mathsf{inl} \bullet f) \triangledown (\mathsf{inr} \bullet g)}$$

We derive the rhs of this rule as follows:

$$
\begin{array}{ll}
& f{+}g \;=\; \alpha \triangledown \beta \\
\equiv & \{ \quad \text{characterisation of } \triangledown \quad \} \\
& f{+}g \bullet \mathsf{inl} \;=\; \alpha \;\wedge\; f{+}g \bullet \mathsf{inr} \;=\; \beta \\
\equiv & \{ \quad \text{computation rules for } + \quad \} \\
& \mathsf{inl} \bullet f \;=\; \alpha \;\wedge\; \mathsf{inr} \bullet g \;=\; \beta \;.
\end{array}
$$

Another fusion rule is the $\triangledown$-$+$ *fusion* rule:

$$\boxed{(f + g) \bullet (h \triangledown k) = (f \bullet h) \triangledown (g \bullet k)}$$

We leave its derivation as an exercise.

These rules are useful by themselves, but they were proved to lead to the result that $+$ respects function composition:

$$\boxed{(f + g) \bullet (h + k) = (f \bullet h) + (g \bullet k)}$$

The proof is simple:

$$
\begin{array}{ll}
& f{+}g \;\bullet\; h{+}k \\
= & \{ \quad \text{definition of } + \quad \} \\
& (\mathsf{inl} \bullet f) \triangledown (\mathsf{inr} \bullet g) \;\bullet\; h{+}k \\
= & \{ \quad \triangledown\text{-}+ \text{ fusion} \quad \} \\
& (\mathsf{inl} \bullet f \bullet h) \triangledown (\mathsf{inr} \bullet g \bullet k) \\
= & \{ \quad \text{definition of } + \quad \} \\
& (f \bullet h) + (g \bullet k) \;.
\end{array}
$$

The proof that $+$ also respects id, that is,

$$\boxed{\mathsf{id} + \mathsf{id} = \mathsf{id}}$$

is also left as an exercise.

An important property that we shall use is that the mapping $\triangledown$ is injective, that is:

$$f \triangledown g = h \triangledown k \quad \equiv \quad f = h \ \wedge\ g = k$$

In the "follows-from" direction this is obvious. In the "implies" direction it follows immediately by applying $(\bullet\mathsf{inl})$ and $(\bullet\mathsf{inr})$ to both sides of the left equality. Further, the mapping is surjective (within the typing constraints): if $h :: c \leftarrow a + b$, then there exist functions $f :: c \leftarrow a$ and $g :: c \leftarrow b$ such that $h = f \triangledown g$. In fact, they can be given explicitly by $f = h \bullet \mathsf{inl}$ and $g = h \bullet \mathsf{inr}$.

The product functor While sums give a choice between values of two types, products combine two values. In Haskell product type formers are built in: to form the product of a and b, we can simply write (a, b). However, we want to introduce them as a datatype. The binary product functor $\times$ is given by:

$$\textbf{data}\ a \times b \ = \ \mathsf{pair}\ a\ b$$

$$\begin{aligned}
\mathsf{exl}(\mathsf{pair}\ u\ v) &= u \\
\mathsf{exr}(\mathsf{pair}\ u\ v) &= v
\end{aligned}$$

$$\begin{aligned}
f \times g \ = \ & h \ \ \textbf{where} \\
& h(\mathsf{pair}\ u\ v) \ = \ \mathsf{pair}\ (f\ u)\ (g\ v)
\end{aligned}$$

$$\begin{aligned}
f \vartriangle g \ = \ & h \ \ \textbf{where} \\
& h\ u \ = \ \mathsf{pair}\ (f\ u)\ (g\ u)
\end{aligned}$$

The functions $\mathsf{exl} :: a \leftarrow a \times b$ and $\mathsf{exr} :: b \leftarrow a \times b$ are called *projections* and are said to *project* a pair onto its components.

Just as for disjoint sum, we have defined $f \times g$ in such a way that it meet the type requirements on a functor. Specifically, if $f :: a \leftarrow b$ and $g :: c \leftarrow d$ then $f \times g :: a \times c \leftarrow b \times d$, as is easily checked. Also, we have defined a second combination of f and g, namely $f \vartriangle g$, which we pronounce f "split" g.

The operational meaning of $f \times g$ is easy to see. Given a pair of values, it produces a pair by applying f to the first component and g to the second component. The operational meaning of $f \vartriangle g$ is that it constructs a pair of values from a single value by applying both f and g to the given value. (In particular, $\mathsf{id} \vartriangle \mathsf{id}$ constructs a pair by "splitting" a given value into two copies of itself.)

A curious fact is the following. *All* the rules for sums are *also* valid for products under the following systematic replacements: replace $+$ by $\times$, $\triangledown$ by $\vartriangle$, inl and inr by exl and exr, and switch the components f and g of each composition $f{\bullet}g$. (In category theory this is called *dualisation*.) This gives us the *characterisations*:

$$\boxed{\begin{array}{lllllllll} f\times g & = & h & \equiv & f\bullet\mathsf{exl} & = & \mathsf{exl}\bullet h & \wedge & g\bullet\mathsf{exr} = \mathsf{exr}\bullet h \\ f\vartriangle g & = & h & \equiv & f & = & \mathsf{exl}\bullet h & \wedge & g = \mathsf{exr}\bullet h \end{array}}$$

the *identity rule*:

$$\boxed{\mathsf{exl}\vartriangle\mathsf{exr} = \mathsf{id}}$$

the *computation rules*:

$$\boxed{\begin{array}{lllllll} \mathsf{exl}\bullet(f\times g) & = & f\bullet\mathsf{exl} & \quad & \mathsf{exr}\bullet(f\times g) & = & g\bullet\mathsf{exr} \\ \mathsf{exl}\bullet(f\vartriangle g) & = & f & & \mathsf{exr}\bullet(f\vartriangle g) & = & g \end{array}}$$

the $\vartriangle$-*fusion rule*:

$$\boxed{(f\vartriangle g)\bullet h = (f{\bullet}h)\vartriangle(g{\bullet}h)}$$

$\times$ expressed in terms of $\vartriangle$:

$$\boxed{f\times g = (f{\bullet}\mathsf{exl})\vartriangle(g{\bullet}\mathsf{exr})}$$

the $\times$-$\vartriangle$-*fusion rule*:

$$\boxed{(f\times g)\bullet(h\vartriangle k) = (f{\bullet}h)\vartriangle(g{\bullet}k)}$$

and finally the fact that $\times$ *respects the* (partial) *monoid structure*:

$$\boxed{\begin{array}{ccc} (f\times g)\bullet(h\times k) & = & (f{\bullet}h)\times(g{\bullet}k) \\ \mathsf{id}\times\mathsf{id} & = & \mathsf{id} \end{array}}$$

Exercise 2.6 ($\triangledown$-$\vartriangle$ abide) Prove that, for all f, g, h and k,

$$(f\triangledown g)\vartriangle(h\triangledown k) \;=\; (f\vartriangle h)\triangledown(g\vartriangle k) \quad .$$

□

Exercise 2.7 (Abide laws) The law proved in exercise 2.6 is called the $\triangledown$-$\vartriangle$ *abide* law because of the following two-dimensional way of writing the law in which the two operators are written either *ab*ove or bes*ide* each other. (The two-dimensional way of writing is originally due to C.A.R.Hoare, the catchy name is due to Richard Bird.)

$$\begin{array}{ccc} f & \triangledown & g \\ & \vartriangle & \\ h & \triangledown & k \end{array} \;=\; \begin{array}{ccc} f & & g \\ \vartriangle & \triangledown & \vartriangle \\ h & & k \end{array}$$

What other operators abide with each other in this way? (You have already seen examples in this text, but there are also other examples from simple arithmetic.)

□

Functional Composition of Functors It is easily verified that the composition of two unary functors F and G is also a functor. By their composition we mean the pair of mappings, the first of which maps type a to $F(Ga)$ and the second maps function f to $F(Gf)$. We use juxtaposition —thus FG— to denote the composition of unary functors F and G. For example, *Maybe Rose* denotes the composition of the functors *Maybe* and *Rose*. The order of composition is important, of course. The functor *Maybe Rose* is quite different from the functor *Rose Maybe*.

It is also possible to compose functors of different arities. For instance we may want to compose a binary functor like disjoint sum with a unary functor like *List*. A simple notational device to help define such a functor is to overload the meaning of the symbol "+" and write *List*+*List*, whereby we mean the functor that maps x to $(List\ x) + (List\ x)$. Similarly we can compose disjoint sum with two unary functors F and G: we use the notation $F{+}G$ and mean the functor that maps x to $(F\ x) + (G\ x)$.

Two ways of reducing the arity of a functor are *specialisation* and *duplication*. An example of specialisation is when we turn the binary disjoint sum functor into a unary functor by specialising its first argument to the unit type. We write $1{+}\mathsf{Id}$ and mean the functor that maps type a to the type $1{+}a$, and function f to the function $\mathsf{id}_1{+}f$. Duplication means that we duplicate the argument as many times as necessary. For example, the mapping $x \mapsto x{+}x$ is a unary functor.

Both duplication and specialisation are forms of functional composition of functors. To formulate them precisely we need to extend the notion of functor so that the arity of the target of a functor may be more than one. (Up till now we have always said that a functor maps an n-tuple of types/functions to a single type/function.) Then a tuple of functors is also a functor, and, for each n, there is a duplication functor of arity n. In this way duplication and specialisation can be expressed as the composition of a functor with a tuple of functors. (In the case of specialisation, one of the functors is a constant functor.)

For our current purposes, a complete formalisation is an unnecessary complication and the ad hoc notation introduced above will suffice. Formalisations can be found in [17, 18, 36, 21].

Polynomial functors A functor built only from constants, extractions, sums, products and composition is called a *polynomial* functor.

An example of a polynomial functor is *Maybe* introduced in section 2.2. Recalling its definition:

$$\textbf{data}\ Maybe\ a\ =\ \mathsf{none}\ \mid\ \mathsf{one}\ a$$

we see that, expressed in the notation introduced above, $Maybe = 1{+}\mathsf{Id}$

The remaining examples introduced in section 2.2 are not polynomial because they are defined recursively. We need one more mechanism for constructing functors. That is the topic of the next section.

Exercise 2.8 Consider the mapping $Square$ that takes a type a to $a \times a$ and a function f to $f \times f$. Check that $Square$ is a functor.

Exercise 2.9 In checking that something is a functor, we must check that it respects composition *and* identity. The last part may not be omitted, as is shown by the existence of "almost-functors". Call F an almost-functor when F is a pair of mappings on types and functions (just like true functors) that respects typing and composition, but fails to respect identity: $F\mathsf{id} \neq \mathsf{id}$. Can you find a simple example of such an almost-functor? (Hint: Look at constant mappings.)

Exercise 2.10 If $\mathsf{inl} :: a + b \leftarrow a$ and $\mathsf{inr} :: a + b \leftarrow b$, what is the typing of id in the identity rule $\mathsf{inl} \triangledown \mathsf{inr} = \mathsf{id}$?

Exercise 2.11 Complete the verification that $+$ is a functor by proving the $\triangledown$-$+$ fusion rule and the identity rule ($\mathsf{id} + \mathsf{id} = \mathsf{id}$). In the calculation you may use all the other rules stated before these two rules.

□

2.4 Datatypes Generically

By now the notion of a functor should be becoming familiar to you. Also, it should be clear how to extend the definition of non-inductive datatypes not involving function spaces to a polynomial functor. In this section we take the step to inductively defined datatypes.

The basic idea is that an inductively defined datatype is a fixed point of a functor, which functor we call the *pattern* functor of the datatype. For the simplest examples (such as the natural numbers) the pattern functor is polynomial but for more complicated examples (like the $Rose$ datatype) it is not. We therefore need to extend the class of functors we can define beyond the polynomial functors to the so-called *regular* functors by adding the *type* functors. The basic technical device to achieve this is the *catamorphism*, which is a generalisation of the fold function on lists.

We begin by discussing pattern functors following which we can, at long last, define the notion of an F-algebra. Catamorphisms form the next —substantial— topic, following which we introduce type functors and the class of regular functors.

Pattern functors and recursion We first look at a simple inductively (= recursively) defined datatype, that of the Peano naturals, which we also saw in section 2.1:

$$\textbf{data}\ Nat\ =\ \mathsf{succ}\ Nat\ \mid\ \mathsf{zero}$$

There is only one number zero, which we can make explicit by:

$$\textbf{data}\ Nat\ =\ \mathsf{succ}\ Nat\ \mid\ \mathsf{zero}\ \mathsf{1}$$

Instead of fancy constructor function names like succ and zero we now employ boring standard ones:

$$\textbf{data}\ Nat\ =\ \mathsf{inl}\ Nat\ \mid\ \mathsf{inr}\ \mathsf{1}$$

The choice here is that afforded by sum, so we replace this by

$$\textbf{data}\ Nat\ =\ \mathsf{in}(Nat\ +\ \mathsf{1})$$

in which there is one explicit constructor function left, called "in".

Now note that Nat occurs both on the left and the right of the datatype definition (which is why it is called an inductively defined or recursive datatype). In order to view this as a fixed point definition, let us abstract from Nat on the right side replacing it by the variable z. In this way we are led to consider the unary functor N defined by

$$N\ z\ =\ z+\mathsf{1}$$

(Note that, although we have only defined N explicitly on types, we understand its extension to a functor. Using the notations introduced earlier, this functor is expressed as $N\ =\ \mathsf{Id}+\mathsf{1}^{\mathsf{K}}$.) The functor N captures the pattern of the inductive formation of the Peano naturals. The point is that we can use this to rewrite the definition of Nat to

$$\textbf{data}\ Nat\ =\ \mathsf{in}(N\ Nat)$$

Apparently, the *pattern functor* N uniquely determines the datatype Nat. Whenever F is a unary polynomial functor, as is the case here, a definition of the form $\textbf{data}\ Z\ =\ \mathsf{in}(F\,Z)$ uniquely determines Z.

We need a notation to denote the datatype Z that is obtained, and write $Z\ =\ \mu F$. So $Nat\ =\ \mu N$. Replacing Z by μF in the datatype definition, and adding a subscript to the single constructor function in in order to disambiguate it, we obtain:

$$\boxed{\textbf{data}\ \mu F\ =\ \mathsf{in}_F(F\ \mu F)}$$

Now in_F is a generic function, with typing

$$\mathsf{in}_F :: \mu F \leftarrow F\ \mu F$$

We can "reconstruct" the original functions succ and zero by defining:

$$\begin{array}{lcll} \mathsf{succ} & = & \mathsf{in}{\bullet}\mathsf{inl} & :: Nat \leftarrow Nat \\ \mathsf{zero} & = & \mathsf{in}{\bullet}\mathsf{inr} & :: Nat \leftarrow \ 1 \end{array}$$

Conversely, $\mathsf{in} :: Nat \leftarrow Nat$ is then of course

$$\mathsf{in} = \mathsf{succ} \triangledown \mathsf{zero}$$

Playing the same game on the definition of $List$ gives us:

$$\textbf{data}\ List\ a\ =\ \mathsf{in}((a \times List\ a)\ +\ 1)$$

Replacing the datatype being defined, $List\ a$, systematically by z, we obtain the "equation"

$$\textbf{data}\ z\ =\ \mathsf{in}((a \times z)\ +\ 1)$$

Thus, we see that the pattern functor here is $(z \mapsto (a \times z)\ +\ 1)$. It has a parameter a, which we make explicit by putting

$$L\ a\ =\ (z \mapsto (a \times z)\ +\ 1)$$

Now $List\ a\ =\ \mu(L\ a)$, or, abstracting from a:

$$List\ =\ (a \mapsto \mu(L\ a))$$

Exercise 2.12 What is the pattern functor for Bin? Is it polynomial? What is the pattern functor for Rose? Is it polynomial?

□

F-algebras Before we traded in the names of the constructor functions for the uniform 'in', we saw that the algebra naturally corresponding to the datatype Nat, together with the generic algebra of its class, were:

$$\begin{array}{ll} (Nat, \mathsf{succ}, \mathsf{zero}), & \text{with}\ \ \mathsf{succ} :: Nat \leftarrow Nat,\ \ \mathsf{zero} :: Nat \leftarrow 1 \\ (\ A\ ,\ f\ ,\ e\ ), & \text{with}\ \ f :: A \leftarrow A,\ \ e :: A \leftarrow 1 \end{array}$$

Using 'in', this should be replaced by:

$$\begin{array}{ll} (Nat, \mathsf{in}), & \text{with}\ \ \mathsf{in} :: Nat \leftarrow Nat + 1 \\ (\ A\ , \varphi), & \text{with}\ \ \varphi :: A \leftarrow A + 1 \end{array}$$

in which the relation between φ and the pair (f, e) is, of course,

$$\begin{array}{lcl} \varphi & = & f \triangledown e \\ f & = & \varphi{\bullet}\mathsf{inl} \\ e & = & \varphi{\bullet}\mathsf{inr} \end{array}$$

Using the pattern functor N, we can also write:

$$\begin{array}{ll} (Nat, \mathsf{in}), & \text{with}\ \ \mathsf{in} :: Nat \leftarrow N\,Nat \\ (\ A\ , \varphi), & \text{with}\ \ \varphi :: A \leftarrow N\ A \end{array}$$

In general, for a functor F, an algebra (A, φ) with $\varphi :: A \leftarrow FA$ is called an *F-algebra* and A is called the *carrier* of the algebra. So Nat is the carrier of an N-algebra, and likewise $List\ a$ is the carrier of an $(L\,a)$-algebra.

Catamorphisms In the class of F-algebras, a homomorphism $h :: (A, \varphi) \leftarrow (B, \psi)$ is a function $h :: A \leftarrow B$ that satisfies:

$$h \bullet \psi = \varphi \bullet F h$$

This can be expressed in a diagram:

$$\begin{tikzcd}
A & FA \arrow[l, "\varphi"'] \\
B \arrow[u, "h"] & FB \arrow[l, "\psi"] \arrow[u, "Fh"'] \arrow[ul, phantom, "\smile"]
\end{tikzcd}$$

The smiley face[3] signifies that the diagram *commutes*: the two paths to A from FB are equivalent.

A specific example of such a diagram is given by the homomorphism even from the natural numbers to the booleans:

$$\begin{tikzcd}
\mathsf{Bool} & \mathsf{Bool}+1 \arrow[l, "\mathsf{not} \triangledown \mathsf{true}"'] \\
\mathsf{Nat} \arrow[u, "\mathsf{even}"] & \mathsf{Nat}+1 \arrow[l, "\mathsf{succ} \triangledown \mathsf{zero}"] \arrow[u, "\mathsf{even}+1"'] \arrow[ul, phantom, "\smile"]
\end{tikzcd}$$

Rather than use such a diagram, the standard way of defining a function on an inductive datatype is by "pattern matching" on the argument, giving a clause for each constructor function. For the naturals, the typical definition has this form:

$$\textbf{data}\ Nat \ = \ \mathsf{succ}\ Nat \mid \mathsf{zero}$$

$$\begin{aligned}
h\,(\mathsf{succ}\,n) &= f\ (h\,n) \\
h\ \mathsf{zero} &= e
\end{aligned}$$

For example, the function even is defined by the equations:

$$\begin{aligned}
\mathsf{even}\ (\mathsf{succ}\ n) &= \mathsf{not}\ (\mathsf{even}\ n) \\
\mathsf{even}\ \mathsf{zero} &= \mathsf{true}
\end{aligned}$$

(Exercise 2.13 asks you to show that these two equations are equivalent to the commuting diagram above.) For lists, the typical pattern-matching has the form:

[3]Like the cheshire cat, only the smile is visible at the moment! Our apologies.

$$\begin{array}{lll} \textbf{data}\ \mathit{List}\ a & = & \mathsf{cons}\ a\ (\mathit{List}\ a) \mid \mathsf{nil} \end{array}$$

$$\begin{array}{lll} h\,(\mathsf{cons}\,x\,xs) & = & x\ \oplus\ h\,xs \\ h\ \mathsf{nil} & = & e \end{array}$$

In these definitions, the function being defined, h, is "pushed down" recursively to the components to which the constructor functions are applied. The effect is to replace the constructor functions by the corresponding arguments in the definition of h — in the case of the natural numbers, succ is replaced by f and zero by e, and in the case of lists cons is replaced by $\oplus$ and nil by e.

For the naturals, the function h defined above is determined *uniquely* by f and e. Likewise, for lists, h is *uniquely* determined by $\oplus$ and e, and there is a standard notation for the function thus defined, namely $\mathsf{foldr}\ \oplus\ e$. Generalizing this, we get the following:

$$\textbf{data}\ \mu F\ =\ \mathsf{in}(F\ \mu F)$$

$$h\,(\mathsf{in}\,xs)\ =\ \varphi\,((F\,h)\,xs)$$

in which simple typing considerations show that φ has to have a typing of the form $A \leftarrow FA$, and then h has the typing $A \leftarrow \mu F$; in other words, φ is the operation of some F-algebra whose carrier is the target type of h. The function h thus defined is uniquely determined by φ. We call such functions *catamorphisms* and use the following notation: $h = (\!|\varphi|\!)$. So $(\!|\_|\!)$ is defined by:

$$\begin{array}{ll} (\!|\varphi|\!) & = h\ \textbf{where} \\ & \quad h\,(\mathsf{in}\,xs)\ =\ \varphi\,((F\,h)\,xs) \end{array}$$

In words, when catamorphism $(\!|\varphi|\!)$ is applied to a structure of type μF, this means it is applied recursively to the components of the structure, and the results are combined by applying its "body" φ. Specialised to lists, the $(\!|\_|\!)$-combinator becomes foldr. The importance of having generic catamorphisms is that they embody a closed expression for a familiar inductive definition technique and thereby allow the generic expression of important programming rules.

Exercise 2.13 Show that the single equation

$$\mathsf{even} \bullet \mathsf{succ}\triangledown\mathsf{zero}\ \ =\ \ \mathsf{not}\triangledown\mathsf{true} \bullet \mathsf{even}{+}1$$

is equivalent to the two equations

$$\begin{array}{lll} \mathsf{even}\ (\mathsf{succ}\ n) & = & \mathsf{not}\ (\mathsf{even}\ n) \\ \mathsf{even}\ \mathsf{zero} & = & \mathsf{true}\quad . \end{array}$$

□

Initial Algebras Catamorphisms enjoy a number of attractive calculational properties which we now discuss.

We start with giving the *typing rule* for $(\!|\_|\!)$:

$$\boxed{\dfrac{\varphi \;::\; a \leftarrow Fa}{(\!|\varphi|\!) \;::\; a \leftarrow \mu F}}$$

Taking the definition

$$h\,(\mathsf{in}\,xs) \;=\; \varphi\,((Fh)\,xs)$$

we can rewrite this equivalently as:

$$(h \bullet \mathsf{in})\,xs \;=\; (\varphi \bullet Fh)\,xs$$

or, abstracting from xs:

$$h \bullet \mathsf{in} \;=\; \varphi \bullet Fh$$

This functional equation in h has a unique solution (provided that φ is a total function, and that h is only applied to finite datavalues), so we conclude that $(\!|\varphi|\!)$ is *characterised* by

$$\boxed{h = (\!|\varphi|\!) \;\;\equiv\;\; h \bullet \mathsf{in} \;=\; \varphi \bullet Fh}$$

The right-hand side of this equivalence states that h is a homomorphism, and if A is the carrier of φ, we can also express this characterisation as:

$$h = (\!|\varphi|\!) \;\;\equiv\;\; h :: (A, \varphi) \leftarrow (\mu F, \mathsf{in})$$

In words, every F-algebra is the target algebra of a unique homomorphism with $(\mu F, \mathsf{in})$ as its source algebra, and the catamorphisms consist of these unique homomorphisms. Source algebras that have the property that there is a unique homomorphism to any target algebra are known as *initial algebras*. So $(\mu F, \mathsf{in})$ is an initial algebra. It is easy to prove that all initial algebras in a given algebra class are isomorphic.

The following diagram expresses the fact that $(\!|\varphi|\!) :: (A, \varphi) \leftarrow (\mu F, \mathsf{in})$ (but not the uniqueness):

$$\begin{tikzcd}
A & FA \arrow[l, "\varphi"'] \\
\mu F \arrow[u, "(\!|\varphi|\!)"] \arrow[ur, phantom, "\smile"] & F\,\mu F \arrow[l, "\mathsf{in}"] \arrow[u, "F(\!|\varphi|\!)"']
\end{tikzcd}$$

In formula form we get the *computation rule* for catamorphisms:

$$\boxed{(\!|\varphi|\!)\bullet\mathsf{in} \;=\; \varphi\bullet F(\!|\varphi|\!)}$$

The function in is itself an F-algebra, so $(\!|\mathsf{in}|\!)$ is defined. What is it? By substituting $(A,\varphi) := (\mu F,\mathsf{in})$ in the last equivalence above, we obtain:

$$h = (\!|\mathsf{in}|\!) \quad\equiv\quad h :: (\mu F,\mathsf{in}) \leftarrow (\mu F,\mathsf{in})$$

But we know that $\mathsf{id} :: (\mu F,\mathsf{in}) \leftarrow (\mu F,\mathsf{in})$! The conclusion is the *identity rule* for catamorphisms:

$$\boxed{(\!|\mathsf{in}|\!) = \mathsf{id}_{\mu F}}$$

This generalises the equality for lists: $\mathsf{foldr\ cons\ nil} \;=\; \mathsf{id}_{List}$.

Further properties of catamorphisms The identity rule is easy to remember if one thinks of a catamorphism as a function that replaces the constructor functions of the datatype by the supplied arguments. Thus $\mathsf{foldr\ cons\ nil}$ is the identity function on lists because cons is replaced by cons and nil is replaced by nil. In general, $(\!|\mathsf{in}|\!)$ replaces all occurrences of in by itself in an element of the datatype μF.

The identity rule is surprisingly important. As an illustration of its importance, we prove that in is a bijection between μF and $F\mu F$. That is, we use the rule to construct a function out of type $F\mu F \leftarrow \mu F$ such that $\mathsf{in}\bullet\mathsf{out} = \mathsf{id}_{\mu F}$ and $\mathsf{out}\bullet\mathsf{in} = \mathsf{id}_{F\mu F}$. Our calculation starts with the first requirement and derives a candidate for out in a systematic way:

$$
\begin{array}{ll}
& \mathsf{in}\bullet\mathsf{out} = \mathsf{id}_{\mu F} \\
\equiv & \qquad \{ \qquad \text{identity rule} \quad \} \\
& \mathsf{in}\bullet\mathsf{out} = (\!|\mathsf{in}|\!) \\
\equiv & \qquad \{ \qquad \text{catamorphism characterisation} \quad \} \\
& \mathsf{in}\bullet\mathsf{out}\bullet\mathsf{in} = \mathsf{in}\bullet F(\mathsf{in}\bullet\mathsf{out}) \\
\Leftarrow & \qquad \{ \qquad \text{cancel } \mathsf{in}\bullet \text{ from both sides} \quad \} \\
& \mathsf{out}\bullet\mathsf{in} = F(\mathsf{in}\bullet\mathsf{out}) \\
\equiv & \qquad \{ \qquad F \text{ respects composition} \quad \} \\
& \mathsf{out}\bullet\mathsf{in} \;=\; F\mathsf{in}\bullet F\mathsf{out} \\
\equiv & \qquad \{ \qquad \text{catamorphism characterisation} \quad \} \\
& \mathsf{out} = (\!|F\mathsf{in}|\!) \quad .
\end{array}
$$

This completes the first step in the calculation: we have derived the candidate $(\!|F\mathsf{in}|\!)$ for out.

Note that the identity rule is not used to *simplify* $(\!|\mathsf{in}|\!)$ to $\mathsf{id}_{\mu F}$ in this calculation; rather, it is used in quite the opposite way to *complicate* $\mathsf{id}_{\mu F}$ to $(\!|\mathsf{in}|\!)$. There is a tendency to view algebraic properties as left-to-right rewrite rules, where the left side is the complicated side and the right side is its simplified form. Calculations that use the rules in this way are straightforward and do not require insight. On the other hand, calculations (such as the one above) which include at least one complication step are relatively difficult and do require insight. The importance of the identity rule for catamorphisms is its use in introducing a catamorphism into a calculation. It can require ingenuity to use because it involves replacing an identity function which is not visible. That is, a step in a calculation may involve replacing some composition $f{\bullet}g$ by $f{\bullet}(\!|\mathsf{in}|\!){\bullet}g$, the invisible intermediate step being to replace $f{\bullet}g$ by $f{\bullet}\mathsf{id}_{\mu F}{\bullet}g$. This is valid if f has source μF (equivalently, g has target μF) so it is important to be aware of the types of the quantities involved.

To complete the calculation we have to check that the candidate $(\!|F\mathsf{in}|\!)$ we have derived for out satisfies the second requirement on out. That is, we have to verify that $(\!|F\mathsf{in}|\!){\bullet}\mathsf{in} = \mathsf{id}_{F\mu F}$. This is an exercise in the use of the computation rule which we leave to the reader (specifically, exercise 2.14).

As another illustration of the use of the properties of catamorphisms we derive a condition under which it is possible to fuse a post-composed function with a catamorphism. The goal of the calculation is to eliminate the catamorphism brackets from the equation.

$$
\begin{array}{ll}
 & h{\bullet}(\!|\varphi|\!) = (\!|\psi|\!) \\
\equiv & \qquad \{ \qquad \text{characterisation of } (\!|\psi|\!) \quad \} \\
 & h{\bullet}(\!|\varphi|\!){\bullet}\mathsf{in} = \psi{\bullet}F(h{\bullet}(\!|\varphi|\!)) \\
\equiv & \qquad \{ \qquad \text{computation rule for } (\!|\varphi|\!) \quad \} \\
 & h{\bullet}\varphi{\bullet}F(\!|\varphi|\!) = \psi{\bullet}F(h{\bullet}(\!|\varphi|\!)) \\
\equiv & \qquad \{ \qquad F \text{ respects composition} \quad \} \\
 & h{\bullet}\varphi{\bullet}F(\!|\varphi|\!) = \psi{\bullet}Fh{\bullet}F(\!|\varphi|\!) \\
\Leftarrow & \qquad \{ \qquad \text{cancel } {\bullet}F(\!|\varphi|\!) \text{ from both sides} \quad \} \\
 & h{\bullet}\varphi = \psi{\bullet}Fh \quad .
\end{array}
$$

So we have derived the $(\!|\_|\!)$-*fusion rule*:

$$\boxed{h \bullet (\!|\varphi|\!) \;=\; (\!|\psi|\!) \quad \Leftarrow \quad h \bullet \varphi \;=\; \psi \bullet Fh}$$

Note that the condition states that h is a homomorphism. So the rule states that composing a homomorphism after a catamorphism is a catamorphism.

The way this rule is typically used is that we want to fuse a given function h into a given catamorphism $(\![\varphi]\!)$, for example to improve efficiency. In order to do so, we try to solve the equation $h{\bullet}\varphi \;=\; \psi{\bullet}Fh$ for the unknown ψ. If we find a solution, we know that the answer is $(\![\psi]\!)$.

An example We show this in action on a simple example: $\mathsf{sum}{\bullet}\mathsf{concat}$ on lists of lists of integers. Recall that the pattern function of $List$ Int is

$$L\ \mathsf{Int} \;=\; (z \mapsto (\mathsf{Int} \times z) \;+\; \mathsf{1}) \quad .$$

By definition, $\mathsf{concat} = (\![(+\!\!+) \triangledown \mathsf{nil}]\!)$, so we try to fuse sum and concat into a catamorphism. Applying the fusion rule we have:

$$\begin{array}{ll}
& \mathsf{sum}{\bullet}\mathsf{concat} = (\![\psi]\!) \\
\Leftarrow & \qquad \{ \qquad \mathsf{concat} = (\![(+\!\!+) \triangledown \mathsf{nil}]\!), \text{ fusion} \quad \} \\
& \mathsf{sum} \bullet (+\!\!+) \triangledown \mathsf{nil} \;\;=\;\; \psi \bullet (L\ \mathsf{Int})\ \mathsf{sum} \quad .
\end{array}$$

Now, the pattern functor $(L\ \mathsf{Int})$ is a disjoint sum of two functors. Also, the composition on the left side can be fused together:

$$\begin{array}{ll}
& \mathsf{sum} \bullet (+\!\!+) \triangledown \mathsf{nil} \\
= & \qquad \{ \qquad \triangledown \text{ fusion} \quad \} \\
& (\mathsf{sum} \bullet (+\!\!+)) \triangledown (\mathsf{sum} \bullet \mathsf{nil}) \quad .
\end{array}$$

This suggests that we should try instantiating ψ to $\alpha \triangledown \beta$ for some α and β. In this way, we get:

$$\begin{array}{ll}
& \mathsf{sum}{\bullet}\mathsf{concat} = (\![\psi]\!) \\
\Leftarrow & \qquad \{ \qquad \text{two steps above, definition of } (L\ \mathsf{Int}) \quad \} \\
& (\mathsf{sum} \bullet (+\!\!+)) \triangledown (\mathsf{sum} \bullet \mathsf{nil}) \;\;=\;\; \psi \;\bullet\; (\mathsf{id} \times \mathsf{sum}) + \mathsf{id} \\
\equiv & \qquad \{ \qquad \text{postulate } \psi = \alpha \triangledown \beta, \text{ fusion} \quad \} \\
& (\mathsf{sum} \bullet (+\!\!+)) \triangledown (\mathsf{sum} \bullet \mathsf{nil}) \;\;=\;\; (\alpha \bullet \mathsf{id}{\times}\mathsf{sum}) \triangledown (\beta{\bullet}\mathsf{id}) \\
\equiv & \qquad \{ \qquad \triangledown \text{ is injective, simplification} \quad \} \\
& \mathsf{sum} \bullet (+\!\!+) \;\;=\;\; \alpha \bullet \mathsf{id}{\times}\mathsf{sum} \;\;\wedge\;\; \mathsf{sum} \bullet \mathsf{nil} \;\;=\;\; \beta \quad .
\end{array}$$

We now continue with each conjunct in turn.

$$\begin{array}{ll}
& \mathsf{sum} \bullet (+\!\!+) \\
= & \qquad \{ \qquad \text{property of summation} \quad \} \\
& (+) \;\bullet\; \mathsf{sum}{\times}\mathsf{sum} \\
= & \qquad \{ \qquad \times \text{ is a binary functor} \quad \} \\
& (+) \bullet \mathsf{sum}{\times}\mathsf{id} \bullet \mathsf{id}{\times}\mathsf{sum} \quad .
\end{array}$$

And thus we have found that $\alpha = (+) \bullet \mathsf{sum}{\times}\mathsf{id}$ satisfies $\mathsf{sum}{\bullet}(+\!\!+) = \alpha \bullet \mathsf{id}{\times}\mathsf{sum}$. The second conjunct is easy: $\mathsf{sum}{\bullet}\mathsf{nil} = \mathsf{zero}$.

Combining everything, we have found that

$$\mathsf{sum}{\bullet}\mathsf{concat} \quad = \quad (\!|((+) \bullet \mathsf{sum}{\times}\mathsf{id}) \triangledown \mathsf{zero}|\!)$$

or, expressed in a more familiar style:

$$\begin{array}{l} \mathsf{sum}{\bullet}\mathsf{concat} = \mathsf{foldr} \ \odot \ 0 \ \textbf{where} \\ \qquad\qquad\qquad xs \odot y \ = \ \mathsf{sum}\ xs \ + \ y \end{array}$$

This derivation was not generic but specific for lists of lists. Meertens [36] shows how to do this generically, and also that the generic solution is much simpler to obtain than this specific one.

Exercise 2.14 We calculated above that $\mathsf{out} = (\!|F\mathsf{in}|\!)$ satisfies $\mathsf{in}{\bullet}\mathsf{out} = \mathsf{id}_{\mu F}$. Verify that $\mathsf{out}{\bullet}\mathsf{in} = \mathsf{id}_{F\mu F}$.

□

Exercise 2.15 Suppose that (A, φ) is an initial F-algebra. Prove that (A, φ) is isomorphic to $(\mu F, \mathsf{in})$. *Hint.* Consider the unique homomorphism $h :: (\mu F, \mathsf{in}) \leftarrow (A, \varphi)$.

Exercise 2.16 Consider the datatype $Bin\ a$ for some arbitrary type a. The pattern functor for this type is F where $Ff = id_a + (f{\times}f)$. Catamorphisms over this type take the form $(\!|f \triangledown (\odot)|\!)$ where f is a function and $(\odot)$ is a binary operator.

Define a catamorphism that counts the number of tips in a Bin. Define, in addition, a catamorphism that counts the number of joins in a Bin. Use the fusion rule for catamorphisms to determine a relation between the number of tips and the number of joins in a Bin. That is, derive the definition of a function f such that

$$f \bullet NoOfTips \quad = \quad NoOfJoins \ .$$

□

Banana split In this subsection we demonstrate the beauty of generic programming. We solve the following problem. Given are two catamorphisms $(\!|f|\!) \ :: \ a \leftarrow \mu F$ and $(\!|g|\!) \ :: \ b \leftarrow \mu F$, and we need to have a function that returns the combined result of both. One solution is the program $(\!|f|\!) \vartriangle (\!|g|\!)$, but this can be inefficient since, computationally, the source data value is traversed twice, once for each of the two catamorphisms. So the question we want to solve is: can we combine these two into a single catamorphism $(\!|\chi|\!)$?

This generic problem is motivated by our knowledge of specific cases. Take, for example, the problem of finding both the sum and the product of a list of numbers. The sum can of course be expressed as a catamorphism —it is the catamorphism $(\![\mathsf{add} \triangledown 0]\!)$, where add is ordinary addition of real numbers— . Similarly the product function is a catamorphism, namely $(\![\mathsf{mul} \triangledown 1]\!)$, where mul is the ordinary multiplication of real numbers. Equally obvious is that it should be possible to combine the sum and product of a list of numbers into one catamorphism. After all, the function $sp = \mathsf{sum} \vartriangle \mathsf{product}$ is straightforward to express as a fold in Haskell:

$$
\begin{array}{l}
sp \;=\; \mathsf{foldr}\ \odot\ e\ \mathbf{where} \\
\qquad x \odot (\mathsf{pair}\ u\ v) \;=\; \mathsf{pair}\ (x+u)\ (x \times v) \\
\qquad e \;=\; \qquad\qquad\quad \mathsf{pair}\ 0\ 1
\end{array}
$$

We can try to derive this special case in our calculus but more effective is to derive the solution to the generic problem. The benefit is not only that we then have a very general result that can be instantiated in lots of ways (one of which is the $\mathsf{sum} \vartriangle \mathsf{product}$ problem), but also that the derivation is much simpler because it omits irrelevant detail.

We begin the calculation of χ as follows:

$$
\begin{array}{ll}
& (\![f]\!) \vartriangle (\![g]\!) \;=\; (\![\chi]\!) \\
\equiv & \quad \{ \quad \text{There is a choice here. We can either use the} \\
& \qquad\quad \text{characterisation of } (\![f]\!) \text{ or the characterisation of } f \vartriangle g. \\
& \qquad\quad \text{For no good reason, we choose the latter.} \quad \} \\
& (\![f]\!) \;=\; \mathsf{exl} \bullet (\![\chi]\!) \;\;\wedge\;\; (\![g]\!) \;=\; \mathsf{exr} \bullet (\![\chi]\!) \;\; .
\end{array}
$$

This first step involves a difficult choice. At this point in time there is no reason why the use of one characterisation is preferable to the other (since both are equivalences). In fact, choosing to use the characterisation of $(\![f]\!)$ first does lead to a successful calculation of χ of a similar length. We leave it as an exercise.

We now have to satisfy two conjuncts. Since the two conjuncts are symmetrical we proceed with just the first.

$$
\begin{array}{ll}
& (\![f]\!) \;=\; \mathsf{exl} \bullet (\![\chi]\!) \\
\equiv & \quad \{ \quad \text{catamorphism characterisation} \quad \} \\
& f \bullet F(\mathsf{exl} \bullet (\![\chi]\!)) \;=\; \mathsf{exl} \bullet (\![\chi]\!) \bullet \mathsf{in} \\
\equiv & \quad \{ \quad F \text{ respects composition,} \\
& \qquad\quad \text{catamorphism computation} \quad \} \\
& f \bullet F\mathsf{exl} \bullet F(\![\chi]\!) \;=\; \mathsf{exl} \bullet \chi \bullet F(\![\chi]\!) \\
\Leftarrow & \quad \{ \quad \text{cancellation of } (\bullet F(\![\chi]\!)), \\
& \qquad\quad \bullet \quad \chi \;:=\; \alpha \vartriangle \beta \;. \quad \}
\end{array}
$$

$$
\begin{array}{lcl}
& f \bullet F\mathsf{exl} \;=\; \mathsf{exl} \;\bullet\; \alpha \vartriangle \beta \\
\equiv & \{ \quad \vartriangle \text{ computation} \quad \} \\
& f \bullet F\mathsf{exl} \;=\; \alpha \;.
\end{array}
$$

The crucial step here is the "$\Leftarrow$" step in which we postulate the form of the solution (indicated by the bullet), the motivation being the step that immediately follows.

In summary we have calculated that

$$(\!|f|\!) \;=\; \mathsf{exl} \bullet (\!|\chi|\!) \quad\Leftarrow\quad \chi = \alpha \vartriangle \beta \;\wedge\; \alpha \;=\; f \bullet F\mathsf{exl} \;.$$

Similarly,

$$(\!|g|\!) \;=\; \mathsf{exr} \bullet (\!|\chi|\!) \quad\Leftarrow\quad \chi = \alpha \vartriangle \beta \;\wedge\; \beta \;=\; g \bullet F\mathsf{exr} \;.$$

Putting everything together, we conclude that

$$(\!|f|\!) \vartriangle (\!|g|\!) \;=\; (\!|(f \bullet F\mathsf{exl}) \vartriangle (g \bullet F\mathsf{exr})|\!) \;.$$

This is affectionately called the *banana-split* theorem (because the brackets denoting a catamorphism look like bananas, and $\vartriangle$ operator is pronounced "split").

Exercise 2.17 Calculate χ but start by using the characterisation of $(\!|f|\!)$. In other words, calculate χ as a solution of the equation

$$(\!|f|\!) \vartriangle (\!|g|\!) \;\bullet\; \mathsf{in} \;=\; \chi \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \;.$$

(You may find that you get a solution that is equivalent to the one above but not syntactically identical.)

□

Type functors In general, a binary functor gives rise to a new functor by a combination of parameterisation and constructing an initial algebra. For example, the binary pattern functor L that maps x and y to $(x{\times}y){+}1$ gives rise to the functor $List$. Such functors are called *type functors*. Here we show how this is done.

For greater clarity we will use an infix notation for binary functors. Suppose that $\oslash$ is a binary functor, which we write as an infix operator. That is, for types a and b, $a{\oslash}b$ is a type and, for functions $f \;::\; a{\leftarrow}b$ and $g \;::\; c{\leftarrow}d$, $f{\oslash}g$ is a function of type $a{\oslash}c \leftarrow b{\oslash}d$. Suppose a is an arbitrary type. Then the pair of mappings $b \;\mapsto\; a{\oslash}b$ and $f \;\mapsto\; \mathsf{id}_a{\oslash}f$ is a functor (the functor formed by specialising the first operand of $\oslash$ to the constant functor a^{K}) We denote this functor by $(a\oslash)$ and call it a *parameterised* functor.

Now, since $(a\oslash)$ is a unary functor, we can consider an initial $(a\oslash)$-algebra with carrier

$\mu(a\oslash)$. Abstracting from a we have constructed a mapping from types to types. Let us introduce a special notation for this mapping:

$$\boxed{\tau(\oslash) \;=\; (a \mapsto \mu(a\oslash))}$$

So $List = \tau(L)$, with L the binary functor defined above.

For $\tau(\oslash)$ to be a functor, we need, in addition to the action on types, an action on functions, which has to satisfy, for a function $f :: a \leftarrow b$,

$$\tau(\oslash)\ f \;::\; \tau(\oslash)\ a \leftarrow \tau(\oslash)\ b \;.$$

We derive a candidate for $\tau(\oslash)\ f$ from type considerations. In the calculation, catamorphisms are $(b\oslash)$ catamorphisms and $\mathsf{in}_{a\oslash}$ is an initial $(a\oslash)$-algebra.

$$\begin{array}{ll}
 & \tau(\oslash)\ f \;::\; \tau(\oslash)\ a \leftarrow \tau(\oslash)\ b \\
\equiv & \qquad \{ \quad \text{definition of } \tau(\oslash) \text{ on types} \quad \} \\
 & \tau(\oslash)\ f \;::\; \mu(a\oslash) \;\leftarrow\; \mu(b\oslash) \\
\Leftarrow & \qquad \{ \quad \bullet \;\; \tau(\oslash)\ f \;:=\; (\!|\varphi|\!) \text{ , typing rule for } (\!|_|\!) \quad \} \\
 & \varphi \;::\; \mu(a\oslash) \;\leftarrow\; b \oslash \mu(a\oslash) \\
\Leftarrow & \qquad \{ \quad \bullet \;\; \varphi \;:=\; \mathsf{in}_{a\oslash} \bullet \psi \text{ , type of } \mathsf{in} \quad \} \\
 & \psi \;::\; a \oslash \mu(a\oslash) \;\leftarrow\; b \oslash \mu(a\oslash) \\
\Leftarrow & \qquad \{ \quad f \;::\; a \leftarrow b, \quad \mathsf{id}_{\mu(a\oslash)} \;::\; \mu(a\oslash) \;\leftarrow\; \mu(a\oslash), \\
 & \qquad\qquad \oslash \text{ respects typing} \quad \} \\
 & \psi \;=\; f \oslash \mathsf{id}_{\mu(a\oslash)} \;.
\end{array}$$

Performing the collected substitutions gives us this candidate definition

$$\boxed{\tau(\oslash)\ f \;=\; (\!|\mathsf{in}_{a\oslash} \bullet (f \oslash \mathsf{id})|\!)}$$

Exercise 2.20 is to show that $\tau(\oslash)$ respects composition and identities.

A final comment: The parameter a in a parameterised functor may actually be an n-tuple if functor $\oslash$ is $(n+1)$-ary, and then $\tau(\oslash)$ is an n-ary functor. However, we only consider unary type functors, derived with $\tau(\oslash)$ from binary functors in these lectures.

Exercise 2.18 Consider the datatype $Bool \;=\; \mu((1+1)^{\mathsf{K}})$. Define $true = \mathsf{in}{\bullet}\mathsf{inl}$, $false = \mathsf{in}{\bullet}\mathsf{inr}$. Examine and explain the meaning of the catamorphism $(\!|u \triangledown v|\!)$ for this datatype.

Exercise 2.19 (cata-map fusion) Derive a fusion rule of the form

$$(\!|f|\!) \bullet (\tau(\oslash)\ g) \;=\; (\!|h|\!) \;.$$

Hint: instantiate the fusion rule for catamorphisms with $F \;:=\; b\oslash$. Note also that $\tau(\oslash)g$ is a catamorphism.

□

Exercise 2.20 Complete the verification of the fact that $\mathcal{T}(\oslash)$ is a functor by showing that $\mathcal{T}(\oslash)\ \mathsf{id}_a = \mathsf{id}_{\mathcal{T}(\oslash)a}$ and $\mathcal{T}(\oslash)\ (f \bullet g) = (\mathcal{T}(\oslash)\ f) \bullet (\mathcal{T}(\oslash)\ g)$. (Hint: make use of exercise 2.19.)

□

Exercise 2.21 Specialise the definition of $\mathcal{T}(\oslash)\ f$ for $\oslash = L$, the bifunctor giving the type functor $List = \mathcal{T}(L)$, using $\mathsf{in} = \mathsf{cons} \triangledown \mathsf{nil}$, and verify that this is the familiar map function for lists. Also, instantiate your solution to exercise 2.19 and use it to express the sum of the squares of a list of numbers as a catamorphism. (That is, express the sum of a list of numbers as a catamorphism, and the list of squares of a list on numbers as a map. Then fuse the two functions together.)

Regular Functors and Datatypes We are now in a position to complete our discussion of the datatypes introduced in section 2.2 by giving a complete analysis of the definition of the Rose datatype. As we saw in exercise 2.12, its pattern functor is $a \oslash z = a \times (List\ z)$, or, in terms of the extraction functors Par and Rec, $(\oslash) = \mathsf{Par} \times (List\ \mathsf{Rec})$. which is not a polynomial functor, because of the appearance of the type functor $List$. Yet $\mathcal{T}(\oslash)$ is well defined. Incorporating type functors into the ways of constructing functors extends the class of polynomial functors to the class of *regular* functors.

A functor built only from constants, extractions, sums, products, composition and $\mathcal{T}()$ is called a *regular* functor. All the datatypes we have seen, including $List$ and $Rose$ are regular functors, and their constructor functions (combined together using the $\triangledown$ combinator) are initial algebras with respect to the pattern functors of the datatype.

This concludes the theory development. We have shown precisely what it means to say that a datatype is both an algebra and a functor.

2.5 A Simple Polytypic Program

We began section 2.2 with four representative examples of datatypes: $List$, $Maybe$, Bin and $Rose$. For each of these datatypes we can define a summation function that sums all the values stored in an instance of the datatype — assuming the values are numbers. Here is how one would do that in a non-generic programming style.

$$
\begin{array}{lllll}
\mathsf{sum}_{List} & (\mathsf{cons}\ u\ x) & = & u & +\ \ \mathsf{sum}_{List}\ x \\
\mathsf{sum}_{List} & \mathsf{nil} & = & \mathsf{0} & \\
\\
\mathsf{sum}_{Maybe} & (\mathsf{one}\ u) & = & u &
\end{array}
$$

$$
\begin{array}{llcl}
\mathsf{sum}_{Maybe} & \mathsf{none} & = & 0 \\
\\
\mathsf{sum}_{Bin} & (\mathsf{join}\ x\ y) & = & \mathsf{sum}_{Bin}\ x \ + \ \mathsf{sum}_{Bin}\ y \\
\mathsf{sum}_{Bin} & (\mathsf{tip}\ u) & = & u \\
\\
\mathsf{sum}_{Rose} & (\mathsf{fork}\ u\ xs) & = & u \ + \ \mathsf{sum}_{List}\ (\mathsf{map}_{List}\ \mathsf{sum}_{Rose}\ xs)
\end{array}
$$

We now want to replace all these definitions by a single generic definition sum_F for arbitrary unary functor F, which can be specialised to any of the above datatype constructors and many more by taking F to be $List$, $Maybe$, Bin, and so on. We do this by induction on the structure of the regular functors. That is, we define summation for a constant functor, for the extraction functors, for the composition of functors, for disjoint sum and cartesian product, and finally for a type functor. Let us begin with the type functors since this is where we see how to formulate the induction hypothesis.

For the type functor $\tau(\oslash)$, the requirement is to construct a function $\mathsf{sum}_{\tau(\oslash)}$ of type $\mathbb{N} \leftarrow \mu(\mathbb{N}\oslash)$. The obvious thing to do here is to define sum as a catamorphism, $(\!|f|\!)$ say. In that case, the type requirement on f is that $f \ :: \ \mathbb{N} \leftarrow \mathbb{N}\oslash\mathbb{N}$. Note that the two arguments to the binary functor $\oslash$ are both $\mathbb{N}$. This suggests the inductive hypothesis that there is a sum function of type $\mathbb{N} \ \leftarrow \ F\ \mathbb{N}$ for all *unary* regular functors F obtained from an arbitrary non-constant n-ary regular functor by copying the (single) argument n times. We also need to define sum for the constant functor $\mathsf{1}$. With this preparation, we can begin the analysis.

For the constant functor $\mathsf{1}$, we define

$$\mathsf{sum}_{\mathsf{1}} \ = \ 0 \ .$$

This is because the sum of zero numbers is zero.

For the extraction functors, it is clear that

$$\mathsf{sum}_{\mathsf{Ex}} \ = \ \mathsf{id}_{\mathbb{N}}$$

since the sum of a single number is that number itself.

For disjoint sum and cartesian product, we have:

$$\mathsf{sum}_{F+G} \ = \ \mathsf{sum}_F \triangledown \mathsf{sum}_G \quad \text{and} \quad \mathsf{sum}_{F\times G} \ = \ \mathsf{sum}_F + \mathsf{sum}_G \ .$$

(The addition on the right of the equation for $\mathsf{sum}_{F\times G}$ is numerical addition, not disjoint sum.) In the case of disjoint sum, either the sum_F function has to be applied, or the sum_G function, depending on the type of the argument. In the case of cartesian product, an element of an $F\times G$ structure is a pair consisting of an element of an F structure and an element of a G structure, and the two sums have to be added together.

For the composition of two functors, we have:

$$\mathsf{sum}_{FG} \quad = \quad \mathsf{sum}_F \bullet F\mathsf{sum}_G \quad .$$

Here the argument is that an FG structure is an F structure of G structures. The function $F\mathsf{sum}_G$ applies sum_G to all the individual G structures, and then sum_F adds their values.

The final case is a type functor, which we have already discussed.

$$\mathsf{sum}_{\mathcal{T}(\oslash)} \quad = \quad (\![\mathsf{sum}_{\oslash}]\!) \quad .$$

We leave it to the reader to check that application of the above rules results in the particular instances of sum given above.

Chapter 3

PolyP

The previous chapter introduces datatypes and functions on datatypes such as the catamorphism. The formal language used to introduce datatypes and functions is the language of category theory. The language of category theory is not a programming language, and although the accompanying text mentions programming, it is impossible to 'run' catamorphisms. This chapter introduces PolyP, a programming language with which generic functions such as the catamorphism can be implemented. The name of PolyP is derived from 'polytypic programming', an alternative name for generic programming.

PolyP is an extension of (a subset of) the functional programming language Haskell. The extension introduces a new kind of (top level) definition, the `polytypic` construct, which is used to define functions by induction over pattern functors, which describe the structure of (a subset of) regular datatypes. PolyP is based on the initial algebra approach to datatypes and work in the Squiggol community on datatypes. It is a tool that supports polytypic programming, and as such it has spurred the development of new polytypic programs.

In Haskell, datatypes are defined by means of the `data` construct, examples of which have been given in chapter 2. PolyP extracts the pattern functor from a datatype definition, and uses this structure information to instantiate generic programs on particular datatypes. We will use the name *polytypic function* for a generic program in PolyP.

PolyP has a number of limitations. The datatypes PolyP can handle are a subset of the datatypes induced by the regular functions defined in the previous chapter: PolyP's functors are binary and hence it can only handle datatypes with one type argument. Furthermore, datatypes cannot be mutually recursive.

Information about PolyP and polytypic programming in general can be found on

`http://www.cs.chalmers.se/~patrikj/poly/`

The names of regular functors in PolyP differ slightly from the names in the previous chapter. Section 3.1 introduces PolyP's functor names. Section 3.2 gives an example of a polytypic function: `psum`. Section 3.3 defines most of the basic polytypic functions. Type checking of polytypic functions is explained in section 3.4. Since we will use a number of polytypic functions in the rest of these notes, section 3.5 gives more examples of polytypic functions, and section 3.6 introduces PolyLib: a library of polytypic functions.

3.1 Regular Functors in PolyP

The previous chapter explains how datatypes are defined by means of pattern functors. A pattern functor is a regular functor, i.e., a polynomial functor possibly extended with a type functor. PolyP defines polytypic functions by induction over the pattern functor of a datatype. The names for the pattern functors constructors used in PolyP differs slightly from the names in the previous chapter. This section defines the syntax for pattern functors used in PolyP.

PolyP's functors are specified by the following context-free grammar:

```
f  ::=  f + g | f * g | Empty | Par | Rec | d @ g | Const t
```

The following table relates this syntax to the functors introduced in the previous chapter.

`+`	`*`	`Empty`	`Par`	`Rec`	`d @ g`	`Const t`
$+$	$\times$	1^{K}	$\mathtt{Ex}^2_0$	$\mathtt{Ex}^2_1$	$a \mapsto b \mapsto \mathtt{d}(\mathtt{g}\ a\ b)$	$\mathtt{t}^{\mathsf{K}}$

`+` and `*` are the standard sum and product functors. `Empty` is the constant binary functor 1^{K}. `Par` and `Rec` are mentioned in chapter 2, and are `Exl` and `Exl`, respectively. Composition of functors `d` and `g` is denoted by `d @ g` and is only defined for a unary functor `d` and a binary functor `g`. Finally, `Const t` is the binary variant of $\mathtt{t}^{\mathsf{K}}$. Although `t` seems to be a variable, it should only be used for monotypes such as `Bool`, `Char`, `Float`, and `Int` (at the moment PolyP knows about nine of Haskell's monotypes).

For each regular datatype defined by means of Haskell's `data` construct, PolyP generates a pattern functor. For example, for

```
data Error a  =  Error String | One a
data List a   =  Nil | Cons a (List a)
data Bin a    =  Tip a | Join (Bin a) (Bin a)
data Rose a   =  Fork a (List (Rose a))
```

PolyP generates the following functors:

```
FError  =  Const String + Par
FList   =  Empty + Par * Rec
FBin    =  Par + Rec * Rec
FRose   =  Par * (List @ Rec)
```

Functors are *only* constructed for datatypes defined by means of the `data` construct. If somewhere in a program a polytypic function is applied to a value of type `Error (List a)`, PolyP will generate an instance of the polytypic function on the datatype `Error a`, not on the type `(Error @ List) a`. This also implies that the functor `d` in the functor composition `d @ g` is always a type functor.

PolyP contains one more functor constructor, namely `FunctorOf`, which returns the binary pattern functor used in a type functor. For example, `FunctorOf List` equals `Empty + Par * Rec`. `FunctorOf D` equals `FD` for all datatypes `D` defined above.

3.2 An Example: `psum`

PolyP introduces a new construct `polytypic` for defining polytypic functions by induction on the structure of a functor:

```
polytypic p :: t = case f of {fi -> ei}
```

where `p` is the name of the value being defined, `t` is its type, `f` is a functor variable, `fi` are functor patterns and `ei` are PolyP expressions. The explicit type in the `polytypic` construct is needed since we cannot in general infer the type from the cases.

The informal meaning is that we define a function that takes a pattern functor (a value describing the structure of a datatype) as its first argument. This function selects the expression in the first branch of the case matching the functor. Thus the polytypic construct is a template for constructing instances of polytypic functions given the pattern functor of a datatype. The functor argument of the polytypic function need not (and cannot) be supplied explicitly but is inserted by the compiler during type inference.

As an example we take the function `psum` defined in figure 3.1. (The subscripts indicating the type are included for readability and are not part of the definition.) Function `psum` sums the integers in a structure with integers. It is the PolyP implementation of the function `sum` defined in section 2.5. The datatype `Either` and function `either` are defined in Haskell's prelude. The definition of functions `cata` and `pmap` (the implementations in PolyP of the catamorphism and the map, see chapter 2) will be given later. A datatype `d a` is regular (satisfies `Regular d`) if it contains no function spaces, and if the argument of the type constructor `d` is the same on the left- and right-hand side of its definition. When `psum` is used on an element of type `Bin Int`, the compiler performs roughly the following rewrite steps to construct the actual instance of `psum` for `Bin`:

$$\mathtt{psum}_{Bin} \;\; \rightarrow \;\; \mathtt{cata}_{Bin}\ \mathtt{fsum}_{FBin}$$

It follows that we need an instance of `cata` on the datatype `Bin a`, and an instance of function `fsum` on the functor of `Bin a`. For the latter instance, we use the definition of *FBin* and the definition of `fsum` to transform $\mathtt{fsum}_{FBin}$ as follows.

$$\mathtt{fsum}_{FBin} \rightarrow \mathtt{fsum}_{Par+Rec\times Rec} \rightarrow \mathtt{either}\ \mathtt{fsum}_{Par}\ \mathtt{fsum}_{Rec\times Rec}$$

```
psum :: Regular d => d Int -> Int
psum =  cata fsum

polytypic fsum_f :: f Int Int -> Int
    = case f of
        g + h    ->  either fsum_g fsum_h
        g * h    ->  \(x,y) -> fsum_g x + fsum_h y
        Empty    ->  \x -> 0
        Par      ->  \x -> x
        Rec      ->  \x -> x
        d @ g    ->  psum_d . (pmap_d fsum_g)
        Const t  ->  \x -> 0

data Either a b  =  Left a | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)   =  f x
either f g (Right x)  =  g x
```

Figure 3.1: The definition of `psum`

We transform the functions $\texttt{fsum}_{Par}$ and $\texttt{fsum}_{Rec\times Rec}$ separately. For $\texttt{fsum}_{Par}$ we have

$$\texttt{fsum}_{Par} \rightarrow \texttt{\textbackslash x -> x}$$

and for $\texttt{fsum}_{Rec\times Rec}$ we have

$$\begin{aligned} & \texttt{fsum}_{Rec\times Rec} \\ \rightarrow\ & \texttt{\textbackslash(x,y) -> fsum}_{Rec}\ \texttt{x + fsum}_{Rec}\ \texttt{y} \\ \rightarrow\ & \texttt{\textbackslash(x,y) -> (\textbackslash x -> x) x + (\textbackslash x -> x) y} \end{aligned}$$

The last function can be rewritten into `uncurry (+)`, and thus we obtain the following function for summing a tree:

$$\texttt{cata}_{Bin}\ \texttt{(either (\textbackslash x -> x) (uncurry (+)))}$$

By expanding $\texttt{cata}_{Bin}$ in a similar way we obtain a Haskell function for the instance of `psum` on `Bin`. The function we obtain is the same as the function $\texttt{sum}_{Bin}$ defined in section 2.5.

3.3 Basic Polytypic Functions

In the definition of function `psum` we used functions like `cata` and `pmap`. This subsection defines these and other basic polytypic functions.

Since polytypic functions cannot refer to constructor names of specific datatypes, we introduce the predefined functions `out` and `inn`. Function `out` is used in polytypic functions instead of pattern matching on the constructors of a datatype. For example `out` on `Bin` is defined as follows:

```
out_Bin (Tip x)      =  Left x
out_Bin (Join l r)   =  Right (l,r)
```

Function `inn` is the inverse of function `out`. It collects the constructors of a datatype into a single constructor function.

```
out  ::  Regular d => d a -> FunctorOf d a (d a)
inn  ::  Regular d => d a <- FunctorOf d a (d a)
```

`FunctorOf` is a special type constructor that takes a datatype constructor, and returns its pattern functor[1]. It is our main means for expressing the relation between datatypes and functors. PolyP generates definitions of `inn` and `out` for all datatypes.

As explained in chapter 2, a functor is a mapping between categories that preserves the algebraic structure of the category. Since a category consists of objects (types) and arrows (functions), a functor consists of two parts: a definition on types, and a definition on functions. The functors used in PolyP are functions that take two types and return a type. The part of the functor that takes two functions and returns a function is called `fmap`, see figure 3.2.

Using `fmap` we can define the polytypic version of function map, `pmap`, as follows:

```
pmap    :: Regular d => (a -> b) -> d a -> d b
pmap f  =  inn . fmap f (pmap f) . out
```

where `out` takes the argument apart, `fmap` applies `f` to parameters and `(pmap f)` recursively to substructures and `inn` puts the parts back together again.

Function `cata` is also defined in terms of function `fmap`:

```
cata    :: Regular d => (FunctorOf d a b -> b) -> (d a -> b)
cata f  =  f . fmap id (cata f) . out
```

Note that this definition is a copy of the computation rule for the catamorphism in chapter 2, with `inn` on the left-hand side replaced by `out` on the right-hand side.

[1] With datatypes as fix-points of functors, `FunctorOf` is the ‘unfix’.

```
polytypic fmap_f :: (a -> c) -> (b -> d) -> f a b -> f c d
    = \p r -> case f of
                g + h    ->  (fmap_g p r) -+- (fmap_h p r)
                g * h    ->  (fmap_g p r) -*- (fmap_h p r)
                Empty    ->  id
                Par      ->  p
                Rec      ->  r
                d @ g    ->  pmap_d (fmap_g p r)
                Const t  ->  id

(-*-) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f -*- g) (x,y) = (f x , g y)

(-+-) :: (a -> c) -> (b -> d) -> Either a b -> Either c d
(f -+- g) = either (Left . f) (Right . g)
```

Figure 3.2: The definition of `fmap`.

3.4 Type Checking Polytypic Functions

We want to be sure that functions generated by polytypic functions are type correct, so that no run-time type errors occur. For that purpose PolyP type checks definitions of polytypic functions. This subsection briefly discusses how to type check polytypic functions, the details of the type checking algorithm can be found in [24].

In order to type check inductive definitions of polytypic functions the system has to know the type of the polytypic function: higher-order unification is needed to infer the type from the types of the functions in the `case` branches, and general higher-order unification is undecidable. This is the reason why inductive definitions of polytypic functions need an explicit type declaration. Given an inductive definition of a polytypic function

```
polytypic foo :: ... f ...
  = case f of
      g + h -> bar
        ...
```

where `f` is a functor variable, the rule for type checking these definitions checks among other things that the declared type of function `foo`, with `g + h` substituted for `f`, is an instance of the type of expression `bar`. For all of the expressions in the branches of the `case` it is required that the declared type is an instance of the type of the expression in the branch with the left-hand side of the branch substituted for `f` in the declared type. The expression `g + h` is an abstraction of a type, so by substituting `g + h` (or any of the other abstract type expressions) for `f` in the type of `foo` we mean the following: substitute `g +`

`h` for `f`, and rewrite the expression obtained thus by interpreting the functor constructors as the following type synonyms:

```
type (f + g) a b  =  Either (f a b) (g a b)
type (f * g) a b  =  (f a b , g a b)
type Empty a b    =  ()
type Par a b      =  a
type Rec a b      =  b
type (d @ g) a b  =  d (g a b)
type Const t a b  =  t
```

As an example we take the case `g * h` in the definition of `fsum`:

```
polytypic fsum :: f Int Int -> Int
  = case f of
      ...
      g * h  ->  \(x,y) -> fsum x + fsum y
      ...
```

The type of the expression `\(x,y) -> fsum x + fsum y` is `(g Int Int, h Int Int) -> Int`. Substituting the functor to the left of the arrow in the case branch, `g * h`, for `f` in the declared type `f Int Int -> Int` gives `(g * h) Int Int -> Int`, and rewriting this type using the type rewrite rules, gives `(g Int Int, h Int Int) -> Int`. This type is equal to (and hence certainly an instance of) the type of the expression to the right of the arrow in the case branch, so this part of the polytypic function definition is type correct.

3.5 More Examples of Polytypic Functions

This section describes some polytypic functions that will be used in the sequel. These functions can be found in PolyLib, the library of PolyP. The next section gives an overview of PolyLib.

Function `flatten` takes a value of type `d a` and flattens it into a list of values of type `[a]`. It is defined using function `fflatten :: f a [a] -> [a]`, which takes a value `v` of type `f a [a]`, and returns the concatenation of all the values (of type `a`) and lists (of type `[a]`) occurring at the top level in `v`. The definition of `flatten` and `fflatten` is given in figure 3.3. As an example, we unfold the definition of `fflatten` when used on the type `List a` (remember that `FunctorOf List` = `Empty+Par*Rec`):

$$
\begin{aligned}
&\texttt{fflatten}_{Empty+Par*Rec}\\
&\quad = \texttt{either fflatten}_{Empty}\ \texttt{fflatten}_{Par*Rec}\\
&\quad = \texttt{either nil (\textbackslash(x,y) -> fflatten}_{Par}\ \texttt{x ++ fflatten}_{Rec}\ \texttt{y)}\\
&\quad = \texttt{either nil (\textbackslash(x,y) -> id x ++ id y)}\\
&\quad = \texttt{either nil (uncurry (++))}
\end{aligned}
$$

```
flatten :: Regular d => d a -> [a]
flatten =  cata fflatten

polytypic fflatten_f :: f a [a] -> [a]
    = case f of
        g + h    ->  either fflatten_g fflatten_h
        g * h    ->  \(x,y) -> fflatten_g x ++ fflatten_h y
        Empty    ->  nil
        Par      ->  singleton
        Rec      ->  id
        d @ g    ->  concat . flatten_d . pmap_d fflatten_g
        Const t  ->  nil

nil x = []
singleton x = [x]
```

Figure 3.3: The definition of `flatten` and `fflatten`.

The expression `pequal eq x y` checks whether or not the values `x` and `y` are equivalent using the equivalence operator `eq` to compare the elements pairwise. It is defined in terms of function `fequal`, the first argument of which compares parameters for equality, the second argument compares the subterms and the third and fourth arguments are the two (unfolded) terms to be compared. These functions are defined in figure 3.4.

3.6 PolyLib: a Library of Polytypic Functions

Using different versions of PolyP (and its predecessors) we have implemented a number of polytypic programs. For example, we have implemented a polytypic equality function, a polytypic show function, and a polytypic parser. Furthermore, we have implemented some more involved polytypic programs for pattern matching, unification and rewriting. These polytypic programs use several basic polytypic functions, such as the relatively well-known `cata` and `pmap`, but also less well-known functions such as `propagate` and `thread`. We have collected these basic polytypic functions in the library of PolyP: PolyLib [26, app. B]. This paper describes the polytypic functions in PolyLib, motivates their presence in the library, and gives a rationale for their design. This section first introduces the format used for describing polytypic library functions, then it gives an overview of the contents of the library, followed by a description of each of the submodules in the library.

```
polytypic fequal_f :: (a -> b -> Bool) -> (c -> d -> Bool) ->
                     f a c -> f b d -> Bool
  = \p r -> case f of
              g + h    ->  sumequal  (fequal_g p r) (fequal_h p r)
              g * h    ->  prodequal (fequal_g p r) (fequal_h p r)
              Empty    ->  \_ _ -> True
              Par      ->  p
              Rec      ->  r
              d @ g    ->  pequal_d (fequal_g p r)
              Const t  ->  (==)

pequal :: (a -> b -> Bool) -> d a -> d b -> Bool
pequal eq x y = fequal eq (pequal eq) (out x) (out y)

sumequal :: (a -> b -> Bool) -> (c -> d -> Bool) ->
            Either a c -> Either b d -> Bool
sumequal f g (Left  x) (Left  v) = f x v
sumequal f g (Right y) (Right w) = g y w
sumequal f g _         _         = False

prodequal :: (a -> b -> Bool) -> (c -> d -> Bool) ->
             (a,c) -> (b,d) -> Bool
prodequal f g (x,y) (v,w) = f x v && g y w
```

Figure 3.4: The definition of `pequal` and `fequal`.

3.6.1 Describing Polytypic Functions

The description of a polytypic function consists of (some of) the following components: its name and type; an (in)formal description of the function; other names the function is known by; known uses of the function; and its background and relationship to other polytypic functions. For example:

> `pmap` :: `(a -> b) -> d a -> d b`
>
> Function `pmap` takes a function `f` and a value `x` of datatype `d a`, and applies `f` ... **Also known as**: `map` [30], $\texttt{map}_n$ [28]. **Known uses**: Everywhere! **Background**: This was one of the first ...

A problem with describing a library of polytypic functions is that it is not completely clear how to *specify* polytypic functions. The most basic combinators have immediate category theoretic interpretations that can be used as a specification, but for more complicated combinators the matter is not all that obvious. Thus, we will normally not provide formal specifications of the library functions, though we try to give references to more in-depth treatments.

The polytypic functions in the library are only defined for regular datatypes `d a`. In the type this is indicated by adding a context `Regular d => ...`, but we will omit this for brevity.

3.6.2 Library Overview

We have divided the library into six parts, see figure 3.5. The first part of the library contains powerful recursion combinators such as `map`, `cata` and `ana`. This part is the core of the library in the sense that it is used in the definitions of all the functions in the other parts. The second part deals with zips and some derivates, such as the equality function. The third part consists of functions that manipulate monads. The fourth and fifth parts consist of simpler (but still very useful) functions, like flattening and summing. The sixth part consists of functions that manipulate constructors and constructor names. The following sections describe each of these parts in more detail.

`pmap`, `fmap`, `cata`
`ana`, `hylo`, `para`
`crush`, `fcrush`

(a) Recursion op's

`pzip`, `fzip`
`punzip`, `funzip`
`pzipWith`, `pzipWith'`
`pequal`, `fequal`

(b) Zips etc.

`pmapM`, `fmapM`, `cataM`
`anaM`, `hyloM`, `paraM`
`propagate`, `cross`
`thread`, `fthread`

(c) Monad op's

`flatten`, `fflatten`
`fl_par`, `fl_rec`, `conc`

(d) Flatten functions

`psum`, `size`, `prod`
`pand`, `pall`
`por`, `pany`, `pelem`

(e) Miscellaneous

`constructorName`, `fconstructorName`
`constructors`, `fconstructors`
`constructor2Int`, `fconstructor2Int`
`int2Constructor`, `int2fConstructor`

(f) Constructor functions

Figure 3.5: Overview of PolyLib

3.6.3 Recursion Operators

```
pmap :: (a -> b) -> d a -> d b
fmap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

Function `pmap` takes a function `f` and a value `x` of datatype `d a`, and applies `f` recursively to all occurrences of elements of type `a` in `x`. With `d` as a functor acting on types, $\texttt{pmap}_d$ is the corresponding functor action on functions. Function $\texttt{fmap}_f$ is the corresponding functor action for a bifunctor `f`. **Also known as**: `map` [30], $\texttt{map}_n$ [28]. In **charity** [12] $\texttt{map}_d$ `f x` is written `d{f}(x)`. **Known uses**: Everywhere! Function `fmap` is used in the definition of `pmap`, `cata`, `ana`, `hylo`, `para` and in many other PolyLib functions. **Background**: The map function was one of the first combinators distinguished in the work of Bird and Meertens, [11, 34]. The traditional `map` in functional languages maps a function over a list of elements. The current Haskell version of map is overloaded:

```
map :: Functor f => (a->b) -> f a -> f b
```

and can be used as the polytypic `pmap` if instance declarations for all regular type constructors are given. Function `pmap` can be used to give default instances for the Haskell `map`.

```
cata :: (FunctorOf d a b -> b) -> (d a -> b)
ana  :: (FunctorOf d a b <- b) -> (d a <- b)
hylo :: (f a b -> d) -> (c -> f a b) -> (c -> d)
para :: (d a -> FunctorOf d a b -> b) -> (d a -> b)
```

Four powerful recursion operators on the type `d a`: The catamorphism, `cata`, "evaluates" a data structure by recursively replacing the constructors with functions. The typing of `cata` may seem unfamiliar but with the explanation of `FunctorOf` above it can seen as equivalent to:

```
cata :: (f a b -> b) -> (Mu f a -> b)
```

The anamorphism, `ana`, works in the opposite direction and builds a data structure. The hylomorphism, `hylo`, is the generalisation of these two functions that simultaneously builds and evaluates a structure. Finally, the paramorphism, `para`, is a generalised form of `cata` that gives its parameter function access not only to the results of evaluating the substructures, but also the structure itself. **Also known as**:

PolyLib	Functorial ML [8]	Squiggol	**charity** [12]
`cata i`	$\texttt{fold}_1$ `i`	$(\!\mid i \mid\!)$	`{| i |}`
`ana o`	-	$[\!(o)\!]$	`(| o |)`

Functions `cata` and `para` are instances of the Visitor pattern in [20]. **Known uses**: Very many polytypic functions are defined using `cata`: `pmap`, `crush`, `thread`, `flatten`, `propagate`, and all our applications use it. Function `para` is used in `rewrite`. **Background**: The catamorphism, `cata`, is the generalisation of the Haskell function `foldr` and the anamorphism, `ana`, is the (category theoretic) dual. Catamorphisms were introduced by Malcolm [32, 33]. A hylomorphism is the fused composition of a catamorphism and an anamorphism specified by: `hylo i o = cata i . ana o`. The paramorphism [35], `para`, is the elimination construct for the type `d a` from Martin Löf type theory. It captures the recursion pattern of primitive recursive functions on the datatype `d a`.

```
crush  :: (a->a->a) -> a -> d a -> a
fcrush :: (a->a->a) -> a -> f a a -> a
```

The function `crush op e` takes a structure `x` and inserts the operator `op` from left to right between every pair of values of type `a` at every level in `x`. (The value `e` is used in empty leaves.) **Known uses**: within the library see section 3.6.7. Many of the functions in that section are then used in the different applications. **Background**: The definition of `crush` is found in [36]. For an associative operator `op` with unit `e`, `crush op e` can be defined as `foldr op e . flatten`. As `crush` has the same arguments as `fold` on lists it can be seen as an alternative to `cata` as the generalisation of `fold` to regular datatypes.

3.6.4 Zips

```
pzip   :: (d a,d b) -> Maybe ( d (a,b) )
punzip :: d (a,b) -> (d a,d b)
fzip   :: (f a b,f c d) -> Maybe ( f (a,c) (b,d) )
funzip :: f (a,c) (b,d) -> (f a b,f c d)
```

Function `punzip` takes a structure containing pairs and splits it up into a pair of structures containing the first and the second components respectively. Function `pzip` is a partial inverse of `punzip`: it takes a pair of structures and zips them together to `Just` a structure of pairs if the two structures have the same shape, and to `Nothing` otherwise. **Also known as**: $\texttt{zip}_m$ [28], **zip.×.d** [22], **Known uses**: Function `fzip` is used in the definition of `pzipWith`. **Background**: The traditional function `zip`

```
zip :: [a] -> [b] -> [(a,b)]
```

combines two lists and does not need the `Maybe` type in the result as the longer list can always be truncated. (In general such truncation is possible for all types that have a nullary constructor, but not for all regular types.) A more general ("doubly polytypic") variant of `pzip`: `transpose` (called **zip.d.e** in [22])

```
transpose :: d (e a) -> e (d a)
```

was first described by Fritz Ruehr [41]. For a formal definition, see Hoogendijk & Backhouse [22].

```
pzipWith  :: ((a,b) -> Maybe c) -> (d a,d b) -> Maybe (d (a,b))
pzipWith' :: (FunctorOf d c e -> e) -> ((d a,d b) -> e) ->
             ((a,b) -> c) -> (d a,d b) -> e
```

Function `pzipWith op` works like `pzip` but uses the operator `op` to combine the values from the two structures instead of just pairing them. As the zip might fail, we also give the operator a chance to signal failure by giving it a `Maybe`-type as a result.[2]

Function `pzipWith'` is a generalisation of `pzipWith` that can handle two structures of different shape. In the call `pzipWith' ins fail op`, `op` is used as long as the structures have the same shape, `fail` is used to handle the case when the two structures mismatch, and `ins` combines the results from the substructures. (The type of `ins` is the same as the type of the first argument to `cata`.) **Also known as**: $\mathtt{zipop}_m$ [28]. **Known uses**: Function `pzipWith'` is used in the definition of equality, matching and even unification. **Background**: Function `pzipWith` is the polytypic variant of the Haskell function `zipWith`

```
zipWith :: (a->b->c) -> [a] -> [b] -> [(a,b)]
```

but `pzipWith'` is new. Function `pzip` is just `pzipWith Just`.

```
pequal :: (a->b->Bool) -> d a -> d b -> Bool
fequal :: (a->b->Bool) -> (c->d->Bool) -> f a c -> f b d -> Bool
```

The expression `pequal eq x y` checks whether or not the structures `x` and `y` are equivalent using the equivalence operator `eq` to compare the elements pairwise. **Known uses**: `fequal` is used in the unification algorithm to determine when two terms are top level equal. **Background**: An early version of a polytypic equality function appeared in [42]. Function `pequal` can be instantiated to give a default for the Haskell `Eq`-class for regular datatypes:

```
(==) :: Eq a => d a -> d a -> Bool
(==) =  pequal (==)
```

In Haskell the equality function can be automatically derived by the compiler, and our polytypic equality is an attempt at moving that derivation out of the compiler into the prelude.

[2] The type constructor `Maybe` can be replaced by any monad with a zero, but we didn't want to clutter up the already complicated type with contexts.

3.6.5 Monad Operations

```
pmapM  :: Monad m => (a -> m b) -> d a -> m (d b)
pmapMr :: Monad m => (a -> m b) -> d a -> m (d b)
fmapM  :: Monad m => (a->m c) -> (b->m d) -> f a b -> m (f c d)
cataM  :: Monad m => (FunctorOf d a b->m b) -> (d a -> m b)
anaM   :: Monad m => (b->m (FunctorOf d a b)) -> (b -> m (d a))
hyloM  :: Monad m => (f a b->m d) -> (c->m (f a b)) -> c -> m d
paraM  :: Monad m => (d a->FunctorOf d a b->m b) -> d a -> m b
```

Function `pmapM` is a variant of `pmap` that threads a monad `m` from left to right through a structure after applying its function argument to all elements in the structure. Function `pmapMr` is the same but for threading a monad `m` from right to left through a structure. For symmetry's sake, the library also contains a function `pmapMl`, which is equal to `pmapM`. Furthermore, the library also contains the left and right variants of functions like `cataM` etc. A monadic map can, for example, use a state monad to record information about the elements in the structure during the traversal. The other recursion operators are generalised in the same way to form even more general combinators. **Also known as**: traversals [28]. **Known uses**: in `unify` and in the parser. **Background**: Monadic maps and catamorphisms are described in [19]. Monadic anamorphisms and hylomorphisms are defined in [37]. The monadic map (also called active traversal) is closely related to `thread` (also called passive traversal):

```
pmapM f  =  thread . pmap f
thread   =  pmapM id
```

```
propagate :: d (Maybe a) -> Maybe (d a)
cross     :: d [a] -> [d a]
```

Function `propagate` propagates `Nothing` to the top level. Function `cross` is the cross (or tensor) product that given a structure `x` containing lists, generates a list of structures of the same shape. This list has one element for every combination of values drawn from the lists in `x`. These two functions can be generalised to `thread` any monad through a value. **Known uses**: `propagate` is used in the definition of `pzip`. **Background**: Function `propagate` is an instance of `transpose` [41], and both `propagate` and `cross` are instances of `thread` below.

```
thread  :: Monad m => d (m a) -> m (d a)
fthread :: Monad m => f (m a) (m b) -> m (f a b)
```

Function `thread` is used to tie together the monad computations in the elements from left to right. **Also known as**: $\texttt{dist}_d$ [19]. **Known uses**: Function `thread` can be used to define the monadic map: `pmapM f = thread . pmap f`. Function `fthread` is also used in the parser to thread the parsing monad through different structures. Function `thread` can be instantiated (with `d` = `[]`) to the Haskell prelude function

```
accumulate :: Monad m => [m a] -> m [a]
```

but also (with `m` = `Maybe`) to `propagate` and (with `m` = `[]`) to `cross`.

3.6.6 Flatten Functions

```
flatten  :: d a -> [a]
fflatten :: f a [a] -> [a]
fl_par   :: f a b -> [a]
fl_rec   :: f a b -> [b]
```

Function `flatten x` traverses the structure `x` and collects all elements from left to right in a list. The other three function are variants of this for a bifunctor `f`. **Also known as**: $\texttt{extract}_{m,i}$ [28], **listify** [22]. **Known uses**: `fl_rec` is used in the unification algorithm to find the list of immediate subterms of a term. Function `fflatten` is used to define `flatten`

```
flatten = cata fflatten
```

Background: In the relational theory of polytypism [22] there is a membership relation **mem.d** for every relator (type constructor) **d** such that `a` **mem.d** `x` $\equiv$ `a ‘elem‘ (`$\texttt{flatten}_d$ `x)`

3.6.7 Miscellaneous

A number of simple polytypic functions can be defined in terms of `crush` and `pmap`. For brevity we present this part of PolyLib below by providing only the name, the type and the definition of each function.

```
psum :: d Int -> Int                     psum = crush (+) 0
prod :: d Int -> Int                     prod = crush (*) 1
conc :: d [a] -> [a]                     conc = crush (++) []
pand :: d Bool -> Bool                   pand = crush (&&) True
por  :: d Bool -> Bool                   por  = crush (||) False

size    :: d a -> Int                    size    = psum . pmap (\_->1)
flatten :: d a -> [a]                    flatten = conc . pmap (:[])
pall    :: (a->Bool) -> d a -> Bool      pall p  = pand . pmap p
pany    :: (a->Bool) -> d a -> Bool      pany p  = por  . pmap p
pelem   :: Eq a => a -> d a -> Bool      pelem x = pany (\y->x==y)
```

3.6.8 Constructors

```
constructorName   :: d a -> String
fconstructorName  :: f a b -> String
constructors      :: [d a]
fconstructors     :: [f a b]
constructor2Int   :: d a -> Int
fconstructor2Int  :: f a b -> Int
int2Constructor   :: Int -> d a
int2fConstructor  :: Int -> f a b
```

Function `constructorName` takes a value of type `d a` and returns its outermost constructor name. Function `constructors` returns a list with all the constructors of a datatype `d a`. For example, for the datatype `Bin` it returns `[Tip undefined,Join undefined undefined]`. The functions `constructor2Int` and `int2Constructor` take constructors to integers and vice versa. **Known uses**: `constructorName` is used in `pshow`, the polytypic version of the derived `show` function in Haskell, `constructors` is used in showing, parsing and compressing values, and both `int2Constructor` and `constructor2Int` in compressing values.

Chapter 4

Generic Unification

This chapter presents a substantial application of the techniques that have been developed thus far. The topic is a generic unification algorithm.

Briefly, unification is the process of making two terms (such as arithmetic expressions or type expressions) equal by suitably instantiating the variables in the terms. It is very widely used in, for example, pattern matching, type checking and theorem proving. For those who haven't already encountered it before, let us first give an informal explanation before giving a summary of the development of the generic algorithm.

We explain the process in terms of a specific case before considering the generic version. Consider the datatype definition

$$\begin{array}{rcl} \mathbf{data}\ \mathit{Expr} & = & \mathsf{plus}\ \mathit{Expr}\ \mathit{Expr} \\ & \mid & \mathsf{times}\ \mathit{Expr}\ \mathit{Expr} \\ & \mid & \mathsf{number}\ \mathit{Nat} \\ & \mid & \mathsf{var}\ V \end{array}$$

This can be read as the datatype of abstract syntax trees for a context-free grammar

$$E\ ::=\ \texttt{(}\,E\,\texttt{+}\,E\,\texttt{)}\ \mid\ \texttt{(}\,E\,\texttt{*}\,E\,\texttt{)}\ \mid\ N\ \mid\ V$$

for *terms* like "`((1+x)*3)`" when N produces numbers and V produces variables.

Another view is that a *term* of the datatype Expr is a tree with the constructors plus, times, number and var at the nodes, and numbers and variables at the leaves. In this view, the constructors are *uninterpreted*, which means that trees corresponding to equal but non-identical arithmetic expressions are considered different. For example, the trees corresponding to `((1+x)*3)` and `(3+(x*3))` are different. It is this view of terms as tree structures that is used in unification. Nevertheless, for ease of writing we shall use the concrete syntax of arithmetic expressions to write terms.

Now consider two terms, say `((1+x)*3)` and `((y+z)*3)`. "Unifying" these terms means substituting terms for the variables `x`, `y` and `z` so that the terms become identical. One possibility, in this case, is to substitute `z` for `x` and `1` for `y`. After this substitution both terms become equal to `((1+z)*3)`. There are many other possibilities. For example, we could substitute `1` for all of `x`, `y` and `z`, thus unifying the two terms in the term `((1+1)*3)`. This latter substitution is however less general than the former. Unification involves seeking a "most general" unifier for two given terms. Of course, some pairs of terms are not unifiable: a trivial example is the pair of terms `0` and `1`. These are not unifiable because they contain no variables. The pair of terms `x` and `(1+x)` is also not unifiable, but for a different reason: namely, the first term will always have fewer constructors than the second whatever substitution we make for `x`.

We have described unification for arithmetic expressions but unification is also used for other term algebras. A major application is in polymorphic type inference, as in most modern functional languages. In this application it is type expressions that are unified. Suppose that a program contains the function application `f x`, and at that stage the term representing the type inferred for `f` is `(`p`->`q`)`, and for `x` it is r. Then first p and r are unified. If that fails, there is a type error. Otherwise, let `(`p'`->`q'`)` be the result of applying the most general unifier to `(`p`->`q`)`. That is the new type inferred for `f`, while we get p' for `x`, and q' for the application `f x`.

In a *generic* unification algorithm we make the term structure a parameter of the algorithm. So, one instance of the algorithm unifies arithmetic expressions, another type expressions. In order to formalise this we use F to denote a functor (the pattern functor of the constant terms we want to unify) and show how to extend F to a functor $F^\star$ such that $F^\star V$, for type V of variables, is the set of all terms. We also define substitution of variables, and most general unifiers.

The functor $F^\star$ is (the functor part of) a *monad*. In the last ten years, monads have been recognised to be an important concept in many applications of functional programming. We therefore begin in section 4.1 by introducing the concept at first without reference to unification. There is much that can be said about monads but our discussion is brief and restricted to just what we need to present the unification algorithm. The monad $F^\star$ defined by an arbitrary functor F is then discussed along with the definition of a substitution.

The discussion of the unification algorithm proper begins in section 4.2. Here the discussion is also brief since we assume that the non-generic algorithm is known from the literature. In order to compare the calculational method of proof with traditional proofs, chapter 5 presents a generic proof of one aspect of the algorithm's correctness, namely that a non-trivial expression is not unifiable with any variable that occurs properly in it.

4.1 Monads and Terms

Monads and Kleisli composition A monad is a concept introduced in category theory that has since proved its worth in functional programming. It is a general concept which we introduce via a particular instance, the *Maybe* monad.

Suppose we have two functions

$$\begin{aligned} f &:: Maybe\ a \leftarrow b \\ g &:: Maybe\ b \leftarrow c \end{aligned}$$

Think of these total functions as modelling partial functions: f computes an a-value from a b-value, except that it may fail, and likewise, g computes a b-value from a c-value, except that it may fail. Can we combine these functions into a single function

$$f \diamond g :: Maybe\ a \leftarrow c$$

that combines the computations of f and g when both succeed, and fails when either of them fails? The types don't fit for normal composition, but here is how to do it:

$$\begin{aligned} (f \diamond g)\ x &= h\ (g\ x)\ \mathbf{where} \\ & \qquad h\ (\mathsf{one}\ y) = f\ y \\ & \qquad h\ \mathsf{none} \quad\ = \mathsf{none} \end{aligned}$$

This form of composition is called *Kleisli composition*. Kleisli composition shares some pleasant properties with normal composition. First, the operation is associative:

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

for f, g and h such that the expressions involved are well-typed. We may therefore drop the parentheses in chains of Kleisli compositions and write $f \diamond g \diamond h$. Moreover, $\diamond$ has neutral element one, which we call the *Kleisli identity*:

$$\mathsf{one} \diamond f \ = \ f \ = \ f \diamond \mathsf{one} \quad .$$

Kleisli composition gives a convenient way to fit functions together that would not fit together with normal composition. Kleisli composition is not just possible for *Maybe*, but for many other functors as well. A functor with a Kleisli composition and Kleisli identity —that satisfy a number of laws to be discussed shortly— is called a *monad*. A trivial example is the functor Id: take normal function composition as its Kleisli composition. A less trivial example is the functor *Set*. For this functor, Kleisli composition takes the form

$$(f \diamond g)x \ = \ \{z \mid \exists(y{::}\ y \in gx \wedge z \in fy)\} \ .$$

Its Kleisli identity is the singleton former $\{\_\}$. We shall encounter more monads later.

Formally, the triple $(M, \diamond, \eta)$ is a monad, where $\diamond$ and η are its Kleisli composition and Kleisli identity, if the following properties hold. First, $\diamond$ is a function of polymorphic type

$$(Ma \leftarrow c) \leftarrow (Ma \leftarrow b) \times (Mb \leftarrow c)$$

and η is a function of polymorphic type

$$Ma \leftarrow a \quad .$$

Second, $\diamond$ is associative with η as neutral element. Finally, the following rules are satisfied:

$$\begin{array}{rcl} Mf \bullet (g \diamond h) & = & (Mf \bullet g) \diamond h \\ (f \diamond g) \bullet h & = & f \diamond (g \bullet h) \\ (f \bullet g) \diamond h & = & f \diamond (Mg \bullet h) \end{array}$$

In fact, these equalities are automatically satisfied in all the monads that we consider here. They are consquences of the so-called *free theorem* for $\diamond$. Their validity depends on a property called *(polymorphic) parametricity* that is satisfied by Haskell restricted to total functions which we discuss in section 5.2.

Exercise 4.1 Let $(M, \diamond, \eta)$ be a monad. Express Mf in terms of Kleisli composition and identity. Define

$$\mu = \mathsf{id} \diamond \mathsf{id} :: Ma \leftarrow MMa$$

What is the function μ for the case $M = \mathit{Set}$? Prove that $f \diamond g = \mu \bullet Mf \bullet g$. Also prove the following three equalities:

$$\mu \bullet \mu \ = \ \mu \bullet M\mu$$

$$\mu \bullet \eta \ = \ \mathsf{id} \ = \ \mu \bullet M\eta \quad .$$

□

Terms with variables Recall the datatype *Expr* introduced at the beginning of this section. We can regard it as a datatype for terms involving numbers, addition and multiplication to which has been added an extra alternative for variables.

Let F be the pattern functor corresponding to the definition of *Expr* without variables. Then $\mathit{Expr} = \mu G$, where $G\,a = F a + V$. This can be done generically. Consider, for unary functor F, the unary functor $F{+}V^{\mathsf{K}}$. This, we recall, is defined by

$$(F{+}V^{\mathsf{K}})a \ = \ Fa{+}V$$

where a ranges over types, and

$$(F{+}V^{\mathsf{K}})f \ = \ Fa{+}\mathsf{id}_V$$

where f ranges over functions. Given a datatype μF, we can then extend it with variables by switching to $\mu(F{+}V^{\mathsf{K}})$. We define two embeddings

$$\mathsf{embl} :: \mu(F{+}V^{\mathsf{K}}) \leftarrow \mu F$$

$$\mathsf{embr} :: \mu(F{+}V^{\mathsf{K}}) \leftarrow V$$

by:

$$\mathsf{embr} \;=\; \mathsf{in}_{F+V^{\mathsf{K}}} \bullet \mathsf{inr} \;\;,$$

$$\mathsf{embl} \;=\; (\!|F\,;\;\; \mathsf{in}_{F+V^{\mathsf{K}}} \bullet \mathsf{inl}|\!) \;\;.$$

Note that we have specified the parameter "F" inside the catamorphism brackets here since there is a possibility of confusion between F-algebras and $(F{+}V^{\mathsf{K}})$-algebras. Note also that

$$(F{+}V^{\mathsf{K}})\;\mu(F{+}V^{\mathsf{K}}) \;=\; F\;\mu(F{+}V^{\mathsf{K}}) + V$$

so that

$$\mathsf{in}_{F+V^{\mathsf{K}}} \;::\; \mu(F{+}V^{\mathsf{K}}) \;\leftarrow\; F\;\mu(F{+}V^{\mathsf{K}}) + V \;\;.$$

For fixed F, the mapping $a \;\mapsto\; \mu(F{+}a^{\mathsf{K}})$ is a functor, namely the type functor $\tau(\oslash)$ of the bifunctor $a{\oslash}b = Fb{+}a$. Denote this type functor by $F^\star$ (so $F^\star V = \mu(F{+}V^{\mathsf{K}})$)[1]. Its action on functions is as follows. For $f :: a \leftarrow b$:

$$F^\star f \;=\; (\!|F{+}b^{\mathsf{K}}\,;\;\; \mathsf{in}_{F+a^{\mathsf{K}}} \bullet \mathsf{id}{+}f|\!)$$

The functor $F^\star$ forms the *substitution monad* $(F^\star\;,\,\diamond\;,\,\eta)$ with, for $f \;::\; F^\star\;a \leftarrow b$ and $g \;::\; F^\star\;b \leftarrow c$,

$$f{\diamond}g \;=\; (\!|F{+}b^{\mathsf{K}}\,;\;\; (\mathsf{in}_{F+a^{\mathsf{K}}} \bullet \mathsf{inl}) \triangledown f|\!) \bullet g$$

$$\eta \;=\; \mathsf{embr} \;\;.$$

Note that the catamorphism in the definition of $f{\diamond}g$ has pattern functor $F{+}b^{\mathsf{K}}$, as indicated by the parameter before the semicolon. We omit explicit mention of this information later, but it is vital to the correct use of the computation and other laws. In addition we omit type information on the initial algebra, although again it is vitally important.

Exercise 4.2 Take F to be $a\times$ for some type a. What is the type $(a\times)^\star\mathbf{1}$? What is the Kleisli identity and what is Kleisli composition?

□

---

[1]The star notation is used here to suggest a link with the Kleene star, denoting iteration in a regular algebra. $F^\star$ can be seen as iterating functor F an arbitrary number of times. More significantly, the notation highlights a formal link between monads and closure operators. See, for example, [2] for more details.

Exercise 4.3 Consider the case $F = 1+$. Show that $(1+)^{\star}V \cong \mathsf{N}\times(1+V)$. (Hint: observe that N is an initial 1+ algebra.) Construct the initial algebra

$$\mathsf{in} \;::\; \mathsf{N}\times(1+V) \leftarrow (1 + (\mathsf{N}\times(1+V))) + V$$

and express catamorphisms on elements of type $(1+)^{\star}V$ in terms of catamorphisms on N.

□

Exercise 4.4 Verify that Kleisli composition as defined above is indeed associative and that embr is its neutral element.

□

Assignments and Substitutions An *assignment* is a mapping of variables to terms, for example $\{\, \mathtt{x} := \mathtt{(y+x)}\,, \mathtt{y} := 0 \,\}$. An assignment can be *performed* on a term. This means a *simultaneous and systematic* replacement of the variables in the term by the terms to which they are mapped. For example, performed on the term `(x+y)` it gives `((y+x)+0)`. We model assignments as functions with the typing $F^{\star}V \leftarrow V$. Because we want functions to be total, this means we also have to define the assignment for all variables in V. If $V = \{\, \mathtt{x}\,, \mathtt{y}\,, \mathtt{z} \,\}$, we can make the above assignment total by writing it as $\{\, \mathtt{x} := \mathtt{(y+x)}\,, \mathtt{y} := 0\,, \mathtt{z} := \mathtt{z} \,\}$. Note that left of ":=" in an assignment we have an element of V, and to the right an element of $F^{\star}V$. So to be precise, if assignment f has "`z := z`", this means that $f\,\mathtt{z} = \eta\,\mathtt{z}$. In particular, the (empty) *identity assignment* is η.

Given an assignment $f :: F^{\star}V \leftarrow V$, we want to define the *substitution* $\mathsf{subst}\, f$ as a function performing f on a term. The result is again a term. The term consisting of the single variable x is $\eta\,\mathtt{x}$. Applying $\mathsf{subst}\, f$ to it, the result should be $f\,\mathtt{x}$. So

$$
\begin{array}{ll}
 & (\mathsf{subst}\; f) \bullet \eta \\
= & \quad \{ \quad \text{desired result} \quad \} \\
 & f \\
= & \quad \{ \quad \text{Kleisli identity} \quad \} \\
 & f \diamond \eta \\
= & \quad \{ \quad \text{monad equality} \quad \} \\
 & (f \diamond \mathsf{id}) \bullet \eta \quad .
\end{array}
$$

Since $\mathsf{subst}\, f$ is clearly a catamorphism that distributes through constructors — for example, $(\mathsf{subst}\, f)\;(x{+}y) \;=\; ((\mathsf{subst}\, f\; x) + (\mathsf{subst}\, f\; y))$ — it is fully determined by its action on variables. We have found:

$$
\begin{array}{rcl}
\mathsf{subst} & :: & (F^{\star}V \leftarrow F^{\star}\, V) \leftarrow (F^{\star}V \leftarrow V) \\
\mathsf{subst}\; f & = & f \diamond \mathsf{id}
\end{array}
$$

Two substitutions can always be merged into a single one:

$$
\begin{array}{ll}
 & (\mathsf{subst}\ f) \bullet (\mathsf{subst}\ g) \\
= & \quad \{ \quad \text{definition of } \mathsf{subst} \quad \} \\
 & (f \diamond \mathsf{id}) \bullet (g \diamond \mathsf{id}) \\
= & \quad \{ \quad \text{monad equalities} \quad \} \\
 & f \diamond (\mathsf{id} \bullet (g \diamond \mathsf{id})) \\
= & \quad \{ \quad \mathsf{id} \text{ is identity of } \bullet \quad \} \\
 & f \diamond (g \diamond \mathsf{id}) \\
= & \quad \{ \quad \diamond \text{ is associative} \quad \} \\
 & (f \diamond g) \diamond \mathsf{id} \\
= & \quad \{ \quad \text{definition of } \mathsf{subst} \quad \} \\
 & \mathsf{subst}\ (f \diamond g) \quad .
\end{array}
$$

4.2 Generic Unification

Unifiers Two terms x and y containing variables can be *unified* if there is some assignment f such that performing f on x gives the same result as performing f on y. For example, the two terms

`(u+((1*v)*2))` and `((w*v)+(u*2))`

can be unified by the assignment

$\{\mathtt{u} := \mathtt{(1*(z+3))}, \mathtt{v} := \mathtt{(z+3)}, \mathtt{w} := \mathtt{1}\}$

into the unification

`((1*(z+3))+((1*(z+3))*2))`

Such a unifying assignment is called a *unifier* of the terms. Unifiers are not unique. Another unifier of the same two terms of the example is

$\{\mathtt{u} := \mathtt{(1*z)}, \mathtt{v} := \mathtt{z}, \mathtt{w} := \mathtt{1}\}$

which results in the unification

`((1*z)+((1*z)*2))`

This last unification is more general. If f is a unifier, then, for any assignment h, the combined substitution $h \diamond f$ is also a unifier, since

$$
\begin{array}{ll}
& h \diamond f \quad \text{is a unifier of } (x,y) \\
\equiv & \quad \{ \quad \text{definition of unifier} \quad \} \\
& (h \diamond f)\ x \ = \ (h \diamond f)\ y \\
\equiv & \quad \{ \quad \text{combined substitutions} \quad \} \\
& (\mathsf{subst}\ h)\ (\mathsf{subst}\ f\ x) \ = \ (\mathsf{subst}\ h)\ (\mathsf{subst}\ f\ y) \\
\Leftarrow & \quad \{ \quad \text{cancel } (\mathsf{subst}\ h) \quad \} \\
& \mathsf{subst}\ f\ x \ = \ \mathsf{subst}\ f\ y \\
\equiv & \quad \{ \quad \text{definition of unifier} \quad \} \\
& f \quad \text{is a unifier of } (x,y) \quad .
\end{array}
$$

In the example, the first, less general unifier, can be formed from the more general one by taking $h = \{\texttt{v} := \texttt{(z+3)}\}$. This notion of generality gives a pre-ordering on unifiers (and actually on all assignments): define

$$f \sqsubseteq g \quad \equiv \quad \exists(h \mid f = h \diamond g)$$

The relation $\sqsubseteq$ is obviously transitive and reflexive, but in general not anti-symmetric. If two unifiers are equally general: $f \sqsubseteq g \ \wedge \ g \sqsubseteq f$, then f and g can be different. But they are to all intents and purposes equivalent: they differ at most in the choice of names for the variables in the result.

If two terms are unifiable at all, then among all unifiers there is a *most general unifier*. That term is commonly abbreviated to *mgu*. Clearly, any two mgu's are equivalent. In the example, the second unifier is an mgu.

A generic shell for unification We develop the unification algorithm in two stages. In this stage we give a generic "shell" in terms of type classes. In the second stage, we show how to make any regular functor into an instance of the classes involved.

Terms may have children, they may happen to be variables, and we should be able to see if superficially —at the top level of the term trees— the constructors are equal. As before, we assume a fixed type V for variables. Here are the corresponding class declarations:

$$
\begin{array}{llll}
\textbf{class}\ \mathit{Children}\ t & \textbf{where} & \mathit{children} & :: \mathit{List}\,t \leftarrow t \\
 & & \mathit{mapChildren} & :: (t \leftarrow t) \leftarrow (t \leftarrow t) \\
\textbf{class}\ \mathit{VarCheck}\ t & \textbf{where} & \mathit{varcheck} & :: \mathit{Maybe}\,V \leftarrow t \\
\textbf{class}\ \mathit{TopEq}\ t & \textbf{where} & \mathit{topEq} & :: \mathit{Bool} \leftarrow t \times t
\end{array}
$$

$$\textbf{class}\ (\mathit{Children}\ t, \mathit{VarCheck}\ t, \mathit{TopEq}\ t)\ \Rightarrow\ \mathit{Term}\ t$$

We give a concrete instantiation as an example — illustrating some fine points at the same time. Let C be some type for representing constructors. Here is the datatype we will use to instantiate the classes:

$$\mathbf{data}\ T \ = \ Con\ C\ (List\,T) \ \mid \ Var\,V$$

First we make T into an instance of $Children$:

$$\begin{array}{l}
\mathbf{instance}\ Children\ T\ \mathbf{where} \\
\quad \begin{array}{lll}
children \quad (Con\ c\ ts) & = & ts \\
children \quad (Var\ v) & = & \mathsf{nil} \\
mapChildren\ f\ (Con\ c\ ts) & = & Con\ c\ (\mathsf{map}\ f\ ts) \\
mapChildren\ f\ (Var\ v) & = & Var\ v
\end{array}
\end{array}$$

Note here that $mapChildren\ f$ only maps function f over the *immediate* children of its argument. No recursion is involved.

Here is how T fits in the $VarCheck$ class:

$$\begin{array}{l}
\mathbf{instance}\ VarCheck\ T\ \mathbf{where} \\
\quad \begin{array}{lll}
varcheck\ (Con\ c\ ts) & = & \mathsf{none} \\
varcheck\ (Var\ v) & = & \mathsf{one}\ v
\end{array}
\end{array}$$

For $TopEq$ we assume that eq is an equality test on C and on V:

$$\begin{array}{l}
\mathbf{instance}\ TopEq\ T\ \mathbf{where} \\
\quad \begin{array}{lll}
topEq\ ((Con\ c_0\ ts_0), (Con\ c_1\ ts_1)) & = & (\mathsf{eq}\ c_0\ c_1\ \wedge\ \mathsf{length}\ ts_0\ =\ \mathsf{length}\ ts_1) \\
topEq\ ((Var\ v_0) \quad , (Var\ v_1) \quad) & = & \mathsf{eq}\ v_0\ v_1 \\
topEq\ (_ \quad , _ \quad) & = & \mathsf{false}
\end{array}
\end{array}$$

(We write (x, y) here, rather than $\mathsf{pair}\,x\,y$.) Note that for this test the children of the terms are irrelevant. Actually, the last two cases could have been omitted: the way $topEq$ is used in the rest of the algorithm, they will never be invoked.

Having made T an instance of the three superclasses of $Term$, we can now proudly announce:

$$\mathbf{instance}\ Term\ T$$

So much for this concrete instantiation. We continue with the generic problem. Here is a function to collect all subterms of a term in the $Term$ class (or actually the $Children$ class):

$$\begin{array}{l}
subTerms :: Children\,t \ \Rightarrow\ List\,t \leftarrow t \\
subTerms\ x \ = \ \mathsf{cons}\ x\ (\mathsf{concat}\ (List\ subTerms\ (children\,x)))
\end{array}$$

and here is a function that uses a set comprehension to collect all variables occurring in a term:

$$\begin{array}{lcl} vars :: Term\, t & \Rightarrow & List\, V \leftarrow t \\ vars\, x & = & [\, v \mid \mathsf{one}\; v \;\in\; List\; varCheck\; (subTerms\, x)\,] \end{array}$$

Earlier we saw a treatment of assignments as functions. Here we introduce a class for assignments, so that it is also possible to make other concrete representations into instances. The parameter a stands for the type of assignments, and t for the terms.

$$\begin{array}{ll} \textbf{class}\; Assig\; a\;\; \textbf{where} & idAssig\;\; :: a \\ & modBind :: (a \leftarrow a) \leftarrow V \times t \\ & lookupIn :: Maybe\, t \leftarrow a \times V \end{array}$$

Assignments as functions can be made into a generic instance by:

$$\begin{array}{l} \textbf{instance}\; Assig\; (\mu(F{+}V) \leftarrow V)\; \textbf{where} \\ \quad \begin{array}{lcl} idAssig & = & \mathsf{embr} \\ modBind\; (v, x) & = & (f \mapsto (v' \mapsto \textbf{if}\; \mathsf{eq}\; v'\; v\; \textbf{then}\; x\; \textbf{else}\; f\; v')) \\ lookupIn\;\; (f, v) & = & \textbf{if}\; \mathsf{eq}\; (f\; v)\; (idAssig\; v)\; \textbf{then}\; \mathsf{none}\; \textbf{else}\; \mathsf{one}\; (f\;\; v) \end{array} \end{array}$$

in which we see both the Kleisli identity embr of the substitution monad, and one of the $Maybe$ monad.

The result none signifies that v is mapped to itself (embedded in the term world). A more concrete instance would be to take a list of pairs (v, x), with v a variable and x a term. Then $idAssig$ is the empty list, $modBind$ can simply cons the pair onto the list, and $lookupIn$ looks for the first pair with the given variable and returns the corresponding term. If the given variable is not found, it fails. An efficient implementation would use balanced trees, or even better hash tables. With the class mechanism the implementation can be *encapsulated*, that is, hidden to the rest of the program, so that the program can first be developed and tested with a simple implementation. It can later be replaced by a more efficient sophisticated implementation without affecting the rest of the program. It should be clear that this is an important advantage.

The unification algorithm proper We give the algorithm — which is basically the algorithm found in the literature — without much explanation. As to notation, we use the monad $(Maybe, \diamond, \eta)$.

$$\begin{array}{lcl} unify\;\, :: (Term\, t, Assig\, s) & \Rightarrow & Maybe\, s \qquad\;\; \leftarrow t \times t \\ unify' :: (Term\, t, Assig\, s) & \Rightarrow & (Maybe\, s \leftarrow s) \leftarrow t \times t \end{array}$$

The definition of $unify$ is now simply to start up $unify'$ with the empty assignment. The function $unify'$ is defined as a higher order function, threading "assignment transformations" together with $\diamond$.

$$
\begin{array}{l}
unify\ (x,y) \;=\; unify'\,(x,y)\ idAssig \\
unify'\ (x,y) \;=\; uni\,(varCheck\,x, varCheck\,y)\ \ \mathbf{where} \\
\quad uni\ (\mathsf{one}\,u\,,\mathsf{one}\,v) \mid \mathsf{eq}\ u\ v \qquad = \eta \\
\quad uni\ (\mathsf{one}\,u\,,\ \_\ ) \qquad\qquad\quad = u \mapsto y \\
\quad uni\ (\ \_\ ,\mathsf{one}\,v) \qquad\qquad\quad = v \mapsto x \\
\quad uni\ (\mathsf{none}\ ,\mathsf{none}\,) \mid topEq\,(x,y) = uniTerms\,(x,y) \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{otherwise} \qquad = \mathsf{const\ none} \\
\quad uniTerms\,(x,y) \;=\; threadList(\mathsf{zipWith}\ unify'\ (children\,x)\ (children\,y))
\end{array}
$$

All the right-hand sides here are functions that return maybe an assignment, given an assignment. The function *threadList* is simply the list catamorphism with Kleisli composition:

$$threadList \;=\; \mathsf{foldr}\ (\diamond)\ \eta$$

The auxiliary operator $(\mapsto)$ should "*modBind*" its arguments into the unifier being collected, but there are two things to be taken care of. No binding may be introduced that would mean an infinite assignment. This is commonly called the *occurs check*. And if the variable is already bound to a term, that term must be unified with the new term, and the unifier obtained must be threaded into the assignment being collected.

$$
\begin{array}{l}
(\mapsto)\ ::\ (Term\,t, Assig\,s)\ \Rightarrow\ (Maybe\,s \leftarrow s) \leftarrow V \times t \\
(v \mapsto x)\ s \;=\; \mathbf{if}\ occursCheck\,(v,s,x) \\
\qquad\qquad\quad \mathbf{then}\ \mathsf{none} \\
\qquad\qquad\quad \mathbf{else\ case}\ lookupIn\,(s,v)\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathsf{one}\ y\ \rightarrow\ ((\eta \bullet modBind\,(v,x)) \diamond unify'\,(x,y))\ s \\
\qquad\qquad\qquad\quad \mathsf{none}\ \ \rightarrow\ \ (\eta \bullet modBind\,(v,x))\ s
\end{array}
$$

The following is a hack to implement the occurs check. This is basically a reachability problem in a graph — is there a cycle from v to itself?, or rather: are we about to create a cycle? We must take account both of the unifier collected already, and the new term. Because we know no cycles were created yet, the graph is more like a tree, so any search strategy terminates. The approach here is not optimally efficient, but in practice quite good with lazy evaluation (and horrible with eager evaluation). There exist linear-time solutions, but they require much more bookkeeping.

$$
\begin{array}{l}
occursCheck\ ::\ (Term\,t, Assig\,s)\ \Rightarrow\ Bool \leftarrow V \times s \times t \\
occursCheck\,(v,s,x) \;=\; v\ \in\ reachlist\ (vars\,x)\ \mathbf{where} \\
\quad reachlist\ \ vs \;=\; vs +\!\!+\ \mathsf{concat}\,(List\,reachable\,vs) \\
\quad reachable\ v \;\;=\; reachlist\,(mayvars\,(lookupIn\,(s,v))) \\
\quad mayvars\,(\mathsf{one}\ y) \;=\; vars\ y \\
\quad mayvars\ \mathsf{none} \quad\ \;=\; [\,]
\end{array}
$$

Here, *reachlist* collects the variables reachable from a *set* of variables, while *reachable* collects the variables reachable from a *single* variable.

The generic $Term$ instance All we have to do now is make $\mu(F+V)$ an instance of the $Term$ class. That is surprisingly easy. For the $Children$ class:

$$\begin{array}{l} \textbf{instance } \mathit{Children}\ \mu(F+V)\ \textbf{where} \\ \quad \mathit{children} \qquad\quad = \mathsf{fl\_rec} \triangledown (\mathsf{nil} \bullet !) \bullet \mathsf{out} \\ \quad \mathit{mapChildren}\ f = \mathsf{in} \bullet F f + \mathsf{id}_V \bullet \mathsf{out} \end{array}$$

where fl_rec is defined in PolyLib, see Section 3.6.6. Here is how $\mu(F+V)$ fits in the $VarCheck$ class:

$$\begin{array}{l} \textbf{instance } \mathit{VarCheck}\ \mu(F+V)\ \textbf{where} \\ \quad \mathit{varcheck} = (\mathsf{none} \bullet !) \triangledown \mathsf{one} \bullet \mathsf{out} \end{array}$$

For $TopEq$ we use the fact that fequal tests on equality of functor structures. fequal is defined in PolyLib, see Section 3.6.4.

$$\begin{array}{l} \textbf{instance } \mathit{TopEq}\ \mu(F+V)\ \textbf{where} \\ \quad \mathit{topEq}\ t\ t' = \mathsf{fequal}\ (==)\ (x \mapsto y \mapsto \mathsf{True})\ (\mathsf{out}\ t)\ (\mathsf{out}\ t') \end{array}$$

For a complete implementation of the generic unification program, see [25].

Chapter 5

From Functions to Relations

In the preceding chapter we have done what we ourselves have decried: we have presented an algorithm without even a verification of its correctness, let alone a construction of the algorithm from its specification. An excuse is that a full discussion of correctness would have distracted from the main goal of that chapter, which was to show how the generic form of the —known to be correct— algorithm is implemented. That is, however, only an excuse since, so far as we know, no proof of correctness of the generic algorithm has ever been constructed. In section 5.4 we remedy this lacuna partially by presenting one lemma in such a proof of correctness. To that end, however, we need to extend the programming calculus from total functions to relations.

5.1 Why Relations?

In a summer school on advanced *functional* programming, it may seem odd to want to introduce relations but there are several good reasons for making it an imperative. In the first place, specifications are typically relations, not total functions. The specification of the unification algorithm is a case in point since it embodies both nondeterminism and partiality. Nondeterminism is embodied in the requirement to compute *a* most general unifier, not *the* most general unifier. It would be infeasible to require the latter since, in general, there is no single most general unifier of two terms. Partiality is also present in the fact that a most general unifier may not exist. Partiality can be got around in the implementation by using the $Maybe$ monad as we did here, but avoiding nondeterminism in the specification is undesirable.

A second reason for introducing relations is that termination arguments are typically based on well-founded relations. Our discussion of the correctness of the unification algorithm in section 5.4 is based on the construction of a well-founded relation, although in this case termination is not the issue at stake.

A third, compelling reason for introducing relations is that the “free theorem” for polymorphic functions alluded to above and discussed in detail below is based on *relations* on functions and necessitates an extension of the concept of functor to relations. Also, the most promising work we know of that aims to be precise about what is really meant by "generic" is that due to Hoogendijk [21] which is based on a relational semantics of higher-order polymorphism .

5.2 Parametric Polymorphism

Space does not allow us to consider the extension to relations in full depth and so we will have to make do with a brief account of the issues involved. For more detail see [10, 1]. We believe, nevertheless, that a discussion of generic programming would be incomplete without a summary of Reynolds' [38] *abstraction theorem* which has since been popularised under the name "theorems for free" by Wadler [43]. (This summary is taken from [22] which may be consulted for additional references.)

Reynolds' goal in establishing the abstraction theorem was to give a precise meaning to the statement that a function is "parametrically polymorphic". Suppose we have a polymorphic function f of type $T\alpha$ for all types α. That is, for each type A there is an instance f_A of type TA. Then *parametricity* of the polymorphism means that for any binary relation R of type $A \sim B$ there is a binary relation TR of type $TA \sim TB$ such that $(f_A\ ,\ f_B) \in TR$. Reynolds' abstraction theorem is the theorem that any polymorphic function expressible in the language defined in his paper is parametric. Wadler called this a "theorem for free" because, as we show shortly, the parametricity of a polymorphic function predicts algebraic properties of that function just from knowing the type of the function! Another way of viewing the theorem is as a healthiness property of functions expressible in a programming language — a programming language that guarantees that all polymorphic functions are parametric is preferable to one that cannot do so.

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

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

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

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

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

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

then the definition of $f{\times}g$, for functions f and g, coincides with the definition of the cartesian product of f and g given in section 2.3. Thus, not only does Reynolds' definition extend the definition of product beyond types, it also extends the definition of the product functor. Note also that the relational composition $f{\bullet}g$ of two functions is the same as their functional composition. That is, $a = f(gc) \;\equiv\; (a,c) \in f{\bullet}g$. So relational composition also extends functional composition. Note finally that $h{\leftarrow}k$ is a *relation* even for *functions* h and k. It is the relation defined by

$$(f,g) \in h{\leftarrow}k \quad \equiv \quad \forall(x,y{::}\quad fx = h(gy) \quad \Leftarrow \quad x = ky) \quad .$$

Simplified and expressed in point-free form this becomes:

$$(f,g) \in h{\leftarrow}k \quad \equiv \quad f{\bullet}k = h{\bullet}g \quad .$$

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

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

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

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

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

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

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

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

Free Theorem for Monads Let us show the abstraction theorem at work on Kleisli composition. Kleisli composition is a polymorphic function of type

$$(Ma \leftarrow c)\ \leftarrow\ (Ma \leftarrow b) \times (Mb \leftarrow c)$$

for all types a, b and c. If it is parametrically polymorphic then it satisfies the property that, for all relations R, S and T and all functions f_0, f_1, g_0 and g_1,

$$(f_0 \diamond g_0\ \ ,\ \ f_1 \diamond g_1)\ \ \in\ \ MR \leftarrow T$$

if

$$((f_0\ ,\ f_1)\ ,\ (g_0\ ,\ g_1))\ \ \in\ \ (MR \leftarrow S) \times (MS \leftarrow T)\ \ .$$

This assumes that we have shown how to extend the functor M to relations. For our purposes here, we will only need to instantiate R, S and T to functions, and it simplifies matters greatly if we use the point-free definition of $h \leftarrow k$ given above. Specifically, we have, for all functions h, k and l,

$$
\begin{array}{cl}
& ((f_0\ ,\ f_1)\ ,\ (g_0\ ,\ g_1))\ \ \in\ \ (Mh \leftarrow k) \times (Mk \leftarrow l) \\
\equiv & \qquad \{ \qquad \text{definition of } \times \quad \} \\
& (f_0\ ,\ f_1)\ \in\ Mh \leftarrow k\ \ \wedge\ \ (g_0\ ,\ g_1)\ \in\ Mk \leftarrow l \\
\equiv & \qquad \{ \qquad \text{point-free definition of } \leftarrow \text{ for functions} \quad \} \\
& f_0 \bullet k\ \ =\ \ Mh \bullet f_1\ \ \ \wedge\ \ \ g_0 \bullet l\ \ =\ \ Mk \bullet g_1\ \ .
\end{array}
$$

In this way, we obtain the property that for all functions f_0, f_1, g_0, g_1, h, k and l,

(5.1) $\quad (f_0 \diamond g_0) \bullet l\ \ =\ \ Mh \bullet (f_1 \diamond g_1)\ \ .$

if

(5.2) $\quad f_0 \bullet k\ \ =\ \ Mh \bullet f_1\ \ \ \wedge\ \ \ g_0 \bullet l\ \ =\ \ Mk \bullet g_1$

With its seven free variables, this is quite a complicated property. More manageable properties can be obtained by instantiating the functions in such a way that the premise becomes true. An easy way to do this is to reduce the premise to statements of the form

$$f_i \;=\; \ldots \quad \wedge \quad g_j \;=\; \ldots \;\;,$$

where i and j are either 0 or 1, by instantiating suitable combinations of h, k and l to the identity function. For instance, by instantiating k and l to the identity function the premise (5.2) reduces to

$$f_0 \;=\; Mh \bullet f_1 \quad \wedge \quad g_0 \;=\; g_1 \;\;.$$

Substituting the right sides for f_0 and g_0 in the conclusion (5.1) together with the identity function for l, we thus obtain

$$(Mh \bullet f_1) \diamond g_1 \;=\; Mh \bullet (f_1 \diamond g_1) \;\;.$$

for all functions h, f_1 and g_1. This is the first of the “free theorems” for Kleisli composition listed in section 4.1.

Exercise 5.3 Derive the other two “free theorems” stated in section 4.1 from the above parametricity property. Investigate other properties obtained by setting combinations of f_0, f_1, g_0, g_1 to the identity function.

□

Exercise 5.4 Instantiating M to the identity functor we see that the free theorem for Kleisli composition predicts that any parametrically polymorphic function with the same type as (ordinary) function composition is associative. Can you show that function composition is uniquely defined by its parametricity property?

□

Exercise 5.5 Derive the free theorem for catamorphisms from the polymorphic type of $f \mapsto (\!|f|\!)$. Show that the fusion law is an instance of the free theorem.

□

5.3 Relators

As we have argued, an extension of the calculus of datatypes to relations is desirable from a practical viewpoint. In view of Reynolds’ abstraction theorem, it is also highly desirable from a theoretical viewpoint, at least if one’s goal is to develop generic programming. We have also shown how the product functor is extended to relations. In a relational theory of datatypes, all functors are extended to relations in such a way that when restricted to functions all their algebraic properties remain unchanged. Functors extended in this way are called *relators*.

The formal framework for this extension is known as an *allegory*. An allegory is a category with additional structure, the additional structure capturing the most essential characteristics of relations. The additional axioms are as follows. First of all, relations of the same type are ordered by the *partial order* $\subseteq$ and composition is monotonic with respect to this order. That is,

$$S_1{\bullet}T_1 \;\subseteq\; S_2{\bullet}T_2 \qquad \Leftarrow \qquad S_1 \;\subseteq\; S_2 \quad \wedge \quad T_1 \;\subseteq\; T_2 \;\; .$$

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

$$X \;\subseteq\; R \;\wedge\; X \;\subseteq\; S \qquad \equiv \qquad X \;\subseteq\; R \cap S \;\; .$$

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

$$R^{\cup} \;\subseteq\; S \qquad \equiv \qquad R \;\subseteq\; S^{\cup} \;\; ,$$

and is contravariant with respect to composition,

$$(R{\bullet}S)^{\cup} \;\; = \;\; S^{\cup} \bullet R^{\cup} \;\; .$$

All three operators of an allegory are connected by the *modular law*, also known as Dedekind's law [39]:

$$R{\bullet}S \;\cap\; T \;\;\subseteq\;\; (R \;\cap\; T{\bullet}S^{\cup}) \bullet S \;\; .$$

Now, a *relator* is a monotonic functor that commutes with converse. That is, the functor F is a relator iff,

(5.6) $\quad FR \bullet FS \;=\; F(R{\bullet}S) \quad$ for each $R\ ::\ A \sim B$ and $S\ ::\ B \sim C$,

(5.7) $\quad F\mathsf{id}_A \;=\; \mathsf{id}_{FA} \quad$ for each A,

(5.8) $\quad FR \;\subseteq\; FS \quad \Leftarrow \quad R \;\subseteq\; S \quad$ for each $R\ ::\ A \sim B$ and $S\ ::\ A \sim B$,

(5.9) $\quad (FR)^{\cup} \;=\; F(R^{\cup}) \quad$ for each $R\ ::\ A \sim B$.

Relators extend functors A design requirement which led to the above definition of a relator [3, 4] is that a relator should extend the notion of a functor but in such a way that it coincides with the latter notion when restricted to functions. Formally, relation $R :: A \sim B$ is *total* iff

$$\mathsf{id}_B \;\subseteq\; R^{\cup}{\bullet}R \quad ,$$

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

$$R{\bullet}R^{\cup} \;\subseteq\; \mathsf{id}_A \quad .$$

A *function* is a relation that is both total and simple. It is easy to verify that total and simple relations are closed under composition. Hence, functions are closed under composition too. In other words, the functions form a sub-category. For an allegory $\mathcal{A}$, we denote the sub-category of functions by $Map(\mathcal{A})$. Moreover, it is easily shown that our definition guarantees that relators preserve simplicity and totality, and thus functionality of relations.

Having made the shift from categories to allegories, the extension of the functional theory of datatypes in chapter 2 is surprisingly straightforward (which is another reason why not doing it is short-sighted). The extension of the disjoint sum functor to a disjoint sum relator can be done in such a way that all the properties of $+$ and $\triangledown$ remain valid, as is the case for the extension of the theory of initial algebras, catamorphisms and type functors. For example, catamorphisms with relations as arguments are well-defined and satisfy the fusion property, the map-fusion property etc. There is, however, one catch — the process of dualising properties of disjoint sum to properties of cartesian product is not valid. Indeed, almost all of the properties of cartesian product that we presented are not valid, in the form presented here, when the variables range over arbitrary relations. (The banana split theorem is a notable exception.)

An example of what goes wrong is the fusion law. Consider $\mathsf{id}{\vartriangle}\mathsf{id} \bullet R$ and $R{\vartriangle}R$, where R is a relation. If R is functional —that is, if for each y there is at most one x such that $(x, y) \in R$ then these two are equal. This is an instance of the fusion law presented earlier. However, if R is not functional then they may not be equal. Take R to be, for example, the relation $\{(0, 0)\ ,\ (1, 0)\}$ in which both 0 and 1 are related to 0. Then,

$$\mathsf{id}{\vartriangle}\mathsf{id} \bullet R \;=\; \{((0, 0)\ ,\ 0)\ ,\ ((1, 1)\ ,\ 0)\}$$

whereas

$$R{\vartriangle}R \;=\; \{((0, 0)\ ,\ 0)\ ,\ ((1, 1)\ ,\ 0)\ ,\ ((0, 1)\ ,\ 0)\ ,\ ((1, 0)\ ,\ 0)\} \quad .$$

The relation $\mathsf{id}{\vartriangle}\mathsf{id}$ is the *doubling* relation: it relates a pair of values to a single value whereby all the values are equal. Thus, $\mathsf{id}{\vartriangle}\mathsf{id} \bullet R$ relates a pair of *equal* values to 0. On the other hand, $R{\vartriangle}R$ relates a pair of values to a single value, whereby each component of the pair is related by R to the single value. The difference thus arises from the nondeterminism in R.

In conclusion, extending the functional theory of datatypes to relations is desirable but not without pitfalls. The pitfalls are confined, however, to the properties of cartesian product. We give no formal justification for this. The reader will just have to trust us that in the ensuing calculations, where one or more argument is a relation, that the algebraic properties that we exploit are indeed valid.

Membership We have argued that a datatype is not just a mapping from types to types but also a functor. We have now argued that a datatype is a relator. For the correctness of the generic unification algorithm we also need to know that a membership relation can be defined on a datatype.

The full theory of membership and its consequences has been developed by Hoogendijk and De Moor [23, 21]. Here we give only a very brief account.

Let F be a relator. A *membership relation* on F is a parametrically polymorphic relation mem of type $a \sim Fa$ for all a. Parametricity means that for all relations R,

$$\mathsf{mem} \bullet FR \quad \supseteq \quad R \bullet \mathsf{mem} \quad .$$

In fact, mem is required to be the largest parametrically polymorphic relation of this type.

The existence of a membership relation captures the idea that a datatype is a structured repository of information. The relation mem_a holds between a value x of type a and an F-structure of a's if x is stored somewhere in the F-structure. The parametericity property expresses the fact that determining membership is independent of the type a, and the fact that mem is the largest relation of its type expresses the idea that determining membership is independent of the position in the data structure at which a value is stored.

The parametricity property has the following consequence which we shall have occasion to use. For all (total) functions f of type $a \leftarrow b$,

$$f \bullet \mathsf{mem}_b \quad = \quad \mathsf{mem}_a \bullet Ff \quad .$$

5.4 Occurs-in

This section contains a proof of the generic statement that two expressions are not unifiable if one occurs in the other. We define a (generic) relation $\mathsf{occurs\_properly\_in}$ and we then show that $\mathsf{occurs\_properly\_in}$ is indeed a proper ordering on expressions (that is, if expression x $\mathsf{occurs\_properly\_in}$ expression y then x and y are different). We also show that the $\mathsf{occurs\_properly\_in}$ relation is invariant under substitution. Thus, if expression x $\mathsf{occurs\_properly\_in}$ expression y no substitution can unify them. To show that

$\mathsf{occurs\_properly\_in}$ is proper we define a (generic) function size of type $\mathbf{N} \leftarrow F^\star V$ and we show that size is preserved by the relation $\mathsf{occurs\_properly\_in}$. The definition of size involves a restriction on the relator F which is used to guarantee correctness of the algorithm[1].

Definition 5.10 The relation $\mathsf{occurs\_properly\_in}$ of type $F^\star\ V \sim F^\star\ V$ is defined by

$$\mathsf{occurs\_properly\_in} \quad = \quad (\mathsf{mem} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^\cup)^+ \quad .$$

(Recall that mem is the membership relation of F and that in is an initial algebra with carrier $F^\star V$.) Informally, the relation $(\mathsf{in}{\bullet}\mathsf{inl})^\cup$ (which has type $FF^\star V \sim F^\star V$) destructs an element of $F^\star V$ into an F structure and then mem identifies the data stored in that F structure. Thus $\mathsf{mem} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^\cup$ destructs an element of $F^\star V$ into a number of immediate subcomponents. Application of the transitive closure operation repeats this process thus breaking the structure down into all its subcomponents.

□

In our first lemma we show that the $\mathsf{occurs\_properly\_in}$ relation is closed under substitutions. That is, for all substitutions f,

$$x \ \mathsf{occurs\_properly\_in}\ y \quad \Rightarrow \quad (fx)\ \mathsf{occurs\_properly\_in}\ (fy) \quad .$$

The property is formulated without mention of the points x and y and proved using point-free relation algebra.

Lemma 5.11 For all substitutions f,

$$\mathsf{occurs\_properly\_in} \quad \subseteq \quad f^\cup \bullet \mathsf{occurs\_properly\_in} \bullet f \quad .$$

Proof Suppose f is a substitution. That is, $f = g\diamond\mathsf{id}$ for some g. Since $\mathsf{occurs\_properly\_in}$ is the transitive closure of the relation $\mathsf{mem} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^\cup$ it suffices to establish two properties: first, that $f^\cup \bullet \mathsf{occurs\_properly\_in} \bullet f$ is transitive and, second,

$$\mathsf{mem} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^\cup \ \subseteq\ f^\cup \bullet \mathsf{occurs\_properly\_in} \bullet f \quad .$$

The first of these is true for all functions f (i.e. relations f such that $f \bullet f^\cup \subseteq \mathsf{id}$). (To be precise, if R is a transitive relation and f is a function then $f^\cup \bullet R \bullet f$ is transitive.) We leave its simple proof to the reader. The second is proved as follows:

$$
\begin{array}{ll}
 & f^\cup \bullet \mathsf{occurs\_properly\_in} \bullet f \\
\supseteq & \{ \quad R^+ \supseteq R \quad \} \\
 & f^\cup \bullet \mathsf{mem} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^\cup \bullet f \\
\supseteq & \{ \quad \mathsf{in}{\bullet}\mathsf{inl} \text{ is a function} \quad \} \\
 & f^\cup \bullet \mathsf{mem} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^\cup \bullet f \bullet \mathsf{in} \bullet \mathsf{inl} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^\cup
\end{array}
$$

[1] A more general proof [6] using the generic theory of F-reductivity [14, 13, 15] avoids this assumption and, indeed, avoids the introduction of the size function altogether.

$$
\begin{array}{ll}
= & \{ \quad f = g\diamond\mathsf{id} = (\!|(\mathsf{in}\bullet\mathsf{inl})\triangledown g|\!), \text{ computation} \quad \} \\
& f^{\cup} \bullet \mathsf{mem} \bullet (\mathsf{in}\bullet\mathsf{inl})^{\cup} \bullet (\mathsf{in}\bullet\mathsf{inl})\triangledown g \bullet Ff{+}\mathsf{id} \bullet \mathsf{inl} \bullet (\mathsf{in}\bullet\mathsf{inl})^{\cup} \\
= & \{ \quad \text{computation} \quad \} \\
& f^{\cup} \bullet \mathsf{mem} \bullet (\mathsf{in}\bullet\mathsf{inl})^{\cup} \bullet \mathsf{in} \bullet \mathsf{inl} \bullet Ff \bullet (\mathsf{in}\bullet\mathsf{inl})^{\cup} \\
= & \{ \quad (\mathsf{in}\bullet\mathsf{inl})^{\cup} \bullet \mathsf{in} \bullet \mathsf{inl} \;=\; \mathsf{id} \quad \} \\
& f^{\cup} \bullet \mathsf{mem} \bullet Ff \bullet (\mathsf{in}\bullet\mathsf{inl})^{\cup} \\
\supseteq & \{ \quad \text{parametricity of } \mathsf{mem} \quad \} \\
& \mathsf{mem} \bullet Ff^{\cup} \bullet Ff \bullet (\mathsf{in}\bullet\mathsf{inl})^{\cup} \\
\supseteq & \{ \quad F \text{ is a relator and } f \text{ is a total function.} \\
& \quad\;\; \text{Thus, } Ff^{\cup} \bullet Ff \;\supseteq\; \mathsf{id} \quad \} \\
& \mathsf{mem} \bullet (\mathsf{in}\bullet\mathsf{inl})^{\cup} \quad .
\end{array}
$$

□

We now define a function size of type $\mathbb{N} \leftarrow F^{\star}V$ by

$$\mathsf{size} \;=\; (\!|(\mathsf{succ} \bullet \Sigma\mathsf{mem}) \triangledown \mathsf{zero}|\!) \quad .$$

Here, Σ is the *summation quantifier*. That is, for an arbitrary relation R with target $\mathbb{N}$,

$$(\Sigma R)x \quad = \quad \Sigma(m\colon\; m\; R\; x\colon\; m) \quad .$$

The assumption in the definition of size is that F is finitely branching: that is, for each F structure x, the number of m such that m mem x is finite.

Expressed in terms of points, the next lemma says that if a term x occurs properly in a term y then the size of x is strictly less than the size of y.

Lemma 5.12

$$\mathsf{occurs\_properly\_in} \quad \subseteq \quad \mathsf{size}^{\cup} \;\bullet\; < \;\bullet\; \mathsf{size} \quad .$$

Proof Note that occurs_properly_in and $<$ are both transitive relations. This suggests that we use the leapfrog rule:

$$a \bullet b^{*} \;\subseteq\; c^{*} \bullet a \quad \Leftarrow \quad a{\bullet}b \subseteq c{\bullet}a$$

which is easily shown to extend to transitive closure:

$$a \bullet b^{+} \;\subseteq\; c^{+} \bullet a \quad \Leftarrow \quad a{\bullet}b \subseteq c{\bullet}a \quad .$$

We have:

$$\begin{array}{ll}
& \mathsf{occurs_properly_in} \;\subseteq\; \mathsf{size}^{\cup} \bullet < \bullet\; \mathsf{size} \\
\equiv & \{ \quad \mathsf{size} \text{ is a total function,} \\
& \quad\; \text{definition of } \mathsf{occurs_properly_in} \quad \} \\
& \mathsf{size} \bullet (\mathsf{mem} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^{\cup})^{+} \;\subseteq\; < \bullet\; \mathsf{size} \\
\Leftarrow & \{ \quad < \text{ is transitive. Thus, } < \;=\; <^{+} . \\
& \quad\; \text{Leapfrog rule} \quad \} \\
& \mathsf{size} \bullet \mathsf{mem} \bullet (\mathsf{in}{\bullet}\mathsf{inl})^{\cup} \;\subseteq\; < \bullet\; \mathsf{size} \\
\equiv & \{ \quad \mathsf{in}{\bullet}\mathsf{inl} \text{ is a total function} \quad \} \\
& \mathsf{size} \bullet \mathsf{mem} \;\subseteq\; < \bullet\; \mathsf{size} \bullet \mathsf{in} \bullet \mathsf{inl} \\
\equiv & \{ \quad \text{definition of } \mathsf{size} \text{ and computation} \quad \} \\
& \mathsf{size} \bullet \mathsf{mem} \;\subseteq\; < \bullet\; \mathsf{succ} \bullet \Sigma\mathsf{mem} \bullet F\mathsf{size} \\
\Leftarrow & \{ \quad < \bullet\; \mathsf{succ} \;=\; \leq \quad \} \\
& \mathsf{size} \bullet \mathsf{mem} \;\subseteq\; \leq \bullet\; \Sigma\mathsf{mem} \bullet F\mathsf{size} \\
\Leftarrow & \{ \quad \text{property of natural numbers: for all } R,\ R \;\subseteq\; \leq \bullet\; \Sigma R \\
& \quad\; \text{That is, } m\ R\ x \;\;\Rightarrow\;\; m \leq \Sigma(m{:}\ m\ R\ x{:}\ m). \quad \} \\
& \mathsf{size} \bullet \mathsf{mem} \;\subseteq\; \mathsf{mem} \bullet F\mathsf{size} \\
\equiv & \{ \quad \mathsf{size} \text{ is a total function,} \\
& \quad\; \text{parametricity of } \mathsf{mem} \text{ for functions} \quad \} \\
& \mathsf{true} \;\; .
\end{array}$$

□

Corollary 5.13 Suppose F is a finitely branching relator. Then

$$x \;\mathsf{occurs\_properly\_in}\; y \;\;\Rightarrow\;\; x \neq y \;\; .$$

Proof By the above lemma,

$$x \;\mathsf{occurs\_properly\_in}\; y \;\;\Rightarrow\;\; \mathsf{size}\; x < \mathsf{size}\; y \;\; .$$

Thus, since $m < n \Rightarrow m \neq n$,

$$x \;\mathsf{occurs\_properly\_in}\; y \;\;\Rightarrow\;\; x \neq y \;\; .$$

□

Corollary 5.14 If x occurs_properly_in y then x and y are not unifiable.

Proof By lemma 5.11, if x $\mathsf{occurs\_properly\_in}$ y then, fx $\mathsf{occurs\_properly\_in}$ fy, for every substitution f. Thus, for every substitution f, $fx \neq fy$.
□

Exercise 5.15 Take F to be $(\mathsf{1}+)$. What is $\mathsf{occurs\_properly\_in}$? Show that the relation is proper. (Note that the membership relation for $(\mathsf{1}+)$ is $\mathsf{inr}^{\cup}$.)

Take F to be $a\times$ for some fixed a. What is $\mathsf{occurs\_properly\_in}$?

□

Chapter 6

Solutions to Exercises

1.1 Take $\otimes$ to be set intersection, $\oplus$ to be set union, $\mathbf{0}$ to be the empty set and $\mathbf{1}$ to be the universe of all colours.
□

2.5

$$
\begin{array}{llll}
\mathsf{map}_{Error} & f & (\mathsf{error}\ s) & = \ \mathsf{error}\ s \\
\mathsf{map}_{Error} & f & (\mathsf{ok}\ x) & = \ \mathsf{ok}\ (f x)
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{map}_{Drawing} & f & (\mathsf{above}\ x\ y) & = \ \mathsf{above}\ (\mathsf{map}_{Drawing}\ f\ x)\ (\mathsf{map}_{Drawing}\ \ f\ y) \\
\mathsf{map}_{Drawing} & f & (\mathsf{beside}\ x\ y) & = \ \mathsf{beside}\ (\mathsf{map}_{Drawing}\ f\ x)\ (\mathsf{map}_{Drawing}\ \ f\ y) \\
\mathsf{map}_{Drawing} & f & (\mathsf{atom}\ x) & = \ \mathsf{atom}\ (f\ \ x)
\end{array}
$$

□

2.6

$$
\begin{array}{ll}
 & (f \triangledown g) \vartriangle (h \triangledown k) \ = \ (f \vartriangle h) \triangledown (g \vartriangle k) \\
\equiv & \{ \quad \vartriangle\text{-characterisation} \quad \} \\
 & f \triangledown g \ = \ \mathsf{exl} \bullet (f \vartriangle h) \triangledown (g \vartriangle k) \ \wedge \ h \triangledown k \ = \ \mathsf{exr} \bullet (f \vartriangle h) \triangledown (g \vartriangle k) \\
\equiv & \{ \quad \triangledown\text{-fusion} \quad \} \\
 & \quad f \triangledown g \ = \ (\mathsf{exl} \bullet (f \vartriangle h)) \triangledown (\mathsf{exl} \bullet (g \vartriangle k)) \\
 & \wedge\ h \triangledown k \ = \ (\mathsf{exr} \bullet (f \vartriangle h)) \triangledown (\mathsf{exr} \bullet (g \vartriangle k)) \\
\equiv & \{ \quad \text{injectivity of } \triangledown \quad \} \\
 & \quad f \ = \ \mathsf{exl} \bullet (f \vartriangle h) \ \wedge \ g \ = \ \mathsf{exl} \bullet (g \vartriangle k) \\
 & \wedge\ h \ = \ \mathsf{exr} \bullet (f \vartriangle h) \ \wedge \ k \ = \ \mathsf{exr} \bullet (g \vartriangle k) \\
\equiv & \{ \quad \vartriangle\text{- computation} \quad \} \\
 & \mathsf{true}\ \ .
\end{array}
$$

□

2.7 The most obvious example is multiplication and division in ordinary arithmetic. (Indeed this is where the two-dimensional notation is commonly used.) Addition and subtraction also abide with each other.

Examples in the text are: disjoint sum and composition, and cartesian product and composition. (Indeed all binary functors abide with composition.)

The example used by Hoare was conditionals. The binary operator **if** p, where p is a predicate, (which has two statements as arguments) abides with **if** q, where q is also a predicate.

□

2.11 First, the $\triangledown$-$+$ fusion rule:

$$
\begin{array}{ll}
 & f\triangledown g \;\bullet\; h{+}k \quad = \quad (f{\bullet}h) \triangledown (g{\bullet}k) \\
\equiv & \qquad \{ \quad \triangledown \text{ characterisation} \quad \} \\
 & f\triangledown g \;\bullet\; h{+}k \;\bullet\; \mathsf{inl} \quad = \quad f{\bullet}h{\bullet}\mathsf{inl} \quad \wedge \quad f\triangledown g \;\bullet\; h{+}k \;\bullet\; \mathsf{inr} \quad = \quad g{\bullet}k{\bullet}\mathsf{inr} \\
\equiv & \qquad \{ \quad \text{computation rules (applied four times)} \quad \} \\
 & \mathsf{true} \;\; .
\end{array}
$$

Second, the identity rule:

$$
\begin{array}{ll}
 & \mathsf{id}{+}\mathsf{id} \\
= & \qquad \{ \quad \text{definition of } + \quad \} \\
 & \mathsf{inl}\triangledown\mathsf{inr} \\
= & \qquad \{ \quad \text{above} \quad \} \\
 & \mathsf{id} \;\; .
\end{array}
$$

□

2.12 The pattern functor for Bin is $\mathsf{Exl} + (\mathsf{Exr}{\times}\mathsf{Exr})$ and for Rose is $(\mathsf{Exl} \times (\mathit{List}\ \mathsf{Exr}))$. That is, for Bin it is the binary functor mapping a and z to $a + (z{\times}z)$ and for Rose it is the binary functor mapping a and z to $a \times (\mathit{List}\ z)$.

□

2.13

$$
\begin{array}{ll}
 & \mathsf{even} \bullet \mathsf{succ}\triangledown\mathsf{zero} \quad = \quad \mathsf{not}\triangledown\mathsf{true} \bullet \mathsf{even}{+}\mathsf{1} \\
\equiv & \qquad \{ \quad \triangledown \text{ fusion and } \triangledown\text{-}+ \text{ fusion,} \\
 & \qquad\quad\;\; \text{definition of functor } {+}\mathsf{1} \quad \} \\
 & (\mathsf{even}{\bullet}\mathsf{succ}) \triangledown (\mathsf{even}{\bullet}\mathsf{zero}) \quad = \quad (\mathsf{not}{\bullet}\mathsf{even}) \triangledown (\mathsf{true}{\bullet}\mathsf{id}_1) \\
\equiv & \qquad \{ \quad \mathsf{true}{\bullet}\mathsf{id}_1 = \mathsf{true},\ \triangledown \text{ is injective} \quad \} \\
 & \mathsf{even}{\bullet}\mathsf{succ} \quad = \quad \mathsf{not}{\bullet}\mathsf{even} \quad \wedge \quad \mathsf{even}{\bullet}\mathsf{zero} \quad = \quad \mathsf{true}
\end{array}
$$

$$
\begin{array}{lll}
\equiv & \{ & \text{extensionality, identifying values } \mathsf{zero} \text{ and } \mathsf{true} \\
 & & \text{with functions } \mathsf{zero} \text{ and } \mathsf{true} \text{ with domain } \mathsf{1} \quad \} \\
 & \multicolumn{2}{l}{\forall(n{::}\ \mathsf{even}(\mathsf{succ}\ n) \ = \ \mathsf{not}(\mathsf{even}\ n)) \ \wedge \ \mathsf{even\ zero} \ = \ \mathsf{true} \ .}
\end{array}
$$

□

2.14

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{\mathsf{out}{\bullet}\mathsf{in}} \\
= & \{ & \text{definition of } \mathsf{out} \quad \} \\
 & \multicolumn{2}{l}{(\!|F\mathsf{in}|\!){\bullet}\mathsf{in}} \\
= & \{ & \text{computation rule} \quad \} \\
 & \multicolumn{2}{l}{F\mathsf{in}{\bullet}F(\!|F\mathsf{in}|\!)} \\
= & \{ & F \text{ is a functor} \quad \} \\
 & \multicolumn{2}{l}{F(\mathsf{in}{\bullet}(\!|F\mathsf{in}|\!))} \\
= & \{ & \text{definition of } \mathsf{out} \quad \} \\
 & \multicolumn{2}{l}{F(\mathsf{in}{\bullet}\mathsf{out})} \\
= & \{ & \mathsf{in}{\bullet}\mathsf{out} = \mathsf{id}_{\mu F} \quad \} \\
 & \multicolumn{2}{l}{F\mathsf{id}_{\mu F}} \\
= & \{ & F \text{ is a functor} \quad \} \\
 & \multicolumn{2}{l}{\mathsf{id}_{F\mu F} \ .}
\end{array}
$$

□

2.16 We have

$$NoOfTips \ = \ (\!|1^{\mathsf{K}}\triangledown add0|\!)$$

where $m\ 'add0'\ n = m{+}0{+}n$, and

$$NoOfJoins \ = \ (\!|0^{\mathsf{K}}\triangledown add1|\!)$$

where $m\ 'add1'\ n = m{+}1{+}n$. Now,

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{f \bullet NoOfTips \ = \ NoOfJoins} \\
\Leftarrow & \{ & \text{definitions and fusion} \quad \} \\
 & \multicolumn{2}{l}{f \ \bullet \ 1^{\mathsf{K}}\triangledown add0 \ = \ 0^{\mathsf{K}}\triangledown add1 \ \bullet \ \mathsf{id} + (f{\times}f)} \\
\equiv & \{ & \text{fusion} \quad \} \\
 & \multicolumn{2}{l}{(f \bullet 1^{\mathsf{K}}) \triangledown (f \bullet add0) \ = \ 0^{\mathsf{K}} \triangledown (add1 \bullet f{\times}f)}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \quad \text{injectivity} \quad \} \\
& f \bullet 1^{\mathsf{K}} = 0^{\mathsf{K}} \quad \wedge \quad f \bullet add0 \ = \ add1 \bullet f \times f \\
\equiv & \{ \quad \text{pointwise definitions, for all } m \text{ and } n \quad \} \\
& f1 = 0 \ \wedge \ f(m+n) \ = \ fm+1+fn \\
\Leftarrow & \{ \quad \text{arithmetic, for all } m \quad \} \\
& fm = m-1 \quad .
\end{array}
$$

We conclude that there is always one less join in a Bin than there are tips.
□

2.17

$$
\begin{array}{ll}
& (\!|f|\!) \vartriangle (\!|g|\!) \ \bullet \ \mathsf{in} \quad = \quad \chi \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \\
\equiv & \{ \quad \vartriangle \text{fusion} \quad \} \\
& ((\!|f|\!) \bullet \mathsf{in}) \vartriangle ((\!|g|\!) \bullet \mathsf{in}) \quad = \quad \chi \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \\
\equiv & \{ \quad \text{catamorphism computation} \quad \} \\
& (f \bullet F(\!|f|\!)) \vartriangle (g \bullet F(\!|g|\!)) \quad = \quad \chi \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \\
\equiv & \{ \quad \vartriangle \text{characterisation} \quad \} \\
& \quad f \bullet F(\!|f|\!) \quad = \quad \mathsf{exl} \bullet \chi \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \\
& \wedge \ g \bullet F(\!|g|\!) \quad = \quad \mathsf{exr} \bullet \chi \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \quad .
\end{array}
$$

Once again, we continue with just one of the conjuncts, the other being solved by symmetry.

$$
\begin{array}{ll}
& f \bullet F(\!|f|\!) \quad = \quad \mathsf{exl} \bullet \chi \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \\
\equiv & \{ \quad \text{postulate } \chi = \alpha \vartriangle \beta \quad \} \\
& f \bullet F(\!|f|\!) \quad = \quad \mathsf{exl} \ \bullet \ \alpha \vartriangle \beta \ \bullet \ F((\!|f|\!) \vartriangle (\!|g|\!)) \\
\equiv & \{ \quad \vartriangle \text{computation} \quad \} \\
& f \bullet F(\!|f|\!) \quad = \quad \alpha \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \\
\equiv & \{ \quad \text{postulate } \alpha = f \bullet \gamma \quad \} \\
& f \bullet F(\!|f|\!) \quad = \quad f \bullet \gamma \bullet F((\!|f|\!) \vartriangle (\!|g|\!)) \\
\Leftarrow & \{ \quad F \text{ respects composition, } \vartriangle \text{ computation} \quad \} \\
& \gamma \quad = \quad F\mathsf{exl} \quad .
\end{array}
$$

Combining the two postulates with the final statement, we get

$$
(\!|\chi|\!) \quad = \quad (\!|f|\!) \vartriangle (\!|g|\!) \quad \Leftarrow \quad \chi \quad = \quad (f \bullet F\mathsf{exl}) \vartriangle (g \bullet F\mathsf{exr}) \quad .
$$

□

2.19 Substituting $(a\oslash)$ for F in the catamorphism rule we get the rule:

$$h\bullet(\![\varphi]\!) \;=\; (\![\psi]\!) \quad\Leftarrow\quad h\bullet\varphi \;=\; \psi\bullet\mathsf{id}\oslash h \quad .$$

This is the fusion rule used below.

$$\begin{array}{ll}
 & (\![f]\!)\bullet(\tau(\oslash)\;g) \;=\; (\![h]\!) \\
\equiv & \qquad\{\qquad \tau(\oslash)\;g = (\![\mathsf{in}\bullet g\oslash\mathsf{id}]\!) \quad\} \\
 & (\![f]\!)\bullet(\![\mathsf{in}\bullet g\oslash\mathsf{id}]\!) \;=\; (\![h]\!) \\
\Leftarrow & \qquad\{\qquad \text{fusion rule} \quad\} \\
 & (\![f]\!)\bullet\mathsf{in}\bullet g\oslash\mathsf{id} \;=\; h\bullet\mathsf{id}\oslash(\![f]\!) \\
\equiv & \qquad\{\qquad \text{catamorphism computation} \quad\} \\
 & f\bullet\mathsf{id}\oslash(\![f]\!)\bullet g\oslash\mathsf{id} \;=\; h\bullet\mathsf{id}\oslash(\![f]\!) \\
\equiv & \qquad\{\qquad \oslash \text{ is a binary functor. Thus,} \\
 & \qquad\qquad\quad \mathsf{id}\oslash(\![f]\!)\bullet g\oslash\mathsf{id} \;=\; g\oslash(\![f]\!) \;=\; g\oslash\mathsf{id}\bullet\mathsf{id}\oslash(\![f]\!) \quad\} \\
 & f\bullet g\oslash\mathsf{id}\bullet\mathsf{id}\oslash(\![f]\!) \;=\; h\bullet\mathsf{id}\oslash(\![f]\!) \\
\Leftarrow & \qquad\{\qquad \text{cancellation} \quad\} \\
 & f\bullet g\oslash\mathsf{id} \;=\; h \quad .
\end{array}$$

We have thus established the rule:

$$(\![f]\!)\bullet(\tau(\oslash)\;g) \;=\; (\![f\bullet g\oslash\mathsf{id}]\!) \quad .$$

□

2.20 First,

$$\begin{array}{ll}
 & \tau(\oslash)\;\mathsf{id}_a \\
= & \qquad\{\qquad \text{definition} \quad\} \\
 & (\![\mathsf{in}\bullet\mathsf{id}\oslash\mathsf{id}]\!) \\
= & \qquad\{\qquad \oslash \text{ respects identities,} \\
 & \qquad\qquad\quad \text{identity is the unit of composition} \quad\} \\
 & (\![\mathsf{in}]\!) \\
= & \qquad\{\qquad \text{identity rule} \quad\} \\
 & \mathsf{id}_{\tau(\oslash)\;a} \quad .
\end{array}$$

Second,

$$
\begin{array}{ll}
 & \tau(\oslash)\ (f{\bullet}g) \\
= & \qquad \{ \qquad \text{definition} \quad \} \\
 & (\!|\mathsf{in} \bullet (f{\bullet}g){\oslash}\mathsf{id}|\!) \\
= & \qquad \{ \qquad \oslash \text{ respects identities,} \\
 & \qquad\qquad\quad \text{identity is the unit of composition} \quad \} \\
 & (\!|\mathsf{in} \bullet f{\oslash}\mathsf{id} \bullet g{\oslash}\mathsf{id}|\!) \\
= & \qquad \{ \qquad \text{exercise 2.19} \quad \} \\
 & (\!|\mathsf{in} \bullet f{\oslash}\mathsf{id}|\!) \bullet (\tau(\oslash)\ g) \\
= & \qquad \{ \qquad \text{definition} \quad \} \\
 & (\tau(\oslash)\ f) \bullet (\tau(\oslash)\ g) \quad .
\end{array}
$$

□

4.1 To express Mf we use the last of the three monad equalities:

$$
\begin{array}{ll}
 & Mf \\
= & \qquad \{ \qquad \text{identities} \quad \} \\
 & \eta \diamond (Mf \bullet \mathsf{id}) \\
= & \qquad \{ \qquad \text{monad equality} \quad \} \\
 & (\eta \bullet f) \diamond \mathsf{id} \quad .
\end{array}
$$

Using $\mu = \mathsf{id}{\diamond}\mathsf{id}$, we obtain that, for the functor Set,

$$\mu x = \{y\colon y{\in}x\colon y\} \quad .$$

The equalities are proven as follows: First,

$$
\begin{array}{ll}
 & \mu \bullet M\mu \\
= & \qquad \{ \qquad \mu = \mathsf{id}{\diamond}\mathsf{id} \quad \} \\
 & (\mathsf{id}{\diamond}\mathsf{id}) \bullet M\mu \\
= & \qquad \{ \qquad \text{2nd monad equality, } \mathsf{id} \text{ is identity of composition} \quad \} \\
 & \mathsf{id}{\diamond}M\mu \\
= & \qquad \{ \qquad \mathsf{id} \text{ is identity of composition,} \\
 & \qquad\qquad\quad \text{3rd monad equality, } \mathsf{id} \text{ is identity of composition} \quad \} \\
 & \mu{\diamond}\mathsf{id} \\
= & \qquad \{ \qquad \mu = \mathsf{id}{\diamond}\mathsf{id}, \text{ Kleisli composition is associative,} \\
 & \qquad\qquad\quad \mu = \mathsf{id}{\diamond}\mathsf{id} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{id}\diamond\mu \\
= & \{ \quad \mathsf{id} \text{ is identity of composition, 2nd monad equality} \quad \} \\
 & (\mathsf{id}\diamond\mathsf{id}) \bullet \mu \\
= & \{ \quad \mu = \mathsf{id}\diamond\mathsf{id} \quad \} \\
 & \mu \bullet \mu \quad .
\end{array}
$$

Second,

$$
\begin{array}{ll}
 & \mu \bullet \eta \\
= & \{ \quad \mu = \mathsf{id}\diamond\mathsf{id}, \text{ 2nd monad equality} \quad \} \\
 & \mathsf{id}\diamond\eta \\
= & \{ \quad \eta \text{ is unit of Kleisli composition} \quad \} \\
 & \mathsf{id} \quad .
\end{array}
$$

Third,

$$
\begin{array}{ll}
 & \mu \bullet M\eta \\
= & \{ \quad \mu = \mathsf{id}\diamond\mathsf{id}, \text{ 2nd monad equality} \quad \} \\
 & \mathsf{id}\diamond M\eta \\
= & \{ \quad \mathsf{id} \text{ is identity of composition, 3rd monad equality,} \\
 & \quad\;\; \mathsf{id} \text{ is identity of composition} \quad \} \\
 & \eta\diamond\mathsf{id} \\
= & \{ \quad \eta \text{ is unit of Kleisli composition} \quad \} \\
 & \mathsf{id} \quad .
\end{array}
$$

$\square$

4.2 $(a\times)^\star 1$ is *List a*.
$\square$

4.3 Since $(1+)^\star\perp\!\!\!\perp = \mathbb{N}$ we obtain from the fusion theorem that $(1+)^\star V \;=\; \mathbb{N}\times(1+V)$. Specifically,

$$
\begin{array}{ll}
 & \mathbb{N}\times(1+V) \text{ is an initial } X{::}\ (1+X)+V \text{ algebra} \\
\Leftarrow & \{ \quad \text{fusion, } \mathbb{N} \text{ is an initial } 1+ \text{ algebra} \quad \} \\
 & \forall(X{::}\ \ (1+X)\times(1+V) \cong (1+(X\times(1+V)))+V) \\
\Leftarrow & \{ \quad \mathsf{rig} \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

The witness to the last step, rig, is a natural isomorphism of type

$$(1{+}X){\times}(1{+}Y) \leftarrow (1{+}(X{\times}(1{+}Y))){+}Y \quad .$$

It is easily constructed:

$$\mathsf{rig} \quad = \quad ((\mathsf{inl}{\vartriangle}\mathsf{inl}) \triangledown (\mathsf{inr}{\times}\mathsf{id})) \triangledown ((\mathsf{inl}{\bullet}!){\vartriangle}\mathsf{inr}) \quad .$$

The initial algebra is $\mathsf{in}_{(1+)^\star} \quad = \quad (\mathsf{zero}{\triangledown}\mathsf{succ}) \times \mathsf{id} \;\bullet\; \mathsf{rig}$.
□

4.4

$$\begin{array}{ll}
 & f{\diamond}(g{\diamond}h) \;=\; (f{\diamond}g){\diamond}h \\
\equiv & \qquad \{ \quad \text{definition} \quad \} \\
 & (\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown f|\!) \bullet (g{\diamond}h) = (\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown (f{\diamond}g)|\!) \bullet h \\
\equiv & \qquad \{ \quad \text{definition, cancel } {\bullet}h \quad \} \\
 & (\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown f|\!) \bullet (\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown g|\!) \;=\; (\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown (f{\diamond}g)|\!) \\
\Leftarrow & \qquad \{ \quad \text{fusion} \quad \} \\
 & (\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown f|\!) \;\bullet\; (\mathsf{in} \bullet \mathsf{inl}) \triangledown g \;=\; (\mathsf{in} \bullet \mathsf{inl}) \triangledown (f{\diamond}g) \;\bullet\; F(\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown f|\!) + \mathsf{id} \\
\equiv & \qquad \{ \quad \text{fusion properties of disjoint sum,} \\
 & \qquad \quad\;\; \triangledown \text{ is injective} \quad \} \\
 & \quad (\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown f|\!) \bullet \mathsf{in} \bullet \mathsf{inl} \;=\; \mathsf{in} \bullet \mathsf{inl} \bullet F(\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown f|\!) \\
 & \wedge\; (\!|(\mathsf{in} \bullet \mathsf{inl}) \triangledown f|\!) \bullet g \;=\; f{\diamond}g \\
\equiv & \qquad \{ \quad \text{computation laws, definition of } f{\diamond}g \quad \} \\
 & \mathsf{true} \quad .
\end{array}$$

The verification that embr is its neutral element is a straightforward use of the computation rules.
□

5.3 Substituting the identity function for h and l, we get

$$f_0 \diamond (Mk \bullet g_1) \quad = \quad (f_0 \bullet k) \diamond g_1 \quad .$$

Substituting the identity function for k and h, we get

$$(f_0 \diamond g_0) \bullet l \quad = \quad f_1 \diamond (g_0 \bullet l) \quad .$$

□

5.4 Suppose $\circ$ is a function that has the same polymorphic type as function composition. Then, if it satisfies the parametricity property of composition, it is the case that, for all relations R, S and T and all functions f_0, f_1, g_0 and g_1, if

$$(f_0\ ,\ f_1)\ \in\ R \leftarrow S \quad \wedge \quad (g_0\ ,\ g_1)\ \in\ S \leftarrow T$$

then

$$(f_0 \circ f_1\ \ ,\ \ g_0 \circ g_1)\ \ \in\ \ R \leftarrow T\ \ .$$

Take R to be the singleton set $\{(f(gc)\ ,\ f(gc))\}$, S to be the singleton set $\{(gc\ ,\ gc)\}$ and T to be the singleton set $\{(c,c)\}$. Then $(f\ ,\ f) \in R \leftarrow S$ and $(g\ ,\ g) \in S \leftarrow T$. So $(f \circ g\ ,\ f \circ g)\ \in\ R \leftarrow T$. That is, $(f \circ g)(c) = f(gc)$. Thus, by extensionality, $f \circ g = f \bullet g$. The parametricity property does indeed uniquely characterise function composition!
□

5.5 The type of an F-catamorphism is

$$(a \leftarrow \mu F) \leftarrow (a \leftarrow Fa)\ \ .$$

The free theorem is thus that, for all relations R and all functions f and g, if

$$(f\ ,\ g)\ \in\ R \leftarrow FR$$

then

$$(([f])\ ,\ ([g]))\ \ \in\ \ R \leftarrow \mathsf{id}_{\mu F}\ \ .$$

Taking R to be a function h and use the point-free definition of $\leftarrow$, this is the statement that

$$([f])\ =\ h \bullet ([g]) \quad \Leftarrow \quad f \bullet Fh\ =\ h \bullet g\ \ .$$

□

5.15 Instantiating F to $(1+)$ we get

$$
\begin{array}{cl}
 & \mathsf{occurs\_properly\_in}_{1+} \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & (\mathsf{mem}_{1+} \bullet (\mathsf{in}_{(1+)^\star} \bullet \mathsf{inl})^\cup)^+ \\
= & \quad \{ \quad \mathsf{mem}_{1+} = \mathsf{inr}^\cup, \quad \mathsf{in}_{(1+)^\star} \ = \ (\mathsf{zero} \triangledown \mathsf{succ}) \times \mathsf{id} \bullet \mathsf{rig} \quad \} \\
 & ((\mathsf{zero} \triangledown \mathsf{succ}) \times \mathsf{id} \bullet \mathsf{rig} \bullet \mathsf{inl} \bullet \mathsf{inr})^{\cup +} \\
= & \quad \{ \quad \text{definition of } \mathsf{rig}\text{, computation} \quad \} \\
 & (\mathsf{succ} \times \mathsf{id})^{\cup +}\ \ .
\end{array}
$$

A pair (m,x) "occurs properly in" a pair (n,y) if $m < n$ and $x = y$. This particular instance of occurs_properly_in is thus proper in the sense that if u "occurs properly in" v then u and v are not equal.

$F^\star 1$ is $\mathsf{List}.A$, membership is the projection exr and occurs_properly_in is the relation "is a (proper) tail of".
□

Bibliography

[1] C.J. Aarts, R.C. Backhouse, P. Hoogendijk, T.S. Voermans, and J. van der Woude. A relational theory of datatypes. Available via World-Wide Web at `http://www.win.tue.nl/cs/wp/papers`, September 1992.

[2] R. C. Backhouse, M. Bijsterveld, R. van Geldrop, and J.C.S.P. van der Woude. Category theory as coherently constructive lattice theory. Department of Mathematics and Computing Science, Eindhoven University of Technology. Working document. Available via World-Wide Web at `http://www.win.tue.nl/cs/wp/papers`, Last revision: March 1997, 146 pages, 1995.

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

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

[5] R.C. Backhouse and B.A. Carré. Regular algebra applied to path-finding problems. *Journal of the Institute of Mathematics and its Applications*, 15:161–186, 1975.

[6] Roland Backhouse. Fixed point calculus applied to generic programming: Part 1. In Zoltan Esik, editor, *Proceedings, Workshop on Fixed Points in Computer Science*, August 1998.

[7] Roland C. Backhouse, J.P.H.W. van den Eijnde, and A.J.M. van Gasteren. Calculating path algorithms. *Science of Computer Programming*, 22(1–2):3–19, 1994.

[8] G. Bellè, C.B. Jay, and E. Moggi. Functorial ML. In *PLILP96*, volume 1140 of *LNCS* . Springer-Verlag, 1996.

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

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

[11] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987. NATO ASI Series, vol. F36.

[12] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.

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

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

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

[16] R.W. Floyd. Algorithm 97. Shortest Path. *Comm. ACM*, 5(6):345, June 1962.

[17] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, The Netherlands, 1992.

[18] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

[19] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

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

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

[24] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[25] P. Jansson and J. Jeuring. Polytypic unification. *Journal of Functional Programming*, 1998. In press.

[26] Patrik Jansson. Functional polytypic programming — use and implementation. Technical report, Chalmers Univ. of Tech., Sweden, 1997. Lic. thesis. Available from `http://www.cs.chalmers.se/~patrikj/lic/`.

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

[28] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. Extended version of [8] in press for Journal of Functional Programming '98, 1998.

[29] 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.

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

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

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

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

[34] L. Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[35] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

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

[37] A. Pardo. Monadic corecursion —definition, fusion laws, and applications—. *Electronic Notes in Theoretical Computer Science*, 11, 1998.

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

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

[40] B. Roy. Transitivité et connexité. *C.R. Acad. Sci.*, 249:216, 1959.

[41] Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types*. PhD thesis, University of Michigan, 1992.

[42] Tim Sheard. Automatic generation and use of abstract structure operators. *ACM TOPLAS*, 13(4):531–557, 1991.

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

[44] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, 1962.

Glossary of Symbols

$F^\star$	substitution monad	60, 63
$R^\cup$	converse of R	76
x^{K}	constant functor	20
μF	carrier of initial algebra	28
$\mathsf{1}$	unit type	10, 20
$\tau(\oslash)$	type functor	39
η	Kleisli identity	62
$\leftarrow$	function space	10
$\leftarrow$	on relations	72
$+$	disjoint sum functor	21
$\triangledown$	"junk"	21
$\vartriangle$	"split"	24
$\times$	cartesian product	24
$\times$	of relations	72
$\downarrow$	minimum operator	4
$\uparrow$	maximum operator	4
$\oslash$	binary functor	38
$\diamond$	Kleisli composition	61
Bin	binary tree	17
even	Boolean function	30
exl	project left	24
exr	project right	24
foldr	Haskell function	31
Id	identity functor	20
inl	inject left	21
inr	inject right	21
in	initial algebra	28
$L\ a$	pattern functor of $List\ a$	29
$List$	list datatype	17
map	higher order function	17
$Maybe$	datatype	17
N	pattern functor of Nat	28
Nat	Peano numbers	11
Par	extraction functor	20, 44
Rec	extraction functor	20
$Rose$	datatype	17
source	(of an edge)	3
subst	substitution	64
succ	successor function	11
sum	function	13
target	(of an edge)	3

Index