Rijks*universiteit* Groningen

# Algebraic Data Types and Program Transformation

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus Dr. L.J. Engels
in het openbaar te verdedigen op
vrijdag 14 september 1990
des namiddags te 1.00 uur

door

**Grant Reynold Malcolm**
geboren op 9 september 1962
te Dundee

Promotor: prof. dr. R.C. Backhouse

## Acknowledgements

I would like to take this opportunity to thank my *promotor*, Roland Backhouse, firstly for accepting me as a Ph.D. student, and further for his help and encouragement in producing this thesis. I can only hope that the following pages reflect some of the many things I have learnt from him, particularly from his insistence on the importance of presentation. I am also grateful to his family for the generosity and hospitality they showed me on my first coming to Groningen.

Members of the type theory group at Groningen University, Peter de Bruin, Albert Thijs, Ed Voermans and particularly Paul Chisholm, gave useful criticism of earlier versions and drafts of this work, which has led to its being considerably improved. I am also grateful to Wim Hesselink and Lambert Meertens for their careful and critical reading of earlier drafts of this thesis; my interest in the topic is moreover largely due to the latter's work on constructive algorithmics.

Special thanks are due to Jaap van der Woude, for his friendship and his annoying habit of always being able to find a more elegant proof: I am only sorry that I haven't had time to incorporate all of the suggestions he made for improving this thesis. Thanks also to Julie, who came to Holland.

## Contents

## Chapter 1

# Introduction

The derivation and proof of computer programs requires the mastering of a great body of relevant details. These details arise in large part from the essential formality of the enterprise: the specification of the problem, the proof of correctness of the algorithm which solves this problem, and the implementation of the algorithm in the form of a program, all of these demand a high degree of formality. Moreover, the requirement that the algorithm provides an efficient solution means that the programmer must be aware of the details of how the target machine stores and manipulates data. The fact that all of these aspects — specification, proof, implementation, and the inner workings of the target machine — interact with each other seems only to increase the possibility of making oversights.

Methodologies for program derivation typically attempt to simplify the process by decomposing the task in such a way that a smaller body of details becomes relevant to each subtask: for example, transformational techniques, by which a correct algorithm is transformed into an equivalent but more efficient algorithm, separate the issues of correctness and efficiency. Yet within each subtask, the programmer glosses over any details at his own peril. This problem is especially acute in the construction of formal proofs, where the transition from what we might call a "semi-formal" proof to a formal proof involves a ten-fold or greater increase in size (this statistic is taken from Backhouse's discussion of the problem in [5]). Since the reliability of the programs we produce is proportional to the rigour with which they are proven, the feasibility of constructing formal proofs is of considerable importance.

This thesis is concerned with the rôle that type structure plays in the process of deriving, proving, and transforming programs. Any theory of data types in computing science should satisfy certain basic requirements: that it provide a clear notation for defining type structures; that it provide mechanisms for deriving programs which

operate on the defined types; and that the notation for these programs lend itself to formal manipulation. This last requirement is perhaps the most important, insofar as it concerns the feasibility of constructing formal proofs. We seek a program notation with which we can calculate — the importance of calculation in constructing concise proofs has been set out convincingly by, amongst others, Backhouse [5], Dijkstra and Scholten [16], Goguen [20] and Meertens [35]. We shall not rehearse their arguments here, but rather let the examples of the following chapters speak for themselves.

We investigate a paradigm of type-definition due to Hagino (see his [21]), an important aspect of which is that types are characterised by a universal property, which prescribes the construction of recursive functions on the defined type. (Slightly abusing standard terminology, we shall refer to a type's universal property as its "unique extension property".) A full account is given in chapter 2, but we note here that the unique extension property allows the construction of concise, calculational proofs: our aim in the body of this thesis is to present a systematic exploitation of such uniqueness properties, particularly in transforming and proving properties of programs. The proofs that we give, while generally short, exhibit a high degree of formality and explicitness. Although we do not tackle any problems of great complexity, the ease and brevity with which simple examples can be treated argue strongly for the feasibility of constructing formal proofs.

Unique extension properties play a central rôle in the Bird-Meertens formalism, a mathematical approach to program derivation and transformation (good introductions can be found in Meertens [34], Bird [9] and Backhouse [4]). Until recently, most of the research into this formalism concentrated on one data structure, the type of finite lists. The algebraic structure of the type was exploited in various ways, most usefully in deriving fundamental operations on the type and in deriving transformation rules by means of which programs constructed from these primitive operations could be transformed into more efficient programs. Another important consequence of the algebraic approach was the development of a concise notation which favoured a calculational style of proof. The work presented in this thesis arose from an attempt to extend the Bird-Meertens formalism to other data types, and to capture in a more general setting the above exploitation of the algebraic structure of data types.

Section 1.1 below gives an example of the uses of unique extension properties in deriving some simple programs; the section also introduces some of the notation and basic notions that we use in subsequent chapters. Chapter 2 gives an account of Hagino types, together with some examples. A simple consequence of the unique extension property, called the "promotion theorem," is proven for an arbitrary data



type. This theorem provides a useful proof technique which is exploited throughout the thesis. The chapter also presents a simple mechanism for defining parameterised data types, and explores the properties of certain fundamental operations on these types, as well as identifying a class of types which enjoy a particularly rich algebraic structure. Chapter 3 extends these results to infinite data structures such as infinite lists. We show that many basic properties of finite types can be reformulated in a very simple way for infinite types; in particular, we give the promotion theorem for an arbitrary infinite data structure.

Chapter 4 reports on recent work instigated by Backhouse [2], which extends the results of chapter 2 to a relational setting: the functions of the latter chapter are considered more generally as relations. In particular, Backhouse has shown that the unique extension properties and promotion theorems of Hagino types also hold in the more general setting of relations. Among the advantages that accrue from considering relations is the ability to reason calculationally about subtypes: we present some examples for a type of ordered binary trees. Finally, chapter 5 is of a more theoretical nature. Hagino types can be constructed as fixed points of functions from types to types ("functors"). We discuss conditions which guarantee the existence of these fixed points and prove that functors constructed from parameterised Hagino types satisfy these conditions. We construct thereby a recursively defined class of functors whose fixed points do exist, and which may therefore be used in defining Hagino types.

## 1.1 The Pair-calculus

The purpose of this section is twofold: to illustrate the means by which a unique extension property can be exploited in deriving and proving properties of programs, and also to provide a leisurely introduction to the notation and concepts used in later chapters.

We shall be primarily concerned with data types: collections of objects which share a common structure, such as the type of natural numbers, or the type of lists of natural numbers. If $A$ is a type and $a$ is an object of type $A$, we denote this by writing $a \in A$. A very common type is the type of functions: the type of functions with domain $B$ and range $A$ is denoted by $B \longleftarrow A$. We shall impose the restriction that all functions be *total*, that is, if $f \in B \longleftarrow A$ and $a \in A$, then $f$ may be applied to $a$, and will always return a well-defined value of type $B$. We denote application of functions by an infix dot, thus: $f.a$.

Another means of constructing types from other types is by forming the cartesian product: if $A$ and $B$ are types, then their cartesian product, $A \times B$, is a type whose objects are pairs $(a, b)$, where $a \in A$ and $b \in B$. The type has two primitive operations, the projections $\ll \in A \longleftarrow A \times B$ ("first") and $\gg \in B \longleftarrow A \times B$ ("second"). We shall refer to functions defined on cartesian product types as "binary operators." For binary operators we make an exception to our notation for function application, writing $a \oplus b$ instead of $\oplus.(a, b)$ (we use the symbols "$\oplus$," "$\otimes$," etc., as variables standing for arbitrary binary operators). In this infix notation, the projection operators are defined by: for all $a \in A$ and $b \in B$,

$$\begin{aligned} a \ll b &= a \\ a \gg b &= b \end{aligned}$$

These projections are our first examples of polymorphic functions, polymorphic in the sense that $\ll \in A \longleftarrow A \times B$ for *all* types $A$ and $B$. The identity function, I, is another example of a polymorphic function, since $\mathrm{I} \in A \longleftarrow A$ for all types $A$. If we need to make it clear that we are referring to the identity function on a particular type, then we write that type as a subscript, thus: $\mathrm{I}_{\mathbb{N}}$, the identity function on the natural numbers.

Some further conventions regarding binary operators are as follow. If $\oplus$ has type $C \longleftarrow A \times B$ and $a \in A$, then we write $(a\oplus) \in C \longleftarrow B$ for the function such that

$$(a\oplus).b = a \oplus b.$$

Similarly, if $b \in B$, then we have the function $(\oplus b) \in C \longleftarrow A$. We also write $\hat{\oplus}$ ("$\oplus$ lifted") for the operator of type $(C \longleftarrow D) \longleftarrow (A \longleftarrow D) \times (B \longleftarrow D)$, such that for all $f \in A \longleftarrow D$, $g \in B \longleftarrow D$ and $x \in D$,

$$(f \hat{\oplus} g).x = f.x \oplus g.x.$$

Here and throughout, we adopt the convention that function application is more binding than infix binary application, so the right-hand side of this equation is to be read as

$$(f.x) \oplus (g.x).$$

Finally, if $\oplus \in A \longleftarrow A \times A$, we write $1_{\oplus}$ for the unique object (if it exists) of type $A$ such that for all $a \in A$,

$$1_{\oplus} \oplus a = a = a \oplus 1_{\oplus}.$$

For example, $1_{+} = 0$ and $1_{\circ} = \mathrm{I}$, where "$\circ$" is the composition operator: for $f \in C \longleftarrow B$ and $g \in B \longleftarrow A$, we denote their composition by

$$f \circ g \in C \longleftarrow A.$$



Let us return to cartesian product and examine the converse of binary operators; i.e., functions which construct pairs of values. Given functions $f \in B \longleftarrow A$ and $g \in C \longleftarrow A$, we can construct the function $\langle f, g \rangle \in B \times C \longleftarrow A$, which, given argument $a \in A$, returns the pair of values $(f.a,\ g.a)$. That is,

$$\langle f, g \rangle.a = (f.a,\ g.a) \qquad \text{for all } a \in A. \tag{1.1}$$

This is one way in which to define the function $\langle f, g \rangle$; very often, however, we shall define functions and types by stating a property which *uniquely characterises* that type or function. Thus, for example,

**Definition 1.1.1** For functions $f \in B \longleftarrow A$ and $g \in C \longleftarrow A$, the function $\langle f, g \rangle \in B \times C \longleftarrow A$ is characterised by the following uniqueness property.
For all $h \in B \times C \longleftarrow A$,

$$h = \langle f, g \rangle \quad \equiv \quad \ll \circ h = f \ \wedge \ \gg \circ h = g. \tag{1.2}$$

□

We refer to the function $\langle f, g \rangle$ as the unique extension of $f$ and $g$. The above definition is thus our first example of a unique extension property. The advantage of the definition given in (1.1) is that it makes it quite clear how the function $\langle f, g \rangle$ can be implemented, given implementations of $f$ and $g$; in definition 1.1.1, however, this is less clear. The problem recurs in later chapters when we come to define types by means of unique extension properties: it is not at all obvious that the type so defined exists (we discuss sufficient conditions for existence in chapter 5). The advantage of the second definition is that it lends itself better to deriving properties and to a calculational style of proof — this will be the subject of the remainder of this section (and, indeed, of the thesis).

For example, by taking $h := \langle f, g \rangle$ , the left-hand side of (1.2) becomes true, which gives us the two laws:

$$\ll \circ \langle f, g \rangle = f \tag{1.3}$$

$$\gg \circ \langle f, g \rangle = g \tag{1.4}$$

Similarly, by taking $f := \ll \circ h$ and $g := \gg \circ h$ so that the right-hand side becomes true, we obtain, for all $h$:

$$h = \langle \ll \circ h,\ \gg \circ h \rangle. \tag{1.5}$$

The significance of this is that all functions $h \in B \times C \longleftarrow A$ can be expressed as a function of the form $\langle f, g \rangle$. In particular, by taking $h := \mathrm{I}$, we have

$$\mathrm{I} = \langle \ll, \gg \rangle. \tag{1.6}$$

The same technique can be used to prove another useful property:

**Property 1.1.2** $\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle$.

**Proof:**

$$\begin{array}{ll} & \langle f, g \rangle \circ h \\ = & \quad \{ \ (1.5) \ \} \\ & \langle \ll \circ \langle f, g \rangle \circ h \ , \gg \circ \langle f, g \rangle \circ h \rangle \\ = & \quad \{ \ (1.3), (1.4) \ \} \\ & \langle f \circ h, g \circ h \rangle \end{array}$$

□

Operations which apply functions pointwise to elements of a structure, but leave the shape of the structure unchanged are discussed generally in section 2.3.1 below. For cartesian product we can define a pointwise operation as follows. The symbol "$\triangleq$" denotes definitional equality.

**Definition 1.1.3** For $f \in C \longleftarrow A$ and $g \in D \longleftarrow B$, define

$$f \times g \triangleq \langle f \circ \ll, g \circ \gg \rangle \in C \times D \longleftarrow A \times B.$$

□

Thus $f \times g$ takes the pair $(a, b) \in A \times B$ to the pair $(f.a, g.b)$. Such pointwise operations are interesting objects of study, for a pointwise operation can be defined for all parameterised types (the type $A \times B$ is parameterised by the types $A$ and $B$). Moreover, pointwise operations all enjoy certain properties; one such property is that the pointwise application of the identity function does not alter the argument in any way:

**Property 1.1.4** $\mathrm{I} \times \mathrm{I} = \mathrm{I}$.

**Proof:**



$$\begin{array}{ll} & \mathrm{I} \times \mathrm{I} \\ = & \quad \{ \text{ definition 1.1.3 } \} \\ & \langle \mathrm{I} \circ \ll, \mathrm{I} \circ \gg \rangle \\ = & \quad \{ \text{ identity } \} \\ & \langle \ll, \gg \rangle \\ = & \quad \{ \ (1.6) \ \} \\ & \mathrm{I} \end{array}$$

□

Another property shared by pointwise operators is that they distribute over composition. We use the following property as a lemma to prove this, but the property is useful in its own right (in later chapters we shall make frequent use of the properties in this section; so much so that it will be convenient in proofs to refer generally to these properties with the bare hint, "calculus"). To keep the number of parentheses in check, we let $\times$ bind more tightly than composition.

**Property 1.1.5** $f \times g \circ \langle h, i \rangle = \langle f \circ h, g \circ i \rangle$.

**Proof:**

$$\begin{array}{ll} & f \times g \circ \langle h, i \rangle \\ = & \quad \{ \text{ definition 1.1.3 } \} \\ & \langle f \circ \ll, g \circ \gg \rangle \circ \langle h, i \rangle \\ = & \quad \{ \text{ property 1.1.2 } \} \\ & \langle f \circ \ll \circ \langle h, i \rangle \ , \ g \circ \gg \circ \langle h, i \rangle \rangle \\ = & \quad \{ \ (1.3), (1.4) \ \} \\ & \langle f \circ h, g \circ i \rangle \end{array}$$

□

Taking $h := h \circ \ll$ and $i := i \circ \gg$, we obtain

**Corollary 1.1.6** $f \times g \circ h \times i = (f \circ h) \times (g \circ i)$.

□

The fact that these properties are common to all parameterised types suggests that it will be useful to give a name to this collection of properties. This motivates

**Definition 1.1.7 (type-functor)** A *type-functor* is a function F from types to types with a corresponding action on functions which respects identity and composition. That is,

- for all types $A$, the application of F to $A$, denoted by $A\mathsf{F}$, is a type;
- for all functions $f \in B \longleftarrow A$, there is a function $f\mathsf{F} \in B\mathsf{F} \longleftarrow A\mathsf{F}$;
- $\mathrm{I}\mathsf{F} = \mathrm{I}$;
- $(f \circ g)\mathsf{F} = f\mathsf{F} \circ g\mathsf{F}$.

□

This definition may be generalised to type-functors of more than one argument: cartesian product is a case in point. Thus for all types $A$ and $B$, we have the type $A \times B$ (according to the above definition we should write $(A, B)\times$, but we are following here our convention of writing all binary operators as infix operators). Moreover, for all functions $f \in C \longleftarrow A$ and $g \in D \longleftarrow B$ we have the function (again using infix notation)

$$f \times g \in C \times D \longleftarrow A \times B.$$

Property 1.1.4 and corollary 1.1.6 show, respectively, that $\times$ respects identity and composition.

We give some further examples of type-functors, leaving it to the reader to check that these do indeed respect identity and composition.

**Examples 1.1.8**

- For all types $A$, the type of finite lists over $A$, denoted by $A*$, is a type. The operator $*$ becomes a functor by defining $f*$ to be the function which, given a list as argument, returns the list obtained by applying function $f$ "pointwise" to each element in the given list. The functor $*$ is the main reason why we have chosen to use postfix notation for the application of functors.
- The identity functor, I is such that $X\mathrm{I} = X$, for $X$ a type or a function.
- If F and G are type-functors, so too is their composition FG. We maintain the convention that application of postfix operators associates to the left. Thus $A\mathsf{FG}$ is to be read as $(A\mathsf{F})\mathsf{G}$.



- If $\oplus$ is a binary type-functor and $A$ is a type then $(A\oplus)$ is also a type-functor, defined by $B(A\oplus) = A \oplus B$ for type $B$, and $f(A\oplus) = \mathrm{I}_A \oplus f$ for function $f$. Thus for example, $(A\ll)$ is the constant functor returning $A$ when applied to a type, and returning $\mathrm{I}_A$ when applied to a function.
- If $\oplus$ is a binary type-functor and F and G unary type-functors, then $\mathsf{F}\hat{\oplus}\mathsf{G}$ is the type-functor such that $X(\mathsf{F}\hat{\oplus}\mathsf{G}) = X\mathsf{F} \oplus X\mathsf{G}$, for $X$ a type or a function.

□

The notion of type-functor is an example of the more general category-theoretical notion of functor (see, for example, Mac Lane [28]). We borrow many other notions from category theory, but it is not worthwhile to state these definitions in their full generality, since we nearly always use them in the setting of types (in technical terms, we are working in a category whose objects are types and whose arrows are total functions; at some points we find it convenient to think of types as sets, in which case we refer to the category of sets). Because we work mostly in this one setting, we sometimes refer to type-functors simply as functors. Another notion we borrow from category theory is that of natural transformation. In the sense that functors are parameterised types, natural transformations are polymorphic functions.

**Definition 1.1.9 (natural transformation)** A *natural transformation* from a type-functor F to a type-functor G is a polymorphic function $\eta$ such that for all types $A$, $\eta \in A\mathsf{G} \longleftarrow A\mathsf{F}$ and for all functions $f \in B \longleftarrow A$,

$$\eta \circ f\mathsf{F} = f\mathsf{G} \circ \eta.$$

We write this as $\eta : \mathsf{G} \stackrel{\cdot}{\longleftarrow} \mathsf{F}$.
□

For example, from (1.3) and definition 1.1.3, we have that

$$\ll \circ f \times g = f \circ \ll \tag{1.7}$$

and so $\ll : \ll \stackrel{\cdot}{\longleftarrow} \times$, where $\ll$ is used both as the projection function and also as the binary type-functor which always returns its left argument. Similarly, we have $\gg : \gg \stackrel{\cdot}{\longleftarrow} \times$. This notation is especially useful as it encapsulates not only the polymorphic type of the function $\ll$, but also the distributivity property of equation (1.7).

We have introduced enough new notation and concepts; we close this section with an example of how the unique extension property of cartesian product can be used in

deriving programs. We shall show that $\times$ is, in a sense, associative: that is, we shall show that $A\times(B\times C)$ is isomorphic to $(A\times B)\times C$, by constructing a bijection between the two types. We note in passing that in category theory, types which are isomorphic are considered to be "abstractly the same," or "equal up to isomorphism." We shall often make use of this abstract notion of equivalence in blurring the distinction between objects of a type and functions: that is, given a type $\mathbb{1}$ which has only one object, say 1, then for all types $A$, $A$ is isomorphic to the type $A\longleftarrow\mathbb{1}$, since for all $a\in A$, we have the constant function $(a\ll)\in A\longleftarrow\mathbb{1}$, and for all functions $e\in A\longleftarrow\mathbb{1}$ we regain the object of type $A$ by the application $e.1$. It is clear that this construction determines a bijection.

Now, to show the associativity of $\times$, we must first of all find a function

$$h\in A\times(B\times C)\longleftarrow(A\times B)\times C,$$

and then construct its inverse. We might begin by proposing

$$h.((a,b),c) \;=\; (a,(b,c)).$$

We want to give an explicit formulation to $h$, for we shall use that formulation in calculating its inverse, so we abstract on the variables $a$, $b$ and $c$ in the above equation, which gives the functional equalities

$$\begin{aligned} \ll\circ h &= \ll\circ\ll \\ \gg\circ h &= \gg\times\mathrm{I}. \end{aligned}$$

The unique extension property, (1.2), therefore yields

$$h \;=\; \langle\ll\circ\ll\,,\gg\times\mathrm{I}\rangle.$$

We now turn to finding the inverse of this function. Although for this simple example it is obvious what that inverse should be, it is possible to combine the finding of the inverse and the proof that it is indeed an inverse in a direct calculation. By the comment following equation (1.5), we should look for a function of the form $\langle f,g\rangle$. We shall try to use the requirement that $\langle f,g\rangle$ be the inverse of $\langle\ll\circ\ll\,,\gg\times\mathrm{I}\rangle$ to calculate suitable definitions of $f$ and $g$. In particular, we shall make use of the following straightforward corollary to the unique extension property:

$$(1.8)\qquad \langle f_1,g_1\rangle = \langle f_2,g_2\rangle \;\equiv\; f_1=f_2\;\wedge\;g_1=g_2.$$

Now, the requirement that $\langle f,g\rangle$ be the inverse of $\langle\ll\circ\ll\,,\gg\times\mathrm{I}\rangle$ is resolved in the following calculation.



$$\begin{aligned}
& \langle\ll\circ\ll\,,\gg\times\mathrm{I}\rangle\circ\langle f,g\rangle \;=\; \mathrm{I} \\
\equiv\;& \quad \{\text{ property 1.1.2, (1.6) }\} \\
& \langle\ll\circ\ll\circ\langle f,g\rangle\,,\gg\times\mathrm{I}\circ\langle f,g\rangle\rangle \;=\; \langle\ll,\gg\rangle \\
\equiv\;& \quad \{\text{ (1.8) }\} \\
& \ll\circ\ll\circ\langle f,g\rangle = \ll \;\wedge\; \gg\times\mathrm{I}\circ\langle f,g\rangle = \gg \\
\equiv\;& \quad \{\text{ (1.3), property 1.1.5 }\} \\
& \ll\circ f = \ll \;\wedge\; \langle\gg\circ f,g\rangle = \gg \\
\equiv\;& \quad \{\text{ (1.5) }\} \\
& \ll\circ f = \ll \;\wedge\; \langle\gg\circ f,g\rangle = \langle\ll\circ\gg,\gg\circ\gg\rangle \\
\equiv\;& \quad \{\text{ (1.8) }\} \\
& \ll\circ f = \ll \;\wedge\; \gg\circ f = \ll\circ\gg \;\wedge\; g = \gg\circ\gg \\
\equiv\;& \quad \{\text{ (1.2) }\} \\
& f = \langle\ll,\ll\circ\gg\rangle \;\wedge\; g = \gg\circ\gg \\
\equiv\;& \quad \{\text{ definition 1.1.3 }\} \\
& f = \mathrm{I}\times\ll \;\wedge\; g = \gg\circ\gg
\end{aligned}$$

We have thus found $\langle\mathrm{I}\times\ll,\gg\circ\gg\rangle$, by means of a very simple calculation. It still remains to verify the other half of the bijection, namely that

$$\langle\mathrm{I}\times\ll,\gg\circ\gg\rangle\circ\langle\ll\circ\ll\,,\gg\times\mathrm{I}\rangle \;=\; \mathrm{I}$$

but this, again, is straightforward and should provide the reader with a chance to put the above properties of cartesian product into practice for himself.

The point of this small exercise was to exhibit the use of uniqueness properties in deriving programs, and in building up a useful body of properties of functions on a data structure.

# Chapter 2

# Initial Data Structures

How much effort should the introduction of a new data type entail? Many programming tasks can be effectively performed using standard data types such as integers, arrays and lists, for which there are well-known programming and proof techniques, but occasionally a need arises for a new or unfamiliar data type. Many programming languages allow the user to define their own types, and it is our belief that this facility should be put to good use: the programmer should be encouraged to explore the properties of various structures in order to find one well-suited to a given problem. A theory of data types for computer science should contrive to make such an exploration as effortless as possible, by highlighting the most useful properties of a given type and offering a notation which facilitates their use. A related point here is that everyone, at some point in their life, is unfamiliar with the standard data types just mentioned: the process of becoming familiar with them should be similarly effortless. Martin-Löf's theory of types (see his [31]) goes quite some way in this direction. A type in his theory is defined by: a number of introduction rules which state how objects of the type are constructed; an elimination rule which both provides a schema for structural induction and prescribes the construction of recursive functions on the type; and a number of computation rules which describe the evaluation of these functions. Moreover, as pointed out by Backhouse [3] and Dybjer [17], the elimination and computation rules can be mechanically derived from the introduction rules, thus in Backhouse's words considerably "reducing the burden of understanding": in introducing a new type, one need provide only the introduction rules; the elimination rule — and hence schemata for recursion and induction — comes, as it were, for free. Another advantage of the theory is that the combining of recursion and induction in the one elimination rule sheds considerable light on the relationship between the two.

The emphasis in Martin-Löf's type theory lies heavily on proof by induction; in

this chapter we consider an alternative paradigm of type definition which lays emphasis more on equational proofs. This paradigm is due to Hagino (see [21]), and we shall refer to the types defined according to this paradigm as Hagino types. As with Martin-Löf types, Hagino types are defined by specifying their constructors; what is obtained "for free" is a unique extension property which prescribes the construction and evaluation of recursive functions (called "catamorphisms") on the defined type. The unique extension property provides a basis for concise and elegant equational proofs: in particular, a corollary of this property, which we call the promotion theorem, turns out to be very useful. In section 2.1 we present Hagino types and our notation for catamorphisms; we also prove the promotion theorem for an arbitrary Hagino type. Section 2.2 gives examples of Hagino types. In section 2.3 we explore the properties of parameterised Hagino types and identify a class of types whose catamorphisms can always be factored into the composition of two catamorphisms, and in the final section of this chapter we examine the algebraic properties of a particular form of catamorphism.

## 2.1 An Algebraic Notation for Type Structures

In typed programming languages such as SML, or in type theories such as Martin-Löf's, types are defined by a number of primitive constructors together with the types of these constructors. In this thesis we shall adopt a similar paradigm of type definition, with this difference only: that the types of the constructors are determined by a type-functor. Thus if we define a type $\mathcal{T}$ by:

$$\mathcal{T} \triangleq \mu(\tau_1 : \mathsf{F}_1, \ldots, \tau_n : \mathsf{F}_n),$$

then we mean by this that the type $\mathcal{T}$ has $n$ constructors $\tau_i$, and the type of each constructor is determined by the type-functor $\mathsf{F}_i$ according to:

$$\tau_i \in \mathcal{T} \longleftarrow \mathcal{T}\mathsf{F}_i.$$

For example the type of lists over a type $A$ could be defined as:

$$A* \triangleq \mu(nil : (\mathbb{1} \ll), \succ : (\times A)),$$

which states that the type $A*$ has two constructors: the empty list $nil \in A* \longleftarrow \mathbb{1}$ and concatenation $\succ \in A* \longleftarrow A* \times A$. (This latter operator is sometimes referred to as "snoc," the reverse of "cons," since it takes a list as its left argument.) Implicit in



this paradigm is that the type so defined be the *least* type whose objects are built up from the given constructors (we give a precise meaning to "least" in chapter 5, where we discuss conditions for the existence of defined types): effectively, for the case of $A*$, saying that $nil \in A*$, and if $l \in A*$ and $a \in A$ then $l \succ a \in A*$, *and nothing else is an object of* $A*$. In Martin-Löf's type theory, this minimality property gives rise to an induction principle and a schema for the construction of recursive functions over the type (see Backhouse [3], Backhouse et al. [6] and Dybjer [17]). The minimality property for the types we consider here is called "initiality": in chapter 4 we give an induction principle which is a consequence of initiality in a more general setting; in the present chapter we concentrate on the construction of recursive functions over the defined type. These turn out to be special cases of homomorphisms on the algebras induced by the type functors of the defined type. We begin by describing these algebras.

**Definition 2.1.1 ($\mathcal{T}$-algebras)** The algebra induced by the type-functors $\mathsf{F}_1, \ldots, \mathsf{F}_n$, which we call a *$\mathcal{T}$-algebra*, is a tuple $(P, f_1, \ldots, f_n)$, where $P$ is a type and for each $i$,

$$f_i \in P \longleftarrow P\mathsf{F}_i.$$

A *$\mathcal{T}$-homomorphism* from $\mathcal{T}$-algebra $(P, f_1, \ldots, f_n)$ to $\mathcal{T}$-algebra $(Q, g_1, \ldots, g_n)$ is a function $h \in Q \longleftarrow P$ such that

$$\forall(i :: h \circ f_i = g_i \circ h\mathsf{F}_i).$$

We write this as

$$h : (Q, g_1, \ldots, g_n) \longleftarrow (P, f_1, \ldots, f_n),$$

using a colon in place of the esti symbol ($\in$) to distinguish between homomorphisms of algebras and functions. Clearly, $\mathcal{T}$-homomorphisms are closed under composition. □

**Example 2.1.2 ($A*$-algebras)** $A*$-algebras are triples $(P, d, \otimes)$, where $d \in P \longleftarrow \mathbb{1}$ and $\otimes \in P \longleftarrow P \times A$. An $A*$-homomorphism $h : (Q, e, \oplus) \longleftarrow (P, d, \otimes)$ is a function $h \in Q \longleftarrow P$ such that:

$$h \circ d = e \quad \wedge \quad h \circ \otimes = \oplus \circ h \times \mathrm{I}.$$

These equations are expressed perhaps more familiarly as:

$$h.d = e \quad \wedge \quad \forall(x \in P, y \in A :: h.(x \otimes y) = h.x \oplus y).$$

(In the left conjunct, we have exploited the isomorphism between the types $P$ and $P \longleftarrow \mathbb{1}$, as stated in section 1.1.) In particular,

$$(A*,\ nil \in A* \longleftarrow \mathbb{1},\ \succ \in A* \longleftarrow A* \times A)$$

and

$$(\mathbb{N},\ 0 \in \mathbb{N} \longleftarrow \mathbb{1},\ (+1) \circ \ll \in \mathbb{N} \longleftarrow \mathbb{N} \times A)$$

are $A*$-algebras. The length function $\# \in \mathbb{N} \longleftarrow A*$ is a homomorphism

$$\# : (\mathbb{N}, 0, (+1) \circ \ll) \longleftarrow (A*, nil, \succ),$$

since $\#.nil = 0$ and $\# \circ \succ\ = (+1) \circ \ll \circ \#\times \mathrm{I}$.
□

As suggested by the above example, the type $\mathcal{T}$ together with its constructors is itself a $\mathcal{T}$-algebra; in fact, we complete the definition of the type $\mathcal{T}$ by ascribing to it the property of initiality. Initiality is the minimality property which gives rise to the construction of recursive functions over the type $\mathcal{T}$. The following paradigm of type definition is due to Hagino [21].

**Definition 2.1.3 (initial Hagino type)** The type $\mathcal{T} = \mu(\tau_1 : \mathsf{F}_1, \ldots, \tau_n : \mathsf{F}_n)$ is characterised ("up to isomorphism") by defining $(\mathcal{T}, \tau_1, \ldots, \tau_n)$ to be the *initial* $\mathcal{T}$-algebra; that is, for every $\mathcal{T}$-algebra $(Q, g_1, \ldots, g_n)$, there is exactly one $\mathcal{T}$-homomorphism

$$(\!|g_1, \ldots, g_n|\!)_{\mathcal{T}} : (Q, g_1, \ldots, g_n) \longleftarrow (\mathcal{T}, \tau_1, \ldots, \tau_n).$$

The fact that this is a homomorphism gives the following equations, which prescribe the evaluation of the function: for all $i$,

$$(2.1) \qquad (\!|g_1, \ldots, g_n|\!)_{\mathcal{T}} \circ \tau_i \ = \ g_i \circ (\!|g_1, \ldots, g_n|\!)_{\mathcal{T}} \mathsf{F}_i.$$

When we regard this homomorphism as a function, its uniqueness is expressed by what we shall refer to as the unique extension property: for all $h \in Q \longleftarrow \mathcal{T}$,

$$(2.2) \qquad h = (\!|g_1, \ldots, g_n|\!)_{\mathcal{T}} \quad \equiv \quad \forall(i :: h \circ \tau_i = g_i \circ h\mathsf{F}_i).$$

□



We defer discussion of the existence of Hagino types until chapter 5; the remainder of this chapter is given over to an examination of the algebraic properties of homomorphisms constructed by means of unique extension properties. In order to distinguish these from the more general notion of homomorphism, we follow Meertens' suggestion in [33], and refer to $\mathcal{T}$-homomorphisms of the form $(\!|g_1, \ldots, g_n|\!)_{\mathcal{T}}$ as "catamorphisms." (Meertens derives the term from the Greek *"cata-"*, meaning "according to" and *"morphe"*, meaning "shape" or "structure.") Furthermore, we shall only use the subscripted $\mathcal{T}$ in $(\!|g_1, \ldots, g_n|\!)_{\mathcal{T}}$ to avoid confusion when discussing catamorphisms on several types. When there is no risk of confusion, these subscripts will be omitted. The uniqueness property of catamorphisms gives us immediately the following useful identity:

**Corollary 2.1.4 (identity catamorphism)** $(\!|\tau_1, \ldots, \tau_n|\!) = \mathrm{I} \in \mathcal{T} \longleftarrow \mathcal{T}$.

**Proof:**

$$\begin{array}{ll} & (\!|\tau_1, \ldots, \tau_n|\!) = \mathrm{I} \\ \equiv & \quad \{\ (2.2)\ \} \\ & \forall(i :: \mathrm{I} \circ \tau_i = \tau_i \circ \mathsf{IF}_i) \\ \equiv & \quad \{\ \text{functors preserve identity}\ \} \\ & \mathsf{true} \end{array}$$

□

**Example 2.1.5 (snoc lists)** $(A*, nil, \succ)$ is defined to be the initial $A*$-algebra. This means that for all types $P$ with operations $d \in P$ and $\otimes \in P \longleftarrow P \times A$, there is a unique function $(\!|d, \otimes|\!) \in P \longleftarrow A*$ such that:

$$(2.3) \qquad (\!|d, \otimes|\!).nil \ = \ d$$

$$(2.4) \qquad (\!|d, \otimes|\!) \circ \succ \ = \ \otimes \circ (\!|d, \otimes|\!) \times \mathrm{I}$$

i.e., $(\!|d, \otimes|\!)$ is the unique $A*$-homomorphism $(P, d, \otimes) \longleftarrow (A*, nil, \succ)$. Thus catamorphisms provide a paradigm of recursive definition of functions, and the equations (2.3) and (2.4) specify how these functions are to be evaluated. From property 2.1.4 we obtain that the identity catamorphism is $(\!|nil, \succ|\!)$.
□

The unique extension property proves useful when one wishes to express a function as a catamorphism. Consider the length function on lists: we want to find $d$ and

$\otimes$ such that $\# = (\!|d, \otimes|\!)$. By the unique extension property, this equality holds iff $\#.nil = d$ and $\#.(l \succ a) = \#.l \otimes a$. This in turn leads to the choice of 0 for $d$ and $(+1)\circ\ll$ for $\otimes$, and so $\# = (\!|0, (+1)\circ\ll|\!)$.

The unique extension property also allows us to derive the promotion theorem:

**Theorem 2.1.6 (promotion)** For a type $\mathcal{T} = \mu(\tau_1 : \mathsf{F}_1, \ldots, \tau_n : \mathsf{F}_n)$,

$$h \circ (\!|f_1, \ldots, f_n|\!) = (\!|g_1, \ldots, g_n|\!) \quad \Leftarrow \quad \forall(i :: h \circ f_i = g_i \circ h\mathsf{F}_i).$$

Proof:

$$\begin{array}{ll} & h \circ (\!|f_1, \ldots, f_n|\!) = (\!|g_1, \ldots, g_n|\!) \\ \equiv & \quad \{\ (2.2)\ \} \\ & \forall(i :: h \circ (\!|f_1, \ldots, f_n|\!) \circ \tau_i = g_i \circ (h \circ (\!|f_1, \ldots, f_n|\!))\mathsf{F}_i) \\ \equiv & \quad \{\ (2.1),\ \text{functors respect composition}\ \} \\ & \forall(i :: h \circ f_i \circ (\!|f_1, \ldots, f_n|\!)\mathsf{F}_i = g_i \circ h\mathsf{F}_i \circ (\!|f_1, \ldots, f_n|\!)\mathsf{F}_i) \\ \Leftarrow & \\ & \forall(i :: h \circ f_i = g_i \circ h\mathsf{F}_i). \end{array}$$

□

We state the promotion theorem as a "reversed" implication of the form $P \Leftarrow Q$, as we mostly use promotion in a top-down manner (we maintain, however, the standard terminology and refer to $P$ as the consequent and $Q$ as the antecedent). Typically, we use promotion when we seek to express a function of the form

$$h \circ (\!|f_1, \ldots, f_n|\!)$$

as a catamorphism: promotion decomposes this goal into the subgoals of distributing $h$ over the functions $f_i$.

The promotion theorem for snoc lists is obtained by instantiating the above theorem:

**Example 2.1.7 (snoc lists)**

$$h \circ (\!|e, \oplus|\!) = (\!|d, \otimes|\!) \quad \Leftarrow \quad h \circ e = d \ \wedge \ h \circ \oplus = \otimes \circ h \times \mathrm{I}.$$

Or, equivalently,

$$h \circ (\!|e, \oplus|\!) = (\!|h.e, \otimes|\!) \quad \Leftarrow \quad h \circ \oplus = \otimes \circ h \times \mathrm{I}. \tag{2.5}$$



Whenever a promotion theorem can be simplified like this, we state and use only the simplified form.
□

If we pronounce the antecedent of formula (2.5) as "$h$ is $\oplus$ to $\otimes$ promotable," then by comparing equations (2.4) and (2.5), we see that all catamorphisms $(\!|d, \otimes|\!)$ are $\succ$ to $\otimes$ promotable (a similar observation holds for all types). We give an illustration of this by using the results we have so far obtained to prove three simple properties of list-concatenation: that the empty list is the left- and right unit of concatenation and that concatenation is associative. First we define the concatenation operator $+\!\!+ \in A* \longleftarrow A* \times A*$. The standard recurrence relations for concatenation are $x +\!\!+ nil = x$ and $x +\!\!+ (y \succ a) = (x +\!\!+ y) \succ a$. From these recurrence relations the unique extension property yields the following definition:

**Definition 2.1.8** For $x, y \in A*$, define $x +\!\!+ y \triangleq (\!|x, \succ|\!).y$.
□

The proofs that the empty list is the unit of concatenation are almost laughably short; the proof of associativity is only barely longer. To prove that the empty list is the left unit of $+\!\!+$:

$$\begin{array}{ll} & nil +\!\!+ x \\ = & \quad \{\ \text{definition 2.1.8}\ \} \\ & (\!|nil, \succ|\!).x \\ = & \quad \{\ \text{corollary 2.1.4, identity}\ \} \\ & x \end{array}$$

and to prove it is the right unit:

$$\begin{array}{ll} & x +\!\!+ nil \\ = & \quad \{\ \text{definition 2.1.8}\ \} \\ & (\!|x, \succ|\!).nil \\ = & \quad \{\ (2.3)\ \} \\ & x \end{array}$$

And finally associativity, where we make use of the observation that all catamorphisms are promotable; in particular, that $(\!|x, \succ|\!)$ is $\succ$ to $\succ$ promotable.

$$
\begin{array}{ll}
 & x +\!\!+ (y +\!\!+ z) \\
= & \quad \{ \text{ definition 2.1.8 } \} \\
 & (([\![ x, \succ ]\!] \circ [\![ y, \succ ]\!]).z \\
= & \quad \{ \text{ (2.4); promotion } \} \\
 & ([\![ [\![ x, \succ ]\!].y \, , \succ ]\!].z \\
= & \quad \{ \text{ definition 2.1.8 } \} \\
 & (x +\!\!+ y) +\!\!+ z
\end{array}
$$

Of course, the proof by induction of the associativity of concatenation is likewise trivial, but the point is that the above proof proceeds extremely simply, and is entirely calculational, which makes it considerably shorter. Proof by induction generally entails a greater organisational overhead — decomposing the proof into a basis and an inductive step, for instance. Proof by promotion, on the other hand, collapses the two cases into one: the basis takes place, as it were, in the first component of the catamorphism, and the inductive step in the second component. If nothing else, the triviality of the above proof matches the triviality of the proposition it proves.

## 2.2 Examples of Hagino Types

In this section we give some examples of initial Hagino types and illustrate some of their basic properties. The types we consider are disjoint sum, the natural numbers, "join lists", and a pair of types defined by mutual induction: trees and forests. The importance of disjoint sum is that it can be used to construct the disjoint sum of a number of functors: given functors $\mathsf{F}_1, \ldots, \mathsf{F}_n$, their disjoint sum $\mathsf{F}_1 \hat{+} \cdots \hat{+} \mathsf{F}_n$, which takes $A$ to $A\mathsf{F}_1 + \cdots + A\mathsf{F}_n$, is also a functor. A consequence of this construction is that the number of type-functors in the definition of a Hagino type is irrelevant — a type defined as $\mu(\tau_1 : \mathsf{F}_1, \ldots, \tau_n : \mathsf{F}_n)$ is isomorphic to the type $\mu(\tau : \mathsf{F}_1 \hat{+} \cdots \hat{+} \mathsf{F}_n)$. This not only allows us to save space (and subscripts) by collapsing several functors into one "summed" functor, but has also some theoretical significance in that it is known that fixed points of polynomial functors exist in suitable categories (polynomial functors are those constructed recursively from a class of primitive functors by disjoint sum and cartesian product: see chapter 5).

The type of natural numbers is used to illustrate how catamorphisms which compute given functions can be derived from the unique extension property. Some of the



examples we give are taken from a series of exercises dealing with this process and devised by Ed Voermans and Roland Backhouse for a course on functional programming given by the latter at Groningen University earlier this year. Using the unique extension property to derive catamorphisms is a standard technique in the early papers on the Bird-Meertens formalism (see Meertens [34] and Bird [8]), though almost exclusively for snoc- or join lists.

Join lists are interesting because they provide an example of a type with equations: the "join" constructor is associative and has a unit element. Although they do not fall properly within the preserve of Hagino types, types with equations can be treated in the same framework, in essentially the same way as the "congruence types" of Backhouse et al. [6] are treated in Martin-Löf's type theory (see, for example, Manes and Arbib [30], chapter 14). Another interesting aspect of join lists is that all join list catamorphisms can be factored into the composition of two catamorphisms, a "reduction" and a "map". We use promotion to prove this factorisation, a subject we return to in section 2.3.2, where we investigate which other types enjoy such a factorisation property.

The types of trees and forests show that Hagino types can be extended quite naturally to accommodate mutual induction.

### 2.2.1 Disjoint Sum

The disjoint sum of two types $A$ and $B$ can be defined by:

$$A + B \triangleq \mu(inl : (A\ll), inr : (B\ll)).$$

That is, the type has two constructors, the standard injections $inl \in A + B \longleftarrow A$ and $inr \in A + B \longleftarrow B$. Catamorphisms are constructed as follows: if $f \in P \longleftarrow A$ and $g \in P \longleftarrow B$, then there is a unique catamorphism $([\![ f, g ]\!]) \in P \longleftarrow A + B$ satisfying

$$([\![ f, g ]\!]) \circ inl = f \tag{2.6}$$

$$([\![ f, g ]\!]) \circ inr = g. \tag{2.7}$$

Since the type is not recursively defined, its promotion theorem has a simple form with no antecedent:

$$h \circ ([\![ f, g ]\!]) = ([\![ h \circ f, h \circ g ]\!]). \tag{2.8}$$

It is straightforward to show that $+$ is a type functor. We define its action on functions as follows: for $f \in C \longleftarrow A$ and $g \in D \longleftarrow B$, define

$$f+g \triangleq ([\![ inl \circ f, inr \circ g ]\!]) \in C+D \longleftarrow A+B. \tag{2.9}$$

Now $+$ respects identity, since $\mathrm{I}+\mathrm{I}$ is, by the above definition, $(\!|inl, inr|\!)$, which by property 2.1.4 is the identity function on $A+B$. To show that $+$ respects composition, we calculate as follows:

$$
\begin{array}{ll}
 & f+g \circ h+i \\
= & \quad \{ \ (2.9) \ \} \\
 & f+g \circ (\!|inl \circ h,\ inr \circ i|\!) \\
= & \quad \{ \ (2.8) \ \} \\
 & (\!|f+g \circ inl \circ h,\ f+g \circ inr \circ i|\!) \\
= & \quad \{ \ (2.9), (2.6), (2.7) \ \} \\
 & (\!|inl \circ f \circ h,\ inr \circ g \circ i|\!) \\
= & \quad \{ \ (2.9) \ \} \\
 & (f \circ h) + (g \circ i)
\end{array}
$$

Generalising disjoint sum to an arbitrary number of types as arguments allows us to make the following notational saving. A type $\mathcal{T} = \mu(\tau_1 : \mathsf{F}_1, \ldots, \tau_n : \mathsf{F}_n)$ defined by more than one type-functor can be collapsed into the type $\mu(\tau : \mathsf{F})$, where $A\mathsf{F} \triangleq A\mathsf{F}_1 + \cdots + A\mathsf{F}_n$. That is, $\mathcal{T}$ is isomorphic to, or in one-to-one correspondence with, the type $\mu(\tau : \mathsf{F})$ — the reader may care to prove this isomorphism for himself: the bijections are constructed entirely from catamorphisms of the relevant types. Because of this isomorphism, we can, without loss of generality, restrict our attention to Hagino types defined by a single type functor (though when we discuss particular types we often find it convenient to use more than one type-functor). When a type is defined by a single type-functor, its constructor is bijective. Let $\mathcal{T} = \mu(\tau : \mathsf{F})$; the inverse of $\tau$ is $(\!|\tau\mathsf{F}|\!)$. We prove the bijection as follows:

$$
\begin{array}{ll}
 & \tau \circ (\!|\tau\mathsf{F}|\!) = \mathrm{I} \\
\equiv & \quad \{ \text{ property 2.1.4 } \} \\
 & \tau \circ (\!|\tau\mathsf{F}|\!) = (\!|\tau|\!) \\
\Leftarrow & \quad \{ \text{ promotion } \} \\
 & \tau \circ \tau\mathsf{F} = \tau \circ \tau\mathsf{F} \\
\equiv & \quad \{ \text{ identity } \} \\
 & \text{true}
\end{array}
$$

and



$$
\begin{array}{ll}
 & (\!|\tau\mathsf{F}|\!) \circ \tau \\
= & \quad \{ \ (2.1) \ \} \\
 & \tau\mathsf{F} \circ (\!|\tau\mathsf{F}|\!)\mathsf{F} \\
= & \quad \{ \text{ functors } \} \\
 & (\tau \circ (\!|\tau\mathsf{F}|\!))\mathsf{F} \\
= & \quad \{ \text{ above; functors preserve identity } \} \\
 & \mathrm{I}
\end{array}
$$

This bijection seems to have been first proven by Lambek in [27]; its significance is that the type $\mathcal{T}$ is a fixed point of the functor $\mathsf{F}$ in that $\mathcal{T} \cong \mathcal{T}\mathsf{F}$ (see chapter 5 below). While it is pleasant to observe that disjoint sum can be defined as a Hagino type and that its calculationally interesting properties arise, as those of other Hagino types, from its unique extension property, it is convenient in reasoning about the existence of Hagino types to treat disjoint sum as a special case and concentrate, as we do in chapter 5, on Hagino types as fixed points of a single functor.

### 2.2.2 Natural Numbers

The type of natural numbers can be defined by means of the constructors $0 \in \mathbb{N}$ and the successor operator $\sigma \in \mathbb{N} \longleftarrow \mathbb{N}$:

$$\mathbb{N} \triangleq \mu(0 : (\mathbb{1} \ll), \sigma : \imath).$$

The instantiation of the unique extension property for $\mathbb{N}$ is as follows. For all $e \in P$ and $f \in P \longleftarrow P$, there is a catamorphism $(\!|e, f|\!) \in P \longleftarrow \mathbb{N}$ satisfying: for all $h \in P \longleftarrow \mathbb{N}$,

$$h = (\!|e, f|\!) \quad \equiv \quad h.0 = e \ \wedge \ h \circ \sigma = f \circ h.$$

By taking $h := (\!|e, f|\!)$, the left-hand side of the above equivalence becomes identically true, which yields the following evaluation rules for $\mathbb{N}$-catamorphisms:

$$
\begin{array}{rcl}
(\!|e, f|\!).0 & = & e \\
(\!|e, f|\!) \circ \sigma & = & f \circ (\!|e, f|\!).
\end{array}
$$

As an example of how the unique extension property can be used to express a function as a catamorphism, consider the exponentiation function, $(x\hat{})$, which, for a given $x \in \mathbb{N}$, is the function that takes $n \in \mathbb{N}$ to $x^n$. We wish to express $(x\hat{})$ as a

catamorphism $(\!|e, f|\!)$, where $e$ and $f$ have yet to be found. From the unique extension property, we have

$$(x\hat{}) = (\!|e, f|\!) \quad \equiv \quad (x\hat{}).0 = e \ \wedge \ (x\hat{}) \circ \sigma \ = \ f \circ (x\hat{}).$$

Let us agree that $0^0 = 1$, then we can choose $e := 1$; it remains to find $f$. We note that $x^{n+1} = x \bullet x^n$ or, in other words, $(x\hat{}) \circ \sigma = (x\bullet) \circ (x\hat{})$, which leads to the choice $f := (x\bullet)$, and we have $(x\hat{}) = (\!|1, (x\bullet)|\!)$. If we wish further to express $(x\bullet)$ as a catamorphism $(\!|e', f'|\!)$, then we remark that $x \bullet 0 = 0$ (hence choose $e' := 0$) and $x \bullet (n+1) = x + x \bullet n$ (hence choose $f' := (x+)$) and we have found $(x\bullet) = (\!|0, (x+)|\!)$. The process can be repeated yet again to give $(x+) = (\!|x, \sigma|\!)$.

Catamorphisms follow the paradigm of primitive recursive functions $h$ such that $h.0 = e$ and $h.(n+1) = f.(h.n)$ for constant $e$ and function $f$ of one argument. What then of the other paradigm of primitive recursive functions $h$ such that $h.0 = e$ and $h.(n+1) = h.n \oplus n$? It is well known that the two paradigms are equivalent in the presence of tupling. For example, the factorial function $fac$ is characterised by $fac.0 = 1$ and

$$fac.(n+1) \ = \ fac.n \oplus n,$$

where $x \oplus y = x \bullet (y+1)$. Abstracting on the variable $n$ gives:

$$fac \circ \sigma \ = \ \oplus \circ \langle fac, \mathrm{I} \rangle.$$

Suppose we can express $\langle fac, \mathrm{I} \rangle$ as a catamorphism $(\!|e, f|\!)$, then it follows that $fac = \ll \circ (\!|e, f|\!)$. Let us find $e$ and $f$ by calculation. First of all, $\langle fac, \mathrm{I} \rangle.0 = (fac.0, 0) = (1, 0)$, so we choose $e := (1, 0)$. We must now find $f$ such that

$$\langle fac, \mathrm{I} \rangle \circ \sigma \ = \ f \circ \langle fac, \mathrm{I} \rangle;$$

with this in mind, we calculate as follows:

$$\begin{array}{ll}
 & \langle fac, \mathrm{I} \rangle \circ \sigma \\
= & \qquad \{ \text{ calculus } \} \\
 & \langle fac \circ \sigma, \ \sigma \rangle \\
= & \qquad \{ \text{ property of } fac \ \} \\
 & \langle \oplus \circ \langle fac, \mathrm{I} \rangle, \ \sigma \rangle \\
= & \qquad \{ \text{ calculus } \} \\
 & \langle \oplus \circ \langle fac, \mathrm{I} \rangle, \ \sigma \circ \gg \circ \langle fac, \mathrm{I} \rangle \rangle \\
= & \qquad \{ \text{ calculus } \} \\
 & \langle \oplus, \ \sigma \circ \gg \rangle \circ \langle fac, \mathrm{I} \rangle
\end{array}$$



So we have found $f := \langle \oplus, \sigma \circ \gg \rangle$, and thus $fac = \ll \circ (\!|(1,0), \langle \oplus, \sigma \circ \gg \rangle|\!)$. In general, if the primitive recursive function $h$ is such that

$$h.0 = e \quad \wedge \quad h.(n+1) = h.n \oplus n$$

then

$$h = \ll \circ (\!|(e,0), \langle \oplus, \sigma \circ \gg \rangle|\!).$$

Meertens, in [33], calls functions of such a form "paramorphisms". Paramorphisms enjoy many delightful algebraic properties, including a unique extension property and a sort of promotion theorem; we shall see more examples of paramorphisms later (in fact, Meertens shows that all functions on Hagino types can be expressed as paramorphisms), but for the present we go on to look at another example of a Hagino type.

### 2.2.3 Join Lists

One view of finite lists has them constructed from the following operators: the empty list $\Box \in A*$, the singleton constructor $\eta \in A* \longleftarrow A$, and the concatenation operator, "join", $+\!\!+ \in A* \longleftarrow A* \times A*$. From the types of these constructors we may make the definition:

$$A* \ \triangleq \ \mu(\Box : (\mathbb{1} \ll), \eta : (A \ll), +\!\!+ : \widehat{\ll}).$$

The type so defined, however, is actually a type of binary trees; to obtain the type of finite lists, one has to add that $\Box$ is the unit of $+\!\!+$, and that $+\!\!+$ is associative. In other words, for all $x, y, z \in A*$: $x +\!\!+ (y +\!\!+ z) = (x +\!\!+ y) +\!\!+ z$, and $\Box +\!\!+ x = x = x +\!\!+ \Box$. These equations constitute an essential part of the definition of join lists. In constructing recursive functions on types with equations, one must ensure that the functions respect the given equations, i.e., give the same result when applied to equal objects. In Martin-Löf's type theory, recursive functions are constructed by means of an elimination rule. That the functions respect any equations on the defined type is ensured by adding extra premises to the elimination rule (see Chisholm [12] and Backhouse et al. [6]). Similar considerations apply to Hagino types: we require that the underlying algebras inherit the same equations. In the case of join lists, this means that an $A*$-algebra is a quadruple $(P, 1_\oplus, f, \oplus)$, where $P$ is a type, $1_\oplus \in P \longleftarrow \mathbb{1}$, $f \in P \longleftarrow A$, and $\oplus \in P \longleftarrow P \times P$ is associative with unit $1_\oplus$ (just as $+\!\!+$ is associative with unit $\Box$). The initial algebra is $(A*, \Box, \eta, +\!\!+)$, so for all $A*$-algebras $(P, 1_\oplus, f, \oplus)$, there is a catamorphism $(\!|1_\oplus, f, \oplus|\!) \in P \longleftarrow A*$ with the uniqueness property that for all $h$,

$$h = (\!|1_\oplus, f, \oplus|\!) \quad \equiv \quad h.\Box = 1_\oplus \ \wedge \ h \circ \eta = f \ \wedge \ h \circ +\!\!+ \ = \ \oplus \circ h \times h.$$

Note that since $(P, 1_\oplus, f, \oplus)$ is an $A*$-algebra, $\oplus$ is associative with unit $1_\oplus$, so that the catamorphism $(\!|1_\oplus, f, \oplus|\!)$ clearly respects the equations on $A*$ and is thus well-defined as a function on join lists. By taking $h := (\!|1_\oplus, f, \oplus|\!)$ in the unique extension property above, we obtain the following evaluation rules for join list catamorphisms:

$$(\!|1_\oplus, f, \oplus|\!).\square = 1_\oplus \tag{2.10}$$

$$(\!|1_\oplus, f, \oplus|\!) \circ \eta = f \tag{2.11}$$

$$(\!|1_\oplus, f, \oplus|\!) \circ +\!\!+ = \oplus \circ (\!|1_\oplus, f, \oplus|\!) \times (\!|1_\oplus, f, \oplus|\!). \tag{2.12}$$

The promotion theorem for join lists is:

$$h \circ (\!|1_\oplus, f, \oplus|\!) = (\!|h.1_\oplus, h \circ f, \otimes|\!) \quad \Leftarrow \quad h \circ \oplus = \otimes \circ h \times h. \tag{2.13}$$

We shall use promotion to prove some elementary properties of join lists; we begin by defining two common forms of catamorphisms, maps and reductions. For $f \in A \longleftarrow B$, the function $f* \in A* \longleftarrow B*$ ("$f$ map") applies $f$ to each element of a given list.

**Definition 2.2.1 (map)** For $f \in A \longleftarrow B$, define

$$f* \triangleq (\!|\square, \eta \circ f, +\!\!+|\!) \in A* \longleftarrow B*.$$

□

The following equality, referred to as "map distribution," is an instance of a general property of maps which we prove in section 2.3:

$$(f \circ g)* = f* \circ g*.$$

The other form of join list catamorphism that we need is the reduction:

**Definition 2.2.2 (reduction)** For associative operator $\oplus \in A \longleftarrow A \times A$ with unit $1_\oplus$, define $\oplus/$ ("$\oplus$ reduce") by:

$$\oplus/ \triangleq (\!|1_\oplus, \mathrm{I}, \oplus|\!) \in A \longleftarrow A*.$$

□

An interesting result, first stated by Meertens [34], is that each join list catamorphism can be expressed as the composition of a reduction and a map:



**Property 2.2.3 (factorisation)** For all join list catamorphisms $(\!|1_\oplus, f, \oplus|\!)$,

$$(\!|1_\oplus, f, \oplus|\!) = \oplus/ \circ f*.$$

**Proof:** the calculation below uses the fact that reductions are promotable over concatenation: from definition 2.2.2 and (2.12), we have

$$\oplus/ \circ +\!\!+ = \oplus \circ \oplus/ \times \oplus/$$

and so promotion is applicable.

$$\begin{array}{ll} & \oplus/ \circ f* \\ = & \quad \{ \text{ definition 2.2.1 } \} \\ & \oplus/ \circ (\!|\square, \eta \circ f, +\!\!+|\!) \\ = & \quad \{ \text{ promotion } \} \\ & (\!|\oplus/.\square, \oplus/ \circ \eta \circ f, \oplus|\!) \\ = & \quad \{ \text{ definition 2.2.2; (2.10), (2.11) } \} \\ & (\!|1_\oplus, \mathrm{I} \circ f, \oplus|\!) \end{array}$$

□

Historically, the join list promotion theorem was first stated by Meertens in [34], where it is called "Law 2". It was later restated independently by Backhouse in [4], where it is called promotion. The term "promotion" was coined, as far as we are aware, by Bird, but is used by him to refer to the following two identities. Both identities involve the flatten function, $+\!\!+/$, so called because it flattens a list of lists into a list. We shall refer to the identities as the map promotion law:

$$f* \circ +\!\!+/ = +\!\!+/ \circ f** \tag{2.14}$$

and the reduce promotion law:

$$\oplus/ \circ +\!\!+/ = \oplus/ \circ \oplus/* \tag{2.15}$$

Since, by definitions 2.2.1 and 2.2.2, both maps and reductions are catamorphisms, these two laws are combined in the following property, which states that any catamorphism can be promoted over the flatten function.

**Property 2.2.4** For all associative binary operators $\oplus$ with a unit $1_\oplus$, and for all functions $f$ of appropriate type,

$$\oplus/ \circ f* \circ +\!\!+/ \;=\; \oplus/ \circ \oplus/* \circ f**.$$

**Proof:** the crucial step in the following proof is the application of the promotion theorem — $\oplus/ \circ f*$ is, by property 2.2.3, a catamorphism, and therefore $+\!\!+$ to $\oplus$ promotable.

$$\begin{array}{ll} & \oplus/ \circ f* \circ +\!\!+/ \\ = & \quad \{ \text{ definition 2.2.2 } \} \\ & \oplus/ \circ f* \circ (\!|\Box, \mathrm{I}, +\!\!+|\!) \\ = & \quad \{ \text{ promotability of catamorphisms } \} \\ & (\!|(\oplus/ \circ f*).\Box,\ \oplus/ \circ f*,\ \oplus|\!) \\ = & \quad \{ (2.10) \} \\ & (\!|1_\oplus,\ +\!\!+/ \circ f*,\ \oplus|\!) \\ = & \quad \{ \text{ property 2.2.3; map distribution } \} \\ & \oplus/ \circ \oplus/* \circ f** \end{array}$$

□

## 2.2.4 Mutually Inductive Types: Trees and Forests

In this section we consider catamorphisms on two types defined by mutual induction: trees and forests. Forests are list-like structures of trees; trees consist of an internal label and a forest of subtrees, so the trees have an arbitrary branching factor. For a base type $A$, we denote the type of trees over $A$ by $A_\mathsf{T}$, and the type of forests over $A$ by $A_\mathsf{F}$. Trees are constructed by the node operator $\surd \in A_\mathsf{T} \longleftarrow A \times A_\mathsf{F}$; forests have as constructors: the empty forest $\Box \in A_\mathsf{F}$, and the "cons" operator $\prec \in A_\mathsf{F} \longleftarrow A_\mathsf{T} \times A_\mathsf{F}$.

In order to define catamorphisms on types defined by mutual induction, we have to extend the construction of the previous section; rather than treat the general case, we consider only the particular example of trees and forests. We begin by giving the functors and algebra which define the types.

We define the pair of types, trees and forests, simultaneously:

$$(A_\mathsf{T}, A_\mathsf{F}) \;\triangleq\; \mu(\surd : \mathsf{G}\,;\ \Box : \mathsf{F}_1,\ \prec : \mathsf{F}_2)$$



where the functors $\mathsf{G}$ and $\mathsf{F}_i$ are defined as follows: each functor takes a pair of arguments; the functor $\mathsf{G}$ pertains to trees, the functors $\mathsf{F}_i$ to forests:

$$\begin{array}{lcllcl} (X,Y)\mathsf{G} & = & A \times Y & (f,g)\mathsf{G} & = & \mathrm{I} \times g \\ (X,Y)\mathsf{F}_1 & = & \mathbb{1} & (f,g)\mathsf{F}_1 & = & \mathrm{I} \\ (X,Y)\mathsf{F}_2 & = & X \times Y & (f,g)\mathsf{F}_2 & = & f \times g. \end{array}$$

This agrees with the types of the constructors, since we have:

$$\begin{array}{rcl} \surd & \in & A_\mathsf{T} \longleftarrow (A_\mathsf{T}, A_\mathsf{F})\mathsf{G} \\ \Box & \in & A_\mathsf{F} \longleftarrow (A_\mathsf{T}, A_\mathsf{F})\mathsf{F}_1 \\ \prec & \in & A_\mathsf{F} \longleftarrow (A_\mathsf{T}, A_\mathsf{F})\mathsf{F}_2. \end{array}$$

The algebra induced by this structure is given by a simple extension to the construction of section 2.1:

**Definition 2.2.5 (T-F-algebras)** A T-F-*algebra* is a quintuple $(P, Q, \oplus, e, \otimes)$ where $P$ and $Q$ are types, and

$$\begin{array}{lll} \oplus \in P \longleftarrow (P,Q)\mathsf{G} & \text{(i.e., } & P \longleftarrow A \times Q \text{ )} \\ e \in Q \longleftarrow (P,Q)\mathsf{F}_1 & \text{(i.e., } & Q \longleftarrow \mathbb{1} \text{ )} \\ \otimes \in Q \longleftarrow (P,Q)\mathsf{F}_2 & \text{(i.e., } & Q \longleftarrow P \times Q \text{ ).} \end{array}$$

A T-F-*homomorphism* from algebra $(P, Q, \oplus, e, \otimes)$ to algebra $(P', Q', \oplus', e', \otimes')$ is a *pair* of functions $(f, g)$ where $f \in P' \longleftarrow P$ and $g \in Q' \longleftarrow Q$ such that:

$$\begin{array}{rcll} f \circ \oplus & = & \oplus' \circ (f,g)\mathsf{G} & \{= \oplus' \circ \mathrm{I} \times g\} \\ g \circ e & = & e' \circ (f,g)\mathsf{F}_1 & \{g.e = e'\} \\ g \circ \otimes & = & \otimes' \circ (f,g)\mathsf{F}_2 & \{= \otimes' \circ f \times g\}. \end{array}$$

□

As for the data types in the previous sections, we define $(A_\mathsf{T}, A_\mathsf{F}, \surd, \Box, \prec)$ to be the initial T-F-algebra. This means that for every T-F-algebra $(P, Q, \oplus, e, \otimes)$ there is a unique catamorphism (which we write as $(\!|\oplus\,; e, \otimes|\!)$ rather than as a pair of functions) which satisfies:

$$\begin{array}{rcl} (\!|\oplus\,; e, \otimes|\!) \circ \surd & = & \oplus \circ \mathrm{I} \times (\!|\oplus\,; e, \otimes|\!) \\ (\!|\oplus\,; e, \otimes|\!).\Box & = & e \\ (\!|\oplus\,; e, \otimes|\!) \circ \prec & = & \otimes \circ (\!|\oplus\,; e, \otimes|\!) \times (\!|\oplus\,; e, \otimes|\!). \end{array}$$

Note that since we write these catamorphisms as one function rather than as a pair of functions, we must consider $(\!|\oplus\,;e,\otimes|\!)$ as having two types: $P\!\longleftarrow\!A_T$ and $Q\!\longleftarrow\!A_F$.

The promotion theorem for trees and forests is again a consequence of the initiality of trees and forests. It can be expressed as: if $(f,g)$ is a T-F-catamorphism from $(P,Q,\oplus,e,\otimes)$ to $(P',Q',\oplus',e',\otimes')$, then:

$$f \circ (\!|\oplus\,;e,\otimes|\!) = (\!|\oplus'\,;e',\otimes'|\!) \in P'\!\longleftarrow\!A_T$$

and

$$g \circ (\!|\oplus\,;e,\otimes|\!) = (\!|\oplus'\,;e',\otimes'|\!) \in Q'\!\longleftarrow\!A_F.$$

Or, equivalently,

**Theorem 2.2.6 (promotion)** If

$$f \circ \oplus = \oplus' \circ \mathrm{I}\times g \qquad \text{and} \qquad g \circ \otimes = \otimes' \circ f\times g,$$

then

$$f \circ (\!|\oplus\,;e,\otimes|\!) = (\!|\oplus'\,;g.e,\otimes'|\!) \in P'\!\longleftarrow\!A_T$$

and

$$g \circ (\!|\oplus\,;e,\otimes|\!) = (\!|\oplus'\,;g.e,\otimes'|\!) \in Q'\!\longleftarrow\!A_F.$$

□

Finally, as was the case with the data types of the previous sections, the identity catamorphism is obtained from the constructors of the types: $(\!|\surd\,;\Box,+\!\!\!<|\!)$ (c.f. property 2.1.4).

## 2.3 Parameterised Data Types

Many of the data structures used in computing science are parameterised by other types: thus we speak of lists *of* natural numbers, lists *of* booleans, trees *of* integers, trees *of* lists *of* integers ... and so on ad infinitum. Such structures are typically labelled with elements of the parameter type, at the tips or nodes of trees, for example, and they have this in common: that there is an "apply-to-all" operation which, given a function and a structure as arguments, applies the function to each labelling element of the parameter type, preserving the shape of the structure. An example is the map operation on join lists, defined in the previous section, which applies a given function



to each element of a list. By analogy, we shall refer to all "apply-to all" operations as "maps". In section 2.3.1 we define a map operation for an arbitrary parameterised type, and show that maps preserve identity and composition, with the result that each parameterised data type induces a type-functor. In section 2.3.2, we address the question: for which data structures is it always possible to factor catamorphisms into the composition of a reduction and a map.

### 2.3.1 Maps and Functors

In order to discuss parameterised data types in general, we take a binary type functor and fix one of its arguments, thus producing a unary type-functor. So, let $\otimes$ be a binary type functor, and define the type constructor $\varpi$ by: for all $A$,

$$A\varpi \triangleq \mu(\tau : (A\otimes)).$$

Note that, as discussed in section 2.2.1, no generality is lost by considering a Hagino type of one functor. Note also that $A$ may be a vector of types; an example is given below in which this is the case. For the sake of simplicity, however, we shall treat $A$ as though it were a single type. As a final remark on the above definition, we should, strictly speaking, insist that each type $A\varpi$, $B\varpi$, etc., be treated as a distinct type, with distinct constructors $\tau_A \in A\varpi\!\longleftarrow\!A\otimes A\varpi$, $\tau_B \in B\varpi\!\longleftarrow\!B\otimes B\varpi$, and so on. But subscripts are made to be dropped, and we simply write $\tau \in A\varpi\!\longleftarrow\!A\otimes A\varpi$, thus letting $\tau$ be polymorphic in $A$.

The map operation for $\varpi$ is defined by:

**Definition 2.3.1 (map)** For $f \in B\!\longleftarrow\!A$, define the map function $f\varpi$ by

$$f\varpi \triangleq (\!|\tau \circ f\otimes \mathrm{I}|\!) \in B\varpi\!\longleftarrow\!A\varpi.$$

□

The definition is well typed: we have $\tau \in B\varpi\!\longleftarrow\!B\otimes B\varpi$ and, since $\otimes$ is a functor, $f\otimes\mathrm{I} \in B\otimes B\varpi\!\longleftarrow\!A\otimes B\varpi$, hence

$$\tau \circ f\otimes\mathrm{I} \in B\varpi\!\longleftarrow\!A\otimes B\varpi$$

so by definition 2.1.3, $f\varpi \in B\varpi\!\longleftarrow\!A\varpi$.

We can see how maps are evaluated by composing with the constructor $\tau$:

$$
\begin{array}{ll}
 & f\varpi \circ \tau \\
= & \quad \{ \text{ definition 2.3.1 } \} \\
 & (\!|\tau \circ f\otimes \mathrm{I}|\!) \circ \tau \\
= & \quad \{ (2.1) \} \\
 & \tau \circ f\otimes \mathrm{I} \circ \mathrm{I}\otimes(\!|\tau \circ f\otimes \mathrm{I}|\!) \\
= & \quad \{ \otimes \text{ is a functor; definition 2.3.1 } \} \\
 & \tau \circ f\otimes f\varpi
\end{array}
$$

That is,

$$f\varpi \circ \tau = \tau \circ f\otimes f\varpi \tag{2.16}$$

which has the form of a natural transformation property, provided that $\varpi$ is a type functor. We show that this is indeed so, by proving that $\varpi$ respects identity and composition. First identity:

**Property 2.3.2** $\mathrm{I}\varpi = \mathrm{I} \in A\varpi \longleftarrow A\varpi$.

Proof:

$$
\begin{array}{ll}
 & \mathrm{I}\varpi \\
= & \quad \{ \text{ definition 2.3.1 } \} \\
 & (\!|\tau \circ \mathrm{I}\otimes \mathrm{I}|\!) \\
= & \quad \{ \otimes \text{ is a functor } \} \\
 & (\!|\tau \circ \mathrm{I}|\!) \\
= & \quad \{ \text{ identity, property 2.1.4 } \} \\
 & \mathrm{I}
\end{array}
$$

□

The following lemma allows us to give an elegant proof that $\varpi$ respects composition; it is important in its own right as the statement that a map can always be brought inside a catamorphism.

**Lemma 2.3.3** For $\phi \in C \longleftarrow B\otimes C$ and $f \in B \longleftarrow A$,

$$(\!|\phi|\!) \circ f\varpi = (\!|\phi \circ f\otimes \mathrm{I}|\!).$$

Proof:



$$
\begin{array}{ll}
 & (\!|\phi|\!) \circ f\varpi = (\!|\phi \circ f\otimes \mathrm{I}|\!) \\
\equiv & \quad \{ \text{ definition 2.3.1 } \} \\
 & (\!|\phi|\!) \circ (\!|\tau \circ f\otimes \mathrm{I}|\!) = (\!|\phi \circ f\otimes \mathrm{I}|\!) \\
\Leftarrow & \quad \{ \text{ promotion } \} \\
 & (\!|\phi|\!) \circ \tau \circ f\otimes \mathrm{I} = \phi \circ f\otimes \mathrm{I} \circ \mathrm{I}\otimes(\!|\phi|\!) \\
\equiv & \quad \{ \text{ evaluation of catamorphisms: (2.1) } \} \\
 & \phi \circ \mathrm{I}\otimes(\!|\phi|\!) \circ f\otimes \mathrm{I} = \phi \circ f\otimes \mathrm{I} \circ \mathrm{I}\otimes(\!|\phi|\!) \\
\equiv & \quad \{ \otimes \text{ is a functor } \} \\
 & \text{true}
\end{array}
$$

□

**Property 2.3.4** $(f \circ g)\varpi = f\varpi \circ g\varpi$.

**Proof:**

$$
\begin{array}{ll}
 & f\varpi \circ g\varpi \\
= & \quad \{ \text{ definition 2.3.1 } \} \\
 & (\!|\tau \circ f\otimes \mathrm{I}|\!) \circ g\varpi \\
= & \quad \{ \text{ lemma 2.3.3 } \} \\
 & (\!|\tau \circ f\otimes \mathrm{I} \circ g\otimes \mathrm{I}|\!) \\
= & \quad \{ \text{ functors preserve composition } \} \\
 & (\!|\tau \circ (f \circ g)\otimes \mathrm{I}|\!) \\
= & \quad \{ \text{ definition 2.3.1 } \} \\
 & (f \circ g)\varpi
\end{array}
$$

□

We conclude that $\varpi$ is a type-functor. Equation (2.16) thus becomes indeed a naturality property:

**Property 2.3.5** $\tau : \varpi \stackrel{\cdot}{\longleftarrow} \mathrm{I}\hat{\otimes}\varpi$.

□

Not only constructors, but also catamorphisms enjoy a naturality property. Property 2.3.6 below states that the catamorphism operator $(\!|.|\!)$ preserves natural transformations, in that if $\phi : \mathsf{G} \stackrel{\cdot}{\longleftarrow} \mathsf{F}\hat{\otimes}\mathsf{G}$, then $(\!|\phi|\!) : \mathsf{G} \stackrel{\cdot}{\longleftarrow} \mathsf{F}\varpi$. Note how the statement

of the property mirrors the typing rule for catamorphisms: if $\phi \in B \longleftarrow A \otimes B$, then $(\!|\phi|\!) \in B \longleftarrow A^{\varpi}$. These two naturality properties, of constructors and of catamorphisms, are a consequence of the polymorphism of the operators involved, viz.

$$\begin{aligned} \tau &\in A^{\varpi} \longleftarrow A \otimes A^{\varpi} \\ (\!|.|\!) &\in (B \longleftarrow A^{\varpi}) \longleftarrow (B \longleftarrow A \otimes B) \end{aligned}$$

for all types $A$ and $B$. The principle that polymorphism and naturality are closely related was first remarked by Reynolds in [37] (see also Freyd et al. [19]), where the emphasis is on constructing models of the polymorphic $\lambda$-calculus. The principle has been revived recently by Wadler [41] and, independently, by de Bruin [13]. In both papers, it is shown that many properties of polymorphic functions can be derived from their types alone. For example, the promotion theorem follows from the polymorphic type of the operator $(\!|.|\!)$. For the present, we shall not enter any further into the topic of naturality, beyond remarking that its main principle can be verbalised as "polymorphic functions map related objects to related objects," and that this principle was the motivating force behind Backhouse's investigation of relational catamorphisms in [2]. This investigation provides a basis for a theory of "relational programming" similar to that of Hoare and He [24]. In chapter 4 we present some of Backhouse's results, and use this as a background for a theory of assertional guards. A forthcoming paper by Backhouse, de Bruin, Voermans, van der Woude and the present author will develop more fully a formalism for relational programming based on unique extension properties and naturality.

Let us return to the preservation of natural transformations:

**Property 2.3.6 (naturality)** If $\phi : \mathsf{G} \stackrel{\cdot}{\longleftarrow} \mathsf{F}\hat{\otimes}\mathsf{G}$, then $(\!|\phi|\!) : \mathsf{G} \stackrel{\cdot}{\longleftarrow} \mathsf{F}^{\varpi}$.

**Proof:**

$$\begin{array}{ll} & (\!|\varphi|\!) : \mathsf{G} \stackrel{\cdot}{\longleftarrow} \mathsf{F}^{\varpi} \\ \equiv & \quad \{ \text{ definition 1.1.9 } \} \\ & f\mathsf{G} \circ (\!|\varphi|\!) = (\!|\phi|\!) \circ f\mathsf{F}^{\varpi} \qquad \text{for all } f \\ \equiv & \quad \{ \text{ lemma 2.3.3 } \} \\ & f\mathsf{G} \circ (\!|\phi|\!) = (\!|\phi \circ f\mathsf{F}\otimes\mathrm{I}|\!) \qquad \text{for all } f \\ \Leftarrow & \quad \{ \text{ promotion } \} \\ & f\mathsf{G} \circ \phi = \phi \circ f\mathsf{F}\otimes\mathrm{I} \circ \mathrm{I}\otimes f\mathsf{G} \qquad \text{for all } f \\ \equiv & \quad \{ \otimes \text{ is a functor; definition 1.1.9 } \} \end{array}$$



$$\phi : \mathsf{G} \stackrel{\cdot}{\longleftarrow} \mathsf{F}\hat{\otimes}\mathsf{G}$$

□

These results can be instantiated to Hagino types of more than one functor according to the isomorphism of section 2.2.1; essentially, one merely introduces subscripts. Thus, for example, if $A^{\varpi} = \mu(\tau_1 : (A \otimes_1), \tau_2 : (A \otimes_2))$, where $\otimes_1$ and $\otimes_2$ are binary type-functors, then the definition of maps (definition 2.3.1) becomes

$$f^{\varpi} \triangleq (\!|\tau_1 \circ f\otimes_1\mathrm{I},\ \tau_2 \circ f\otimes_2\mathrm{I}|\!) \tag{2.17}$$

and property 2.3.3 becomes

$$(\!|\phi,\psi|\!) \circ f^{\varpi} = (\!|\phi \circ f\otimes_1\mathrm{I},\ \psi \circ f\otimes_2\mathrm{I}|\!). \tag{2.18}$$

In the following subsection, we return to the factorisation of catamorphisms exemplified by the type of join lists. We present a sufficient condition for the catamorphisms of a given data type to be factorable, and show that the types which satisfy this condition have a rich algebraic structure which includes analogues of map- and reduce promotion.

## 2.3.2 Factoring Catamorphisms

We shall consider a parameterised Hagino type $A^{\varpi}$. Our factorability condition requires that $A^{\varpi}$ be expressed as a Hagino type of two functors:

**Definition 2.3.7 (free)** A type $A^{\varpi}$ is *free* if it can be expressed in the form:

$$A^{\varpi} = \mu(\eta : (A\ll), \theta : \mathsf{F}),$$

where $\mathsf{F}$ does not depend upon $A$.

□

The definition states that a free type $A^{\varpi}$ has one constructor $\eta \in A^{\varpi} \longleftarrow A$, and another $\theta \in A^{\varpi} \longleftarrow A^{\varpi}\mathsf{F}$ which does not depend upon $A$. For example, the type of join lists, $A*$, is equal (up to isomorphism) to the type $\mu(\eta : (A\ll), \theta : \mathsf{F})$, where $X\mathsf{F} = \mathbb{1} + (X \times X)$; in this case, the constructor $\theta$ represents the "sum" of the two constructors $\Box : (\mathbb{1}\ll)$ and $+\!\!+ : \mathrm{I}\hat{\times}\mathrm{I}$.

We shall show that catamorphisms on free types can be factored into a reduction and a map. We assume, then, for the remainder of this section, that the type $A^{\varpi} =$

$\mu(\eta:(A\ll), \theta:\mathsf{F})$ is free. In order to be able to discuss maps on the type, we must first express the type in such a way as agrees with the definition of parameterised types in the previous section. The functor $\mathsf{F}$ is obviously equal to the functor $(A\otimes)$, where for all $X$ and $Y$, we define $X \otimes Y = Y\mathsf{F}$. Thus

$$A\varpi = \mu(\eta:(A\ll), \theta:(A\otimes)), \tag{2.19}$$

which agrees with the definition of parameterised types in the previous section, so that we can appeal to the properties of maps proven there. A property of the defined functor $\otimes$ is that for all $X$, $Y$ and $Z$,

$$X \otimes Z = Y \otimes Z. \tag{2.20}$$

The unique extension property states that for $f \in P\longleftarrow A$ and $g \in P\longleftarrow P\mathsf{F}$, there is a unique catamorphism $(\!|f,g|\!) \in P\longleftarrow A\varpi$ satisfying:

$$(\!|f,g|\!) \circ \eta = f \tag{2.21}$$
$$(\!|f,g|\!) \circ \theta = g \circ (\!|f,g|\!)\mathsf{F} \tag{2.22}$$

and the promotion theorem for $A\varpi$ is:

$$h \circ (\!|f,g|\!) = (\!|h\circ f, i|\!) \Leftarrow h \circ g = i \circ h\mathsf{F}. \tag{2.23}$$

We begin by looking at maps and reductions on the type $A\varpi$.

**Property 2.3.8** $f\varpi = (\!|\eta\circ f, \theta|\!)$.

**Proof:** From definition 2.3.1 (see (2.17) above) we have $f\varpi = (\!|\eta\circ f\ll \mathrm{I}, \theta\circ f\otimes\mathrm{I}|\!)$. But $f\ll\mathrm{I} = f$, and by (2.20), $f\otimes\mathrm{I} = \mathrm{I}\otimes\mathrm{I} = \mathrm{I}$; thus $f\varpi = (\!|\eta\circ f, \theta|\!)$.
□

From this property and equations (2.21) and (2.22) we obtain the evaluation rules for maps:

$$f\varpi \circ \eta = \eta \circ f \tag{2.24}$$
$$f\varpi \circ \theta = \theta \circ f\varpi\mathsf{F} \tag{2.25}$$

Reductions are defined analogously to join list reductions. (Verwer in [40] takes a different approach to factoring catamorphisms, giving essentially a special case of lemma 2.3.3 above, while his definition of reduction is more general than that given here, his definition being applicable to any paramaterised Hagino type, not just free Hagino types.)



**Definition 2.3.9 (reductions)** For $g \in A\longleftarrow A\mathsf{F}$, define the reduction $g/$ by

$$g/ \triangleq (\!|\mathrm{I}, g|\!) \in A\longleftarrow A\varpi.$$

□

Immediate from this definition is:

$$g/ \circ \eta = \mathrm{I} \tag{2.26}$$
$$g/ \circ \theta = g \circ g/\mathsf{F} \tag{2.27}$$

Now, all $\varpi$-catamorphisms may be factored into the composition of a reduction and a map. The factoring is achieved as follows.

**Theorem 2.3.10 (factorisation)** For all $\varpi$-catamorphisms $(\!|f,g|\!)$,

$$(\!|f,g|\!) = g/ \circ f\varpi.$$

**Proof:**

$$\begin{aligned}
& g/ \circ f\varpi \\
= & \quad \{ \text{ definition 2.3.9 } \} \\
& (\!|\mathrm{I}, g|\!) \circ f\varpi \\
= & \quad \{ \text{ property 2.3.3 — see (2.18) } \} \\
& (\!|\mathrm{I}\circ f\ll\mathrm{I},\ g\circ f\otimes\mathrm{I}|\!) \\
= & \quad \{ \triangleq\ll, (2.20) \} \\
& (\!|f,\ g\circ\mathrm{I}\otimes\mathrm{I}|\!) \\
= & \quad \{ \text{ functors respect identity } \} \\
& (\!|f,g|\!)
\end{aligned}$$

□

This factorisation allows us to simplify the promotion theorem for free types. We can ignore the map component of a catamorphism:

$$h \circ g/ = i/ \circ h\varpi \Leftarrow h \circ g = i \circ h\mathsf{F} \tag{2.28}$$

We turn now to properties peculiar to free types. Our goal is to derive laws analogous to the map and reduce promotion laws for join lists. In fact, we shall prove a more general result, namely that free types induce a special sort of algebraic structure, a monad:

**Definition 2.3.11 (monad)** A *monad* is a triple, $(\mathsf{T}, \eta, \zeta)$, where $\mathsf{T}$ is a type-functor, $\eta$ and $\zeta$ are natural transformations, $\eta : \mathsf{T} \xleftarrow{\cdot} \mathsf{I}$ and $\zeta : \mathsf{T} \xleftarrow{\cdot} \mathsf{TT}$, such that:

$$\begin{aligned} \zeta \circ \eta\mathsf{T} &= \mathrm{I} \\ \zeta \circ \eta &= \mathrm{I} \\ \zeta \circ \zeta &= \zeta \circ \zeta\mathsf{T}. \end{aligned}$$

□

We shall now construct a monad from the type constructor $\varpi$. We know that $\varpi$ is a type functor, and from (2.24) we have $\eta : \varpi \xleftarrow{\cdot} \mathrm{I}$, so it only remains to find $\zeta : \varpi \xleftarrow{\cdot} \varpi\varpi$ satisfying the equations in the definition of monads. Type considerations alone suggest choosing $\zeta := \theta/$. (We note that $\theta/ \in A\varpi \longleftarrow A\varpi\varpi$ is a "flattening" function, analogous to the list function $+\!\!+/ \in A* \longleftarrow A**$.) We must now show that $\theta/$ is natural:

**Property 2.3.12** $\theta/ : \varpi \xleftarrow{\cdot} \varpi\varpi$.

**Proof:** Equation (2.25) states that $f\varpi \circ \theta = \theta \circ f\varpi\mathsf{F}$, whence promotion (2.28) gives

$$f\varpi \circ \theta/ = \theta/ \circ f\varpi\varpi$$

as desired.
□

All that remains is to show that $\eta$ and $\theta/$ satisfy the equations of a monad.

**Theorem 2.3.13** $(\varpi, \eta, \theta/)$ is a monad.
**Proof:** by properties 2.3.14, 2.3.15 and corollary 2.3.17 below.
□

**Property 2.3.14** $\theta/ \circ \eta\varpi = \mathrm{I}$.
**Proof:** Immediate from theorem 2.3.10 and the fact that $(\!|\eta, \theta|\!)$ is the identity catamorphism.
□

**Property 2.3.15** $\theta/ \circ \eta = \mathrm{I}$.
**Proof:** Immediate from (2.26).
□



**Property 2.3.16** For all $g$, $g/ \circ \theta/ = g/ \circ g/\varpi$.
**Proof:** Apply promotion (2.28) to the evaluation of reductions law (2.27).
□

**Corollary 2.3.17** $\theta/ \circ \theta/ = \theta/ \circ \theta/\varpi$.
□

Note that, since $\theta/$ corresponds to the flatten function, $+\!\!+/$, of join lists, properties 2.3.12 and property 2.3.16 correspond to the map and reduce promotion laws for join lists, equations (2.14) and (2.15) of section 2.2.3. Thus we have shown that these identities hold for all free data types.

### 2.3.3 Example

An example of a free data type is the type of rose trees, presented by Meertens in [32]. Rose trees are tree structures with an arbitrary branching factor: a tree is either a tip, or consists of a list of subtrees. Thus, denoting the type of rose trees over a base type $A$ by $A\varrho$, we note that the type is defined by the two constructors:

$$\begin{aligned} \rho &\in A\varrho \longleftarrow A \\ \pi &\in A\varrho \longleftarrow A\varrho*. \end{aligned}$$

From the types of which, we see that $A\varrho = \mu(\rho : (A \ll), \pi : *)$. Now the functor $*$ does not depend on $A$, so we conclude that $A\varrho$ is free, and therefore all $\varrho$-catamorphisms are factorable. That is, for all catamorphisms $(\!|f, g|\!)$, we have $(\!|f, g|\!) = g\not/ \circ f\varrho$, where $g\not/ = (\!|1, g|\!)$ and $f\varrho = (\!|\rho \circ f, \pi|\!)$. (We use $\not/$ as the reduction operator to avoid confusion with join list reductions in the following section.) Moreover, we have the analogues of the map and reduce promotion laws:

$$\begin{aligned} f\varrho \circ \pi\not/ &= \pi\not/ \circ f\varrho\varrho \\ g\not/ \circ \pi\not/ &= g\not/ \circ g\not/\varrho. \end{aligned}$$

## 2.4 Zygomorphisms

In our discussion of catamorphisms on the natural numbers in section 2.2.2, we referred briefly to Meertens' paper on paramorphisms, [33]. Paramorphisms are of interest for

two reasons. The first is that all functions on the natural numbers can be expressed as paramorphisms; Meertens, in fact, defined a general notion of paramorphism applicable to all initial Hagino types and showed that any function on an initial Hagino type could be expressed as a paramorphism. The second reason why paramorphisms are so interesting is that, as Meertens demonstrates, they enjoy very many useful algebraic properties. In short, paramorphisms constitute an attractive blend of strength and manipulability. In this section we present a generalisation of paramorphisms, which we call "zygomorphisms", these being a yoking together of a paramorphism and a catamorphism. Zygomorphisms also enjoy many algebraic properties, including a unique extension property and a form of promotion. We shall give a general definition of zygomorphisms for all initial Hagino types, but we begin by motivating the definition with a discussion of zygomorphisms on the natural numbers.

Recall that a paramorphism of type $P \longleftarrow \mathbb{N}$ is a function of the form

$$\ll \circ (\!|(e,0), \langle \oplus, \sigma \circ \gg \rangle|\!)$$

for some $e \in P$ and $\oplus \in P \longleftarrow P \times \mathbb{N}$, and that for all functions $h$, if $h.0 = e$ and $h.(n+1) = h.n \oplus n$, then $h$ is equal to the above paramorphism.

Consider now the function $H$ defined by

$$H.n \triangleq \Sigma(i : 0 \leq i < n : x^i).$$

Straightforward calculation establishes that

$$H.0 = 0 \tag{2.29}$$

$$H.(n+1) = H.n + x^n \tag{2.30}$$

This is sufficient to give an implementation of $H$ as the paramorphism

$$\ll \circ (\!|(0,0), \langle \oplus, \sigma \circ \gg \rangle|\!),$$

where $h \oplus n \triangleq h + x^n$. However, we can go one stage further by noting that (2.30) is expressed more usefully by $H \circ \sigma = + \circ \langle H, (x^\wedge) \rangle$, and that $(x^\wedge) = (\!|1, (x\bullet)|\!)$. We use the unique extension property for $\mathbb{N}$ to find a catamorphism which computes $\langle H, (x^\wedge) \rangle$. For the basis, $\langle H, (x^\wedge) \rangle.0 = (0,1)$; now we must find an $f$ such that $\langle H, (x^\wedge) \rangle \circ \sigma = f \circ \langle H, (x^\wedge) \rangle$. We find such an $f$ by calculating:

$$\begin{array}{ll} & \langle H, (x^\wedge) \rangle \circ \sigma \\ = & \quad \{ \text{ calculus } \} \\ & \langle H \circ \sigma, (x^\wedge) \circ \sigma \rangle \end{array}$$



$$\begin{array}{ll} = & \quad \{ \ (2.30), (x^\wedge) = (\!|1, (x\bullet)|\!) \ \} \\ & \langle + \circ \langle H, (x^\wedge) \rangle, (x\bullet) \circ (x^\wedge) \rangle \\ = & \quad \{ \text{ calculus } \} \\ & \langle +, (x\bullet) \circ \gg \rangle \circ \langle H, (x^\wedge) \rangle \end{array}$$

So we have $\langle H, (x^\wedge) \rangle = (\!|(0,1), \langle +, (x\bullet) \circ \gg \rangle|\!)$, and therefore

$$H = \ll \circ (\!|(0,1), \langle +, (x\bullet) \circ \gg \rangle|\!).$$

The right-hand side is not quite in the form of a paramorphism, but is obtained from the definition of paramorphism by abstracting on 0 and $\sigma$. This leads us to define zygomorphisms on the natural numbers as functions of the form

$$\ll \circ (\!|(e,d), \langle \oplus, g \circ \gg \rangle|\!).$$

When $d = 0$ and $g = \sigma$, the above zygomorphism is a paramorphism, so zygomorphisms are indeed more general than paramorphisms.

We encapsulate the reasoning that led to the above implementation of $H$ in the following property:

**Property 2.4.1** For all $h$,

$$h = \ll \circ (\!|(e,d), \langle \oplus, g \circ \gg \rangle|\!) \quad \equiv \quad h.0 = e \ \wedge \ h \circ \sigma = \oplus \circ \langle h, (\!|d, g|\!) \rangle.$$

**Proof:** instance of property 2.4.6 below.
□

**Example 2.4.2** Consider the function $sq \in \mathbb{N} \longleftarrow \mathbb{N}$ such that $sq.n = n^2$. We have that $sq.0 = 0$ and

$$sq \circ \sigma = + \circ \langle sq, \sigma \circ (2\bullet) \rangle.$$

It is easily shown that $\sigma \circ (2\bullet) = (\!|1, \sigma \circ \sigma|\!)$, and therefore

$$sq \circ \sigma = + \circ \langle sq, (\!|1, \sigma \circ \sigma|\!) \rangle.$$

Property 2.4.1 gives

$$sq = \ll \circ (\!|(0,1), \langle +, \sigma \circ \sigma \circ \gg \rangle|\!).$$

□

We have borrowed the function $H$ above, and property 2.4.1 from some lecture notes by Backhouse, who in turn borrowed the example of $H$ from the thesis of Hoogerwoord [25]. Hoogerwoord uses $H$ to exemplify the derivation of algorithms in a calculational style from recurrence relations such as (2.30), and Backhouse's lecture notes show that such derivations can also be performed very effectively by appealing to unique extension properties, as in the example of $H$ which we have just seen. In Backhouse's lecture notes, property 2.4.1 is stated as an implication ($\Leftarrow$); the equivalence in our statement is more interesting since it states a uniqueness property for zygomorphisms, and uniqueness properties often lead to promotion-like theorems. We proceed now to give a general definition of zygomorphisms, which will lead to the statement of a zygomorphism promotion theorem.

## 2.4.1 Properties of Zygomorphisms

We consider an arbitrary initial Hagino type $\mathcal{T} = \mu(\tau : \mathsf{F})$. Zygomorphisms on $\mathcal{T}$ are defined by:

**Definition 2.4.3** For $f \in P \longleftarrow (P \times Q)\mathsf{F}$ and $g \in Q \longleftarrow Q\mathsf{F}$, define the zygomorphism $(f,g)^\sharp \in P \longleftarrow \mathcal{T}$ by

$$(f,g)^\sharp \triangleq \ll \circ (\!|\langle f, g \circ \gg\mathsf{F}\rangle|\!).$$

□

We note in passing that Meertens' paramorphisms are zygomorphisms of the form $(f,\tau)^\sharp$ for some $f \in P \longleftarrow (P \times \mathcal{T})\mathsf{F}$. The term "zygomorphism" has been chosen to suggest the conjunction of a paramorphism and a catamorphism. The catamorphism part is seen in the following property:

**Property 2.4.4** $\gg \circ (\!|\langle f, g \circ \gg\mathsf{F}\rangle|\!) = (\!|g|\!)$.

Proof: immediate from $\mathcal{T}$-promotion and $\gg \circ \langle f, g \circ \gg\mathsf{F}\rangle = g \circ \gg\mathsf{F}$.
□

This yields the conjunction:

**Corollary 2.4.5** $(\!|\langle f, g \circ \gg\mathsf{F}\rangle|\!) = \langle (f,g)^\sharp, (\!|g|\!)\rangle$.
□

We can now prove the uniqueness of zygomorphisms property, of which property 2.4.1 above was an instance.



**Property 2.4.6 (syzygy law)** For all $h \in P \longleftarrow \mathcal{T}$,

$$h = (f,g)^\sharp \quad \equiv \quad h \circ \tau = f \circ \langle h, (\!|g|\!)\rangle\mathsf{F}.$$

**Proof:**

$$\begin{array}{ll} & h = (f,g)^\sharp \\ \equiv & \quad \{ \text{ definition 2.4.3, property 2.4.4, predicate calculus } \} \\ & h = \ll \circ (\!|\langle f, g \circ \gg\mathsf{F}\rangle|\!) \ \wedge \ (\!|g|\!) = \gg \circ (\!|\langle f, g \circ \gg\mathsf{F}\rangle|\!) \\ \equiv & \quad \{ \text{ uep for cartesian product } \} \\ & \langle h, (\!|g|\!)\rangle = (\!|\langle f, g \circ \gg\mathsf{F}\rangle|\!) \\ \equiv & \quad \{ \text{ uep for catamorphisms } \} \\ & \langle h, (\!|g|\!)\rangle \circ \tau = \langle f, g \circ \gg\mathsf{F}\rangle \circ \langle h, (\!|g|\!)\rangle\mathsf{F} \\ \equiv & \quad \{ \text{ calculus } \} \\ & \langle h \circ \tau, (\!|g|\!) \circ \tau\rangle = \langle f \circ \langle h, (\!|g|\!)\rangle\mathsf{F}, g \circ (\!|g|\!)\mathsf{F}\rangle \\ \equiv & \quad \{ \text{ catamorphism evaluation: (2.1); calculus } \} \\ & h \circ \tau = f \circ \langle h, (\!|g|\!)\rangle\mathsf{F} \end{array}$$

□

By taking $h := (f,g)^\sharp$, this in turn gives the evaluation of zygomorphisms law:

**Corollary 2.4.7** $(f,g)^\sharp \circ \tau = f \circ \langle (f,g)^\sharp, (\!|g|\!)\rangle\mathsf{F}$.
□

The existence of a uniqueness property suggests that we may find a promotion-like property. The following seems a combination of Meertens' paramorphism promotion and our catamorphism promotion.

**Property 2.4.8 (syzygy promotion)**

$$h \circ (i,j)^\sharp = (f,g)^\sharp \quad \Leftarrow \quad h \circ i = f \circ (h \times k)\mathsf{F} \ \wedge \ k \circ j = g \circ k\mathsf{F}$$

**Proof:**

$$\begin{array}{ll} & h \circ (i,j)^\sharp = (f,g)^\sharp \\ \equiv & \quad \{ \text{ syzygy law } \} \\ & h \circ (i,j)^\sharp \circ \tau = f \circ \langle h \circ (i,j)^\sharp, (\!|g|\!)\rangle\mathsf{F} \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ corollary 2.4.7 } \} \\
& h \circ i \circ \langle (i,j)^{\natural}, (\!|j|\!)\rangle_{\mathsf{F}} \;=\; f \circ \langle h \circ (i,j)^{\natural}, (\!|g|\!)\rangle_{\mathsf{F}} \\
\Leftarrow & \{ \text{ calculus } \} \\
& h \circ i \;=\; f \circ (h \times k)_{\mathsf{F}} \;\wedge\; k \circ (\!|j|\!) \;=\; (\!|g|\!) \\
\Leftarrow & \{ \mathcal{T}\text{-promotion } \} \\
& h \circ i \;=\; f \circ (h \times k)_{\mathsf{F}} \;\wedge\; k \circ j \;=\; g \circ k_{\mathsf{F}}
\end{array}
$$

□

This promotion theorem, with its two antecedents and six variables, seems rather unwieldy; to show that it is not necessarily so, we illustrate its use with a small problem on rose trees: to compute the minimal sum of proper subtrees. We shall transform a quadratic time algorithm into a linear time one. We begin by instantiating the above properties for the case of rose trees. Let us define the operator & by:

$$f \& g \triangleq \langle f, g \circ \gg * \rangle.$$

Then a rose tree zygomorphism is a function of the form

$$\ll \circ (f \& g)\!\!\downarrow \circ \langle r, s \rangle \varrho$$

for functions $f$, $g$, $r$ and $s$ of the appropriate types. The syzygy law gives the following uniqueness property: for all $h$,

$$
\begin{array}{l}
h \;=\; \ll \circ (f \& g)\!\!\downarrow \circ \langle r, s \rangle \varrho \\
\equiv \\
h \circ \rho = r \;\wedge\; h \circ \pi \;=\; f \circ \langle h, g\!\!\downarrow \circ s \varrho \rangle *
\end{array}
$$

Just as the freeness of the type of rose trees allowed us to simplify the type's promotion theorem by considering only reductions (see equation (2.28) of section 2.3.2), so also the syzygy promotion theorem may be simplified. We leave the reader to check the details. Syzygy promotion for rose trees becomes:

$$
\begin{array}{l}
h \circ \ll \circ (i \& j)\!\!\downarrow \;=\; \ll \circ (f \& g)\!\!\downarrow \circ (h \times k) \varrho \\
\Leftarrow \\
h \circ i \;=\; f \circ (h \times k) * \;\wedge\; k \circ j \;=\; g \circ k *
\end{array}
$$



The minimal sum of subtrees is computed by the following function

$$\Downarrow / \circ \mathrm{sum} * \circ ps$$

if $ps \in \mathbb{Z}\varrho * \longleftarrow \mathbb{Z}\varrho$ computes the proper subtrees of a given rose tree, and $sum \in \mathbb{Z} \longleftarrow \mathbb{Z}\varrho$ sums all the elements of a given rose tree. It is easily seen that $(+/)\!\!\downarrow$ implements $sum$; we must now find a function that computes $ps$. Now the proper subtrees of a tip $\rho.z$ is the empty list $\square$:

$$ps \circ \rho \;=\; (\square \ll) \tag{2.31}$$

where $(\square \ll)$ is the constant function always returning the empty list. Given a node $\pi.l$ where $l$ is a list of subtrees, the proper subtrees of $\pi.l$ comprise all the proper subtrees of all the trees in $l$, and all the trees in $l$ themselves. For each tree in $l$, then, we compute the proper subtrees of the tree, add the tree itself, and collate all the resulting subtrees. That is,

$$ps.(\pi.l) \;=\; +\!\!+/.((\succ \circ \langle ps, \mathrm{I} \rangle) *.l).$$

This gives us the first step in the following calculation; the remaining steps manipulate the right-hand side into a form for which the syzygy law is applicable.

$$
\begin{array}{ll}
& ps \circ \pi \\
= & \{ \text{ above } \} \\
& +\!\!+/ \circ (\succ \circ \langle ps, \mathrm{I} \rangle) * \\
= & \{ \text{ map distribution } \} \\
& +\!\!+/ \circ \succ * \circ \langle ps, \mathrm{I} \rangle * \\
= & \{ \mathrm{I} = \pi\!\!\downarrow \circ \rho \varrho \; \} \\
& +\!\!+/ \circ \succ * \circ \langle ps, \pi\!\!\downarrow \circ \rho \varrho \rangle *
\end{array}
$$

From this calculation and (2.31), the syzygy law gives

$$ps \;=\; \ll \circ ((+\!\!+/ \circ \succ *) \& \pi)\!\!\downarrow \circ \langle (\square \ll), \rho \rangle \varrho.$$

And so the minimal sum of subtrees is computed by:

$$\Downarrow / \circ (+/)\!\!\downarrow * \circ \ll \circ ((+\!\!+/ \circ \succ *) \& \pi)\!\!\downarrow \circ \langle (\square \ll), \rho \rangle \varrho \tag{2.32}$$

In this program, we have mixed notation for join- and snoc lists: this is largely a question of readability. Instead of $\succ$, we might write $+\!\!+ \circ 1 \times \eta$, which gives the derived evaluation rules:

$$(2.33) \qquad \oplus/ \circ f* \circ \succ \;=\; \oplus \circ (\oplus/ \circ f*) \times f$$

For all join list catamorphisms $\oplus/ \circ f*$.

In order to simplify (2.32) by syzygy promotion, we must find $f$, $g$ and $k$ such that

$$(2.34) \qquad \downarrow\!/ \circ (+/)\!\!\not* \circ +\!\!+/ \circ \succ* \;=\; f \circ ((\downarrow\!/ \circ (+/)\!\!\not*) \times k)*$$

and

$$(2.35) \qquad k \circ \pi \;=\; g \circ k*.$$

We find $f$, $g$ and $k$ by direct calculation:

$$\begin{array}{ll} & \downarrow\!/ \circ (+/)\!\!\not* \circ +\!\!+/ \circ \succ* \\ = & \qquad \{ \text{ property 2.2.4, map distribution } \} \\ & \downarrow\!/ \circ (\downarrow\!/ \circ (+/)\!\!\not* \circ \succ)* \\ = & \qquad \{ \text{ (2.33) } \} \\ & \downarrow\!/ \circ (\downarrow \circ (\downarrow\!/ \circ (+/)\!\!\not*) \times (+/)\!\!\not*)* \\ = & \qquad \{ \text{ map distribution } \} \\ & \downarrow\!/ \circ \downarrow* \circ ((\downarrow\!/ \circ (+/)\!\!\not*) \times (+/)\!\!\not*)* \end{array}$$

Thus we have found $f := \downarrow\!/ \circ \downarrow*$ and $k := (+/)\!\!\not*$. Filling these definitions in in the left-hand side of (2.35), we calculate $g$:

$$\begin{array}{ll} & (+/)\!\!\not* \circ \pi \\ = & \qquad \{ \text{ (2.27) } \} \\ & +/ \circ (+/)\!\!\not* \end{array}$$

and thus $g := +/$. Thus we can apply syzygy promotion to (2.32) to obtain:

$$\begin{array}{ll} & \downarrow\!/ \circ (+/)\!\!\not* \circ \ll \circ ((+\!\!+/ \circ \succ*) \& \pi)\!\!\not* \circ \langle (\square \ll), \rho \rangle \varrho \\ = & \qquad \{ \text{ above, syzygy promotion } \} \\ & \ll \circ ((\downarrow\!/ \circ \downarrow*) \& +/)\!\!\not* \circ ((\downarrow\!/ \circ (+/)\!\!\not*) \times (+/)\!\!\not*) \varrho \circ \langle (\square \ll), \rho \rangle \varrho \end{array}$$



$$\begin{array}{ll} = & \qquad \{ \varrho \text{ is a functor, calculus } \} \\ & \ll \circ ((\downarrow\!/ \circ \downarrow*) \& +/)\!\!\not* \circ \langle \downarrow\!/ \circ (+/)\!\!\not* \circ (\square \ll), (+/)\!\!\not* \circ \rho \rangle \varrho \\ = & \qquad \{ \text{ property of constant functions } \} \\ & \ll \circ ((\downarrow\!/ \circ \downarrow*) \& +/)\!\!\not* \circ \langle ((\downarrow\!/ . ((+/)\!\!\not* . \square)) \ll), (+/)\!\!\not* \circ \rho \rangle \varrho \\ = & \qquad \{ \text{ evaluation of catamorphisms } \} \\ & \ll \circ ((\downarrow\!/ \circ \downarrow*) \& +/)\!\!\not* \circ \langle (1_{\downarrow} \ll), \mathrm{I} \rangle \varrho \end{array}$$

The minimal sum of proper subtrees is therefore computed by the above zygomorphism, which can be evaluated in time proportional to the number of nodes in a given tree. This represents an improvement of an order of efficiency over the original algorithm, (2.32).

# Chapter 3

# Terminal Data Structures

One of the chief advantages of a categorial approach to data structures is the emphasis it lays on general and abstract properties which obtain in a host of particular instances. An example is the very general and abstract notion of algebras and homomorphisms on algebras. To say that a homomorphism respects the structure of an algebra is to say that it enjoys certain distributivity properties. The abstract formulation of initial algebras in the previous chapter brought distributivity properties to the fore, which in turn led very naturally to the statement of the promotion theorem. The usefulness of promotion was demonstrated in the proofs and program transformations contained in the previous chapter; proof by promotion will play a central role in the present chapter as well.

Another advantage of the categorial approach lies in the principle of duality, which states that the dual of every true statement is also true. Roughly, the dual of a statement is obtained by "turning all the arrows around": every function $f \in B \longleftarrow A$ becomes $f \in A \longleftarrow B$ (to keep our notation consistent, we in fact keep the arrows pointing to the left, but swap the domain and range of the function) and, accordingly, every composition $f \circ g$ becomes $g \circ f$. A fuller account of duality can be found in category theory textbooks such as Mac Lane [28]; the above rule of thumb will be sufficient for the purposes of the present chapter, namely the dualisation of the results obtained above for initial data types.

Dualising the initial Hagino data types of the previous chapter gives what we shall refer to as "terminal Hagino data types", or just "terminal types" for short. Section 3.1 below gives the details of the dualisation, and states some of the properties of terminal types, including promotion, which are obtained by dualising the results of chapter 2. Terminal types include types with infinite objects, such as infinite lists and infinite trees. As far as we are aware, little has been done in the way of systematically

applying calculational and algebraic methods to the derivation and transformation of programs on infinite data structures. Two examples of which we are aware are the theses of Hoogerwoord and Sijtsma [25,38]. Both take an essentially domain-theoretical approach, considering a type of lists which contains both finite and infinite lists. This has, we feel, the disadvantage that one must often make a case-distinction between finite and infinite lists: for example, the function which reverses a list is its own inverse only when applied to finite lists. We prefer to make such a case-distinction at the level of type definition, treating the type of finite lists and the type of infinite lists as fundamentally distinct. Each type has its own logical and algebraic structure, from which, we maintain, we should be able to extract techniques for deriving and manipulating programs on that specific type. For this reason, it is interesting to investigate to what extent the unique extension and promotion properties of the previous chapter can be usefully applied to programming with infinite data structures. Sections 3.2 and 3.3 give what we hope are convincing examples of the application of these techniques to infinite trees and infinite lists.

## 3.1 Dualising to Terminal Data Structures

Consider the initial data types of section 2.1: they were effectively defined by a number of constructors $\tau_i \in T \longleftarrow T\mathsf{F}_i$. If we turn these arrows around, then we get what we might call *destructors* $\tau_i \in T\mathsf{F}_i \longleftarrow T$. Initiality of the algebra $(T, \tau_1, \ldots, \tau_n)$ meant that for every type $P$ and functions $f_i \in P \longleftarrow P\mathsf{F}_i$ there was a unique homomorphism $(\!|f_1, \ldots, f_n|\!) \in P \longleftarrow T$ such that

$$(\!|f_1, \ldots, f_n|\!) \circ \tau_i \;=\; f_i \circ (\!|f_1, \ldots, f_n|\!)\mathsf{F}_i.$$

Turning these arrows around, we obtain the notion of a "terminal co-algebra": for every type $P$ and functions $f_i \in P\mathsf{F}_i \longleftarrow P$ there is a unique homomorphism $(\!|f_1, \ldots, f_n|\!) \in T \longleftarrow P$ such that

$$\tau_i \circ (\!|f_1, \ldots, f_n|\!) \;=\; (\!|f_1, \ldots, f_n|\!)\mathsf{F}_i \circ f_i.$$

To summarise, in the case of the initial data types, we had constructors whose range was the type being defined, and recursive catamorphisms whose domain was the type being defined; in the case of terminal data types, we have destructors whose domain is the type being defined, and recursive catamorphisms whose range is the type being defined. We shall presently see that terminal data types allow the definition of data types with infinite objects, such as infinite lists. We begin, however, with an example of dualisation: disjoint sum and its dual, cartesian product.



### 3.1.1 Duality

We illustrate the notion of duality by examining disjoint sum and its dual, cartesian product; in particular, we prove the promotion theorem for both types to show how proofs of properties of initial data types can be dualised to proofs of properties of terminal data types. We begin by restating the basic properties of disjoint sum as given in section 2.2.1 above; we shall then dualise each of these properties to obtain a corresponding property of cartesian product.

**Definition 3.1.1 (disjoint sum)** If $A$ and $B$ are types, their disjoint sum $A{+}B$ is defined by:

$$A{+}B \;\triangleq\; \mu(inl : (A\ll),\ inr : (B\ll)).$$

□

The type has therefore two constructors (the standard injections into the left and right summands):

$$\begin{aligned} inl \;&\in\; A{+}B \longleftarrow A \\ inr \;&\in\; A{+}B \longleftarrow B. \end{aligned}$$

Following the construction of section 2.1, $A{+}B$-algebras are triples $(P, f, g)$ where $f \in P \longleftarrow A$ and $g \in P \longleftarrow B$. Moreover, $(A{+}B, inl, inr)$ is the initial $A{+}B$-algebra, so that for every $f \in P \longleftarrow A$ and $g \in P \longleftarrow B$, there is a catamorphism $(\!|f, g|\!) \in P \longleftarrow A{+}B$ with the unique extension property:

$$h = (\!|f, g|\!) \;\equiv\; h \circ inl = f \;\wedge\; h \circ inr = g. \tag{3.1}$$

From the unique extension property we obtain the following identities, which prescribe the evaluation of catamorphisms:

$$(\!|f, g|\!) \circ inl \;=\; f \tag{3.2}$$

$$(\!|f, g|\!) \circ inr \;=\; g. \tag{3.3}$$

And finally the promotion theorem: we supply a proof in order to compare it with the proof of its dual, cartesian product promotion.

**Theorem 3.1.2 (promotion)** $i \circ (\!|f, g|\!) = (\!|i \circ f, i \circ g|\!)$.

**Proof:**

$$
\begin{array}{ll}
& i \circ (\!|f,g|\!) \;=\; (\!|i\circ f,\, i\circ g|\!) \\
\equiv & \quad \{\ (3.1) \text{ with } h := i \circ (\!|f,g|\!)\ \} \\
& i \circ (\!|f,g|\!) \circ inl = i \circ f \;\wedge\; i \circ (\!|f,g|\!) \circ inr = i \circ g \\
\equiv & \quad \{\ (3.2);\ (3.3)\ \} \\
& \text{true.}
\end{array}
$$

□

The dual of disjoint sum is cartesian product. We shall go through each of the above properties of disjoint sum, dualising each to obtain a corresponding property of cartesian product. The cartesian product of two types is defined from the same functors as disjoint sum; we use the symbol $\nu$ to indicate the dual construction:

**Definition 3.1.3 (cartesian product)** If $A$ and $B$ are types, their cartesian product $A\times B$ is defined by:

$$A\times B \;\triangleq\; \nu(\ll : (A\ll),\ \gg : (B\ll)).$$

□

Instead of constructors, the type has two *de*structors (the projections):

$$
\begin{array}{rcl}
\ll & \in & A \longleftarrow A\times B \\
\gg & \in & B \longleftarrow A\times B.
\end{array}
$$

Since all we are doing is turning the arrows around, it should be no surprise that $A\times B$-algebras are triples $(P,f,g)$ where $f \in A \longleftarrow P$ and $g \in B \longleftarrow P$ — i.e., $f \in P(A\ll) \longleftarrow P$ and $g \in P(B\ll) \longleftarrow P$. Above, we defined $(A+B, inl, inr)$ to be the *initial* $A+B$-algebra; this is dualised by defining $(A\times B, \ll, \gg)$ to be the *terminal* $A\times B$-algebra. That is, there is exactly one homomorphism *to* $(A\times B, \ll, \gg)$ *from* any other $A\times B$-algebra; in other words, for every $f \in A \longleftarrow P$ and $g \in B \longleftarrow P$ we have the catamorphism $(\!|f,g|\!) \in A\times B \longleftarrow P$ with the unique extension property that for all $h \in A\times B \longleftarrow P$,

$$h = (\!|f,g|\!) \quad\equiv\quad \ll \circ h = f \;\wedge\; \gg \circ h = g. \tag{3.4}$$

(Catamorphisms $(\!|f,g|\!)$ on cartesian products are therefore the "pair" functions $\langle f,g\rangle$.) This is the dual of the unique extension property for disjoint sum, (3.1), in that all compositions have been reversed and the constructors *inl* and *inr* have been replaced



by the destructors $\ll$ and $\gg$ respectively. By the same process, the evaluation rules for disjoint sum catamorphisms, equations (3.2) and (3.3), can be dualised to obtain the following identities, which similarly prescribe the evaluation of cartesian product catamorphisms.

$$\ll \circ (\!|f,g|\!) \;=\; f \tag{3.5}$$

$$\gg \circ (\!|f,g|\!) \;=\; g. \tag{3.6}$$

We now state and prove the promotion theorem for cartesian product. The proof is redundant in the sense that it is the dual of the proof of disjoint sum promotion.

**Theorem 3.1.4 (promotion)** $(\!|f,g|\!) \circ i \;=\; (\!|f\circ i,\, g\circ i|\!)$.

**Proof:**

$$
\begin{array}{ll}
& (\!|f,g|\!) \circ i \;=\; (\!|f\circ i,\, g\circ i|\!) \\
\equiv & \quad \{\ (3.4) \text{ with } h := (\!|f,g|\!) \circ i\ \} \\
& \ll \circ (\!|f,g|\!) \circ i = f \circ i \;\wedge\; \gg \circ (\!|f,g|\!) \circ i = g \circ i \\
\equiv & \quad \{\ (3.5);\ (3.6)\ \} \\
& \text{true.}
\end{array}
$$

□

Comparing the two proofs, we see that in each step, all compositions are reversed, and the constructors *inl* and *inr* are replaced by the destructors $\ll$ and $\gg$. Moreover, in the hints between the steps, the appeal to (3.1) is replaced by an appeal to its dual, (3.4); likewise, (3.2) and (3.3) are replaced by their duals, (3.5) and (3.6). In future, when a property of a terminal data type is the dual of an already proven property of an initial data type, we shall not give its proof, but simply appeal directly to the principle of duality. There are many properties of initial data types that can be usefully dualised. Section 3.1.3 below gives a list of such properties, obtained by dualising the results of section 2.3 on parameterised data types.

## 3.1.2 Promotability for Terminal Structures

In this section we give the details of the construction of terminal data types, and state the promotion theorem for an arbitrary terminal data type. As with initial data types, terminal data types are defined by a number of type functors. These functors

induce a class of co-algebras, of which the defined type together with its destructors is the terminal co-algebra. Just as initiality gave rise to a unique extension property for catamorphisms in the case of initial data types, so terminality yields a similar unique extension property in the case of terminal data types. When defining a terminal type, however, we shall not go through the construction of the algebras, but instead we state directly the unique extension property for that type.

**Definition 3.1.5 (terminal Hagino types)** Let $\mathsf{F}_1, \ldots, \mathsf{F}_n$ be type functors, each of one argument. The terminal type $\Upsilon = \nu(\upsilon_1 : \mathsf{F}_1, \ldots, \upsilon_n : \mathsf{F}_n)$, has destructors $\upsilon_i$ whose type is determined by the functor $\mathsf{F}_i$ according to

$$\upsilon_i \in \Upsilon_{\mathsf{F}_i} \longleftarrow \Upsilon.$$

The type is defined up to isomorphism by the property that for each type $P$ and functions $f_i \in P_{\mathsf{F}_i} \longleftarrow P$, there is a catamorphism $(\!|f_1, \ldots, f_n|\!) \in \Upsilon \longleftarrow P$ with the unique extension property that for all $h \in \Upsilon \longleftarrow P$,

$$h = (\!|f_1, \ldots, f_n|\!) \quad \equiv \quad \forall(i :: \upsilon_i \circ h = h_{\mathsf{F}_i} \circ f_i). \tag{3.7}$$

By taking $h := (\!|f_1, \ldots, f_n|\!)$, the left-hand side of the above becomes true, yielding: for all $i$,

$$\upsilon_i \circ (\!|f_1, \ldots, f_n|\!) = (\!|f_1, \ldots, f_n|\!)_{\mathsf{F}_i} \circ f_i. \tag{3.8}$$

These equations effectively determine how such a catamorphism is evaluated.
□

We give the type of infinite lists as an example.

**Example 3.1.6 (infinite lists)** For each type $A$, define the type of infinite lists over $A$ by:

$$A\infty \triangleq \nu(hd : (A\ll),\ tl : \imath).$$

Thus the type has two destructors $hd \in A \longleftarrow A\infty$ and $tl \in A\infty \longleftarrow A\infty$. Given an infinite list, one can imagine $hd$ as returning the head and $tl$ the tail of that list.
□

According to definition 3.1.5, an infinite list may be constructed by means of functions $f \in A \longleftarrow P$ and $g \in P \longleftarrow P$, which induce a catamorphism $(\!|f, g|\!) \in A\infty \longleftarrow P$. Corresponding to equations (3.8), this catamorphism is the unique function satisfying:

$$hd \circ (\!|f, g|\!) = f \tag{3.9}$$

$$tl \circ (\!|f, g|\!) = (\!|f, g|\!) \circ g. \tag{3.10}$$

That is, given an object $p \in P$, we can construct the infinite list $(\!|f, g|\!).p$ whose head is given by $f.p$ and whose tail is given by $(\!|f, g|\!).(g.p)$; applying $hd$ to this latter gives the second element of the list, $f.(g.p)$, and so on. In general, the $(n+1)$th element of the list is $f.(g^n.p)$. The reader may visualise the infinite list $(\!|f, g|\!).p$ as the sequence

$$f.p\ ,\ f.(g.p)\ ,\ f.(g^2.p)\ ,\ f.(g^3.p)\ , \ldots$$

but we shall always treat catamorphisms as "lazy" functions, so that expressions of the form $(\!|f, g|\!).p$ are considered to be in canonical form, and not to be further evaluated: the only computation rules for catamorphisms on infinite lists are those given by equations (3.9) and (3.10) above.

An elementary example of a catamorphism on infinite lists is the identity catamorphism, $(\!|hd, tl|\!)$. Given an infinite list $l$, the $(n+1)$th element of the list $(\!|hd, tl|\!).l$ is computed by $hd.(tl^n.l)$, which strongly suggests that $(\!|hd, tl|\!).l = l$. Formally, this identity holds for the general case as a corollary to the uniqueness property (3.7) of catamorphisms.

**Corollary 3.1.7 (identity)** $\mathrm{I} = (\!|\upsilon_1, \ldots, \upsilon_n|\!) \in \Upsilon \longleftarrow \Upsilon$.

**Proof:** dual of corollary 2.1.4
□

The promotion theorem for terminal data structures is the dual of that for initial data structures. Promotion theorems for the particular data types which we consider below are instantiations of this theorem.

**Theorem 3.1.8 (promotion)** For a terminal type $\nu(\upsilon_1 : \mathsf{F}_1, \ldots, \upsilon_n \mathsf{F}_n)$,

$$(\!|g_1, \ldots, g_n|\!) \circ h = (\!|f_1, \ldots, f_n|\!) \quad \Leftarrow \quad \forall(i :: g_i \circ h = h_{\mathsf{F}_i} \circ f_i)$$

**Proof:** dual of theorem 2.1.6.
□

The instantiation of the promotion theorem for infinite lists is, after slight simplification:

**Theorem 3.1.9 ($\infty$-promotion)**

$$(\!|f,g|\!) \circ h = (\!|f \circ h, i|\!) \quad \Leftarrow \quad g \circ h = h \circ i.$$

□

We shall give some examples of the use of promotion in sections 3.2 and 3.3 below; we conclude our brief examination of duality by listing several further properties of terminal types, obtained by duality from section 2.3.

### 3.1.3 Properties Obtained By Duality

The purpose of this section is to list basic properties of parameterised terminal data types, each of which is the dual of a property proven in the previous chapter for initial data types. As such, these properties are derived in a purely mechanical fashion, and the reader is invited to skim through this section quickly. The point we wish to emphasise is that fundamental properties of terminal types, such as the map operator's distributing through composition, are straightforward corollaries of the corresponding properties of initial types.

We begin with properties of maps; let $\varpi$ be the type constructor defined by $A\varpi \triangleq \nu(\delta : (A \hat{\otimes}))$, for all $A$.

**Definition 3.1.10 (maps)** For $f \in B \longleftarrow A$, we define the map function:

$$f\varpi \triangleq (\!|f \hat{\otimes} \mathrm{I} \circ \delta|\!) \in B\varpi \longleftarrow A\varpi$$

(cf definition 2.3.1).
□

**Property 3.1.11** $f\varpi \circ (\!|g|\!) = (\!|f \hat{\otimes} \mathrm{I} \circ g|\!)$.
**Proof:** dual of property 2.3.3.
□

**Property 3.1.12** $\varpi$ is a type functor; i.e., $\mathrm{I}\varpi = \mathrm{I}$ and $(f \circ g)\varpi = f\varpi \circ g\varpi$.
**Proof:** dual of properties 2.3.2 and 2.3.4.
□



**Property 3.1.13** $\delta : \mathrm{I} \hat{\otimes} \varpi \stackrel{\cdot}{\longleftarrow} \varpi$.
**Proof:** dual of property 2.3.5.
□

**Property 3.1.14 (naturality)** If $\phi : \mathsf{F} \hat{\otimes} \mathsf{G} \stackrel{\cdot}{\longleftarrow} \mathsf{G}$, then $(\!|\phi|\!) : \mathsf{F}\varpi \stackrel{\cdot}{\longleftarrow} \mathsf{G}$.
**Proof:** dual of property 2.3.6
□

We turn now to factorisation properties of catamorphisms. A terminal data type is free under the same conditions as for finite data types, i.e., if it can be expressed in the form $\nu(\eta : (A \ll), \theta : \mathsf{F})$, where $\mathsf{F}$ does not depend upon $A$. Let $A\varpi$ satisfy this condition, then the following properties hold.

**Definition 3.1.15 (reductions)** For $g \in A\mathsf{F} \longleftarrow A$, define the reduction $g/$ by

$$g/ \triangleq (\!|\mathrm{I}, g|\!) \in A\varpi \longleftarrow A$$

(cf definition 2.3.9).
□

**Theorem 3.1.16 (factorisation)** For all catamorphisms $(\!|f,g|\!)$,

$$(\!|f,g|\!) = f\varpi \circ g/.$$

**Proof:** dual of theorem 2.3.10.
□

The promotion theorem for free data types admits of the following simplification:

**Property 3.1.17**

$$g/ \circ h = h\varpi \circ i/ \quad \Leftarrow \quad g \circ h = h\mathsf{F} \circ i.$$

**Proof:** dual of (2.28).
□

**Definition 3.1.18 (comonads)** A *comonad* is a triple $(\mathsf{T}, \eta, \zeta)$, where $\eta$ and $\zeta$ are natural transformations, $\eta : \mathsf{I} \xleftarrow{\cdot} \mathsf{T}$ and $\zeta : \mathsf{TT} \xleftarrow{\cdot} \mathsf{T}$, such that:

$$\begin{aligned} \eta\mathsf{T} \circ \zeta &= \mathrm{I} \\ \eta \circ \zeta &= \mathrm{I} \\ \zeta \circ \zeta &= \zeta\mathsf{T} \circ \zeta. \end{aligned}$$

(cf definition 2.3.11)
□

**Theorem 3.1.19** $(\varpi, \eta, \theta/)$ is a comonad.

**Proof:** dual of theorem 2.3.13.
□

The function $\theta/ \in A\varpi\varpi \longleftarrow A\varpi$ satisfies the following identities, which are the duals of properties 2.3.12 and 2.3.16:

$$\theta/ \circ f\varpi = f\varpi\varpi \circ \theta/ \tag{3.11}$$

$$\theta/ \circ g/ = g/\varpi \circ g/. \tag{3.12}$$

## 3.2 Infinite Multiway Trees

In this section we give an example of a terminal data structure: infinite multiway trees. An infinite multiway tree consists of an internal label and a list of subtrees (thus the tree has an arbitrary branching factor). The type should have one destructor to access the internal label at the root of the tree, and one to access the list of subtrees: this consideration leads to the following definition of the type.

**Definition 3.2.1 (multiway trees)** We define the type of infinite multiway trees over $A$ by

$$A\varpi \triangleq \nu(rt : A\ll, sb : *).$$

Thus $A\varpi$ has destructors $rt \in A \longleftarrow A\varpi$ and $sb \in A\varpi * \longleftarrow A\varpi$ which return the root label and the subtrees respectively of a given tree, and for every $f \in A \longleftarrow P$ and $g \in P* \longleftarrow P$, there is a unique function $(\!|f,g|\!) \in A\varpi \longleftarrow P$ such that:

$$rt \circ (\!|f,g|\!) = f \tag{3.13}$$

$$sb \circ (\!|f,g|\!) = (\!|f,g|\!)* \circ g. \tag{3.14}$$



That is, the label at the root of the tree is generated by $f$, and the subtrees are generated by recursively mapping the catamorphism to the list given by $g$. (Note that such a tree may have finite depth if all branches eventually lead to empty lists of subtrees.)
□

From the previous section we already know a lot about the data type: its promotion theorem, the definition of maps, that $\varpi$ is a functor, and so on. Moreover, the functors in the definition of multiway trees are the same as those in the definition of rose trees (section 2.3.3), which means that the type is free, and hence all catamorphisms can be factored into the composition of a map and a reduction. That is,

$$(\!|f,g|\!) = f\varpi \circ g/,$$

where $f\varpi = (\!|f \circ rt, sb|\!)$ and $g/ = (\!|1,g|\!)$. A simple example of a multiway tree reduction is the following:

**Example 3.2.2 (game trees)** Let $A$ be a type which represents valid states of some game and let $mv \in A* \longleftarrow A$ be a function which returns the list of valid states which may be reached in one move from a given state. The reduction $mv/ \in A\varpi \longleftarrow A$ generates the game tree of all possible states which may be reached from a given state.
□

Stated in terms of maps and reductions, the unique extension property for the type of infinite multiway trees is that for all $h$,

$$h = f\varpi \circ g/ \quad \equiv \quad rt \circ h = f \ \wedge \ sb \circ h = h* \circ g.$$

As ever, the unique extension property gives rules for the evaluation of catamorphisms. From the definitions of maps and reductions we obtain:

$$rt \circ f\varpi = f \circ rt \tag{3.15}$$

$$sb \circ f\varpi = f\varpi* \circ sb \tag{3.16}$$

$$rt \circ g/ = \mathrm{I} \tag{3.17}$$

$$sb \circ g/ = g/* \circ g \tag{3.18}$$

Since $\varpi$ is free, the promotion theorem can be simplified to (see property 3.1.17):

$$g/ \circ h = h\varpi \circ i/ \quad \Leftarrow \quad g \circ h = h* \circ i. \tag{3.19}$$

The freeness of $\varpi$ also gives us the following special cases of promotion, the map and reduce promotion laws (see (3.11) and (3.12) of section 3.1.3 above). Map promotion:

(3.20) $$sb/ \circ h\varpi = h\varpi\varpi \circ sb/.$$

Reduce promotion:

(3.21) $$sb/ \circ g/ = g/\varpi \circ g/.$$

In our discussion of freeness in chapter 2 we claimed that the various properties of free types led to very direct program manipulations. We give an example of this concerning two simple operations on trees: a "pruning" operation and a "breadth" function. We start with their definitions.

The idea behind the breadth function, $B \in A*\varpi \longleftarrow A\varpi$, is that, given a tree as an argument, it recursively replaces each label in the tree by the list of its immediate descendants, i.e., the roots of each of the subtrees of the node. That is, the root of the tree is obtained by mapping $rt$ to the list of subtrees:

$$rt \circ B = rt* \circ sb$$

and the subtrees are obtained by recursively mapping $B$ to the subtrees of the given tree:

$$sb \circ B = B* \circ sb.$$

From these recurrence relations, the unique extension property gives the following definition of $B$:

$$B = (rt* \circ sb)\varpi \circ sb/.$$

So much for the breadth function; in order to define the pruning operation, we first introduce the notion of filters on finite lists.

If $p \in \mathrm{Bool} \longleftarrow A$ is a predicate on $A$, then the filter of $p$ is a function $p\triangleleft \in A* \longleftarrow A*$ which removes all elements from a given list which do not satisfy $p$. The function is a join-list catamorphism:

**Definition 3.2.3 (filter)** $p\triangleleft \triangleq +\!\!+/ \circ \vec{p}*$, where for all $a \in A$,

$$\vec{p}.a = \begin{cases} \eta.a & \text{if } p.a \\ \Box & \text{if } \neg p.a \end{cases}$$

□



We shall need the following property of filters, called the "trading law" by Bird: for all $f$,

(3.22) $$p\triangleleft \circ f* = f* \circ (p \circ f)\triangleleft.$$

Now, our pruning operation on infinite trees, which we shall also denote by $p\triangleleft \in A\varpi \longleftarrow A\varpi$, given a tree as argument, constructs a tree whose root is the root of the given tree:

$$rt \circ p\triangleleft = rt$$

and whose subtrees are constructed from the list of subtrees of the given tree by filtering out all those subtrees whose roots do not satisfy $p$, and then recursively applying $p\triangleleft$ to each of the remaining subtrees:

$$sb \circ p\triangleleft = p\triangleleft* \circ (p \circ rt)\triangleleft \circ sb.$$

The unique extension property gives therefore

$$p\triangleleft = rt\varpi \circ ((p \circ rt)\triangleleft \circ sb)/ \in A\varpi \longleftarrow A\varpi.$$

Functions built up from constructors and destructors generally enjoy many nice algebraic properties: the prime example is $+\!\!+/$. The occurrences of $rt$ and $sb$ in the above definition are promising. In fact, pruning enjoys the following two properties.

**Property 3.2.4 (trading)** $p\triangleleft \circ f\varpi = f\varpi \circ (p \circ f)\triangleleft$.

**Proof:** We anticipate using promotion, so we begin by noting the following distributivity property:

$$\begin{array}{ll} & (p \circ rt)\triangleleft \circ sb \circ f\varpi \\ = & \quad \{\ (3.16)\ \} \\ & (p \circ rt)\triangleleft \circ f\varpi* \circ sb \\ = & \quad \{\ (3.22)\ \} \\ & f\varpi* \circ (p \circ rt \circ f\varpi)\triangleleft \circ sb \end{array}$$

so promotion is applicable and we calculate:

$$\begin{array}{ll} & p\triangleleft \circ f\varpi \\ = & \quad \{\ \triangleq\triangleleft\ \} \\ & rt\varpi \circ ((p \circ rt)\triangleleft \circ sb)/ \circ f\varpi \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ above, promotion } \} \\
& rt\varpi \circ f\varpi\varpi \circ ((p \circ rt \circ f\varpi) \triangleleft \circ sb)/\!/ \\
= & \{ \ \varpi \text{ is a functor; (3.15), twice } \} \\
& f\varpi \circ rt\varpi \circ ((p \circ f \circ rt) \triangleleft \circ sb)/\!/ \\
= & \{ \triangleq \triangleleft \} \\
& f\varpi \circ (p \circ f) \triangleleft
\end{array}
$$

□

Property 3.2.5 $p\triangleleft \circ g/ = (p\triangleleft \circ g)/\!/$.

Proof: Again, we foresee using promotion and note

$$
\begin{array}{ll}
& (p \circ rt) \triangleleft \circ sb \circ g/ \\
= & \{ \ (3.18) \ \} \\
& (p \circ rt) \triangleleft \circ g/\!* \circ g \\
= & \{ \ (3.22) \ \} \\
& g/\!* \circ (p \circ rt \circ g/) \triangleleft \circ g
\end{array}
$$

Promotion is therefore once again applicable, and we calculate:

$$
\begin{array}{ll}
& p\triangleleft \circ g/ \\
= & \{ \triangleq \triangleleft \} \\
& rt\varpi \circ ((p \circ rt) \triangleleft \circ sb)/\!/ \circ g/ \\
= & \{ \text{ above, promotion } \} \\
& rt\varpi \circ g/\varpi \circ ((p \circ rt \circ g/) \triangleleft \circ g)/\!/ \\
= & \{ \ \varpi \text{ is a functor; (3.17), twice } \} \\
& (p\triangleleft \circ g)/\!/
\end{array}
$$

□

Let us now put some of this theory into practice. Consider the function $p\triangleleft \circ B \circ mv/$, which prunes the breadth of the game tree generated by $mv \in A\!*\!\longleftarrow\!A$. It is difficult to estimate the efficiency of this program, since we assume that its evaluation will be lazy. Since both $p\triangleleft$ and $B$ contain occurrences of $rt$ and $sb$, the number of unfolding operations (i.e., applications of $rt$ and $sb$) required to access nodes at a given depth



of the resulting tree depends upon the size of the lists produced by $mv$. In any case, we can transform this program into one which requires only $n$ unfolding operations to access nodes at depth $n$:

$$
\begin{array}{ll}
& p\triangleleft \circ B \circ mv/ \\
= & \{ \triangleq B \} \\
& p\triangleleft \circ (rt\!*\!\circ sb)\varpi \circ sb/ \circ mv/ \\
= & \{ \text{ reduce promotion (3.21) } \} \\
& p\triangleleft \circ (rt\!*\!\circ sb)\varpi \circ mv/\varpi \circ mv/ \\
= & \{ \ \varpi \text{ is a functor } \} \\
& p\triangleleft \circ (rt\!*\!\circ sb \circ mv/)\varpi \circ mv/ \\
= & \{ \text{ (3.18); map distribution } \} \\
& p\triangleleft \circ ((rt \circ mv/)\!*\!\circ mv)\varpi \circ mv/ \\
= & \{ \text{ (3.17); I}* = \text{I } \} \\
& p\triangleleft \circ mv\varpi \circ mv/ \\
= & \{ \text{ property 3.2.4 } \} \\
& mv\varpi \circ (p \circ mv) \triangleleft \circ mv/ \\
= & \{ \text{ property 3.2.5 } \} \\
& mv\varpi \circ ((p \circ mv) \triangleleft \circ mv)/\!/ \\
= & \{ \ mv\!*\!\circ (p \circ mv) \triangleleft \circ mv = p\triangleleft \circ mv\!*\!\circ mv, \text{ promotion } \} \\
& (p\triangleleft \circ mv\!*\!)/\!/ \circ mv
\end{array}
$$

The important steps in the above calculation come in the first half, where the occurrences of the destructors $rt$ and $sb$ contained in $p\triangleleft$ and $B$ are eliminated. This has the beneficial effect of making the number of unfolding operations required to access nodes in the resulting program independent of the size of the lists generated by the function $mv$.

## 3.3 Infinite List Programs

We conclude this chapter with a look at some simple programs on infinite lists. Most of the properties of infinite lists that we shall use have already been stated in section 3.1

for an arbitrary terminal data type; we begin by instantiating some of these properties. First of all, recall that we defined the type of infinite lists by

$$A\infty \triangleq \nu(hd : (A\ll), tl : \mathrm{I}).$$

Now, the identity functor I does not depend upon $A$. The type of infinite lists is therefore free (see the remark before property 3.1.15 in section 3.1.3 above), and so all catamorphisms can be factored into the composition of a map and a reduction. We shall accordingly couch all our properties of $\infty$-catamorphisms in terms of maps and reductions. These functions are constructed as follows: for all $f \in A \longleftarrow P$ and $g \in P \longleftarrow P$, there is a catamorphism $f\infty \circ g/ \in A\infty \longleftarrow P$ with the unique extension property that for all $h$,

$$h = f\infty \circ g/ \quad \equiv \quad hd \circ h = f \ \wedge \ tl \circ h = h \circ g.$$

Maps and reductions are themselves catamorphisms, as defined in section 3.1.3; alternatively, to save the reader's having to flick back to the relevant page, we can point out that since the identity function on $A\infty$ is $hd\infty \circ tl/$ and since $\infty$ is a functor and so respects identity,

$$f\infty = f\infty \circ hd\infty \circ tl/ = (f \circ hd)\infty \circ tl/;$$

moreover, since $\infty$ is a functor and so respects identity,

$$g/ = 1\infty \circ g/.$$

By inserting these equalities into the unique extension property we obtain the rules for the evaluation of maps and reductions:

$$hd \circ f\infty = f \circ hd \tag{3.23}$$

$$tl \circ f\infty = f\infty \circ tl \tag{3.24}$$

$$hd \circ g/ = 1 \tag{3.25}$$

$$tl \circ g/ = g/ \circ g \tag{3.26}$$

Finally, we restate the promotion theorem for infinite lists. The freeness of $\infty$ means that we need only consider promotion over reductions:

$$h\infty \circ g/ = i/ \circ h \quad \Leftarrow \quad h \circ g = i \circ h.$$



The above properties are the ones we shall most often have recourse to. Of course, the unique extension property characterises the type uniquely up to isomorphism, and, insofar as each of the above properties is a consequence of this one defining property, they are luxuries that, strictly speaking, can be done without. Nevertheless, the point we wish to make here is that the unique extension property on the one hand, and on the other hand the derived properties such as those stated above, have each their own usefulness. The unique extension property is mainly of use when one wishes to express a given function as a catamorphism. The point of expressing a function as a catamorphism is that catamorphisms are preeminently *manipulable*, and it is in this sense that the unique extension property is subordinate to its corollaries: that the promotion theorem and the evaluation rules for catamorphisms lend themselves more readily to "massaging" a program into a more efficient form, or to proving properties of programs. The following examples provide support for this claim.

## 3.3.1 Lists of Function Results

It is well known that the type $A\infty$ is isomorphic to the type $A \longleftarrow \mathbb{N}$; we shall not prove that isomorphism here, but we note that one half of the bijection takes $f \in A \longleftarrow \mathbb{N}$ to $f\infty.(\sigma/.0)$, where $\sigma$ is the successor function on $\mathbb{N}$. The catamorphism $\sigma/$ applied to 0 produces the infinite list of natural numbers, to which the function $f$ is then mapped, producing the infinite list of the results of $f$:

$$f.0\,,\ f.(\sigma.0)\,,\ f.(\sigma(\sigma.0))\,,\ f.(\sigma(\sigma(\sigma.0)))\ldots.$$

In his thesis, [38], Sijtsma considers infinite lists of function results: his general approach is domain-theoretical and he considers a type of lists which contains finite, partial and infinite lists. Many of his programs are constructed by means of a fixed point combinator, and by introducing a new ordering on lists he obtains a wide class of permissible predicates for fixed point induction. Partial functions, fixed point combinators and fixed point induction are far beyond the scope of the present chapter, where we are concerned with investigating properties of catamorphisms — these are, after all, total functions — but the following is a special case of Sijtsma's "Heuristic 1" (loc. cit., theorem 6.17). Let $f \in A \longleftarrow \mathbb{N}$ and $g \in A \longleftarrow A$ be such that $g$ gives the next element in the enumeration of $f$'s results, i.e., $f \circ \sigma = g \circ f$. Then $g$ can be used to construct the list of $f$'s results:

$$f\infty \circ \sigma/ = g/ \circ f,$$

and hence

$$f\infty.(\sigma/.0) = g/.(f.0).$$

This is, of course, just an application of $\infty$-promotion, but it is a transformation that can always be applied when $f$ is an $\mathbb{N}$-catamorphism. Recall that the evaluation rules for $\mathbb{N}$-catamorphisms were:

$$\begin{aligned} (\!|e,g|\!)_{\mathbb{N}}.0 &= e \\ (\!|e,g|\!)_{\mathbb{N}} \circ \sigma &= g \circ (\!|e,g|\!)_{\mathbb{N}}. \end{aligned}$$

The second of these equalities has the form of the antecedent of $\infty$-promotion:

**Property 3.3.1 ($\mathbb{N}$-catamorphisms)** For all $\mathbb{N}$-catamorphisms $(\!|e,g|\!)_{\mathbb{N}}$,

$$(\!|e,g|\!)_{\mathbb{N}}\infty \circ \sigma/ = g/ \circ (\!|e,g|\!)_{\mathbb{N}}.$$

And so, using the first evaluation rule for $\mathbb{N}$-catamorphisms,

$$(\!|e,g|\!)_{\mathbb{N}}\infty.(\sigma/.0) = g/.e.$$

□

Consider, for example, the Fibonacci function $fib \in \mathbb{N} \longleftarrow \mathbb{N}$ which satisfies the well known equations:

$$\begin{aligned} fib.0 &= 0 \\ fib.1 &= 1 \\ fib.(i+2) &= fib.i + fib.(i+1). \end{aligned}$$

The last of these may be expressed functionally as

$$fib \circ \sigma \circ \sigma = + \circ \langle fib, fib \circ \sigma \rangle.$$

Applying the by now standard machinery, a few calculations and the unique extension property for $\mathbb{N}$-catamorphisms show that

$$\langle fib, fib \circ \sigma \rangle = (\!|(0,1), \langle \gg, + \rangle|\!)_{\mathbb{N}}$$

and so

$$fib = \ll \circ (\!|(0,1), \langle \gg, + \rangle|\!)_{\mathbb{N}}.$$

The Fibonacci series is generated by the program $fib\infty.(\sigma/.0)$. Using the above property, we simplify this as follows;



$$\begin{aligned} & fib\infty.(\sigma/.0) \\ = & \quad \{ \triangleq fib \} \\ & (\ll \circ (\!|(0,1), \langle \gg, + \rangle|\!)_{\mathbb{N}})\infty.(\sigma/.0) \\ = & \quad \{ \text{property 3.3.1, map distribution} \} \\ & \ll\infty.(\langle \gg, + \rangle/.(0,1)) \end{aligned}$$

Although both programs take the same time to compute the $n$th element of the list, the above calculation represents an increase in efficiency in that the transformed program computes the first $n$ elements of the list in time proportional to $n$, whereas the original program was quadratic. The increase in efficiency is due to the fact that the original program calculates each of the first $n$ elements independently, whereas the transformed program uses the preceding values in computing the next value in the list.

## 3.3.2 Initial Segments

The literature on the Bird-Meertens Formalism abounds with problems involving the computation of segments of finite lists (see, for example, Bird [7], Meertens [34]). In this section, we derive some general transformations on programs on the initial segments of infinite lists. The goal is to provide on-line solutions to segment problems: that is, programs which produce an infinite list of successive results, outputting a new result in constant time after a new input is read. In particular, corollary 3.3.5 below shows that mapping a linear snoc-list catamorphism to the list of initial segments of an infinite list can be transformed to a constant-time on-line algorithm. We begin by deriving an implementation of a function which produces an initial segment of an infinite list.

Let $a \in A\infty$; we seek an implementation of $F \in A* \longleftarrow \mathbb{N}$ such that, given $n \in \mathbb{N}$, $F$ produces the initial segment of $a$ of length $n$. This gives the recurrence relations:

$$\begin{aligned} F.0 &= nil \\ F.(n+1) &= F.n \succ hd.(tl^n.a). \end{aligned}$$

Expressing the latter equality functionally:

$$F \circ \sigma = \succ \circ 1 \times hd \circ \langle F, G \rangle,$$

where $G.n = tl^n.a$. If we can express $G$ as a $\mathbb{N}$-catamorphism $(\!|e, g|\!)_{\mathbb{N}}$, then the uniqueness of zygomorphisms property (property 2.4.1 of section 2.4) gives an implementation of $F$ as the zygomorphism

$$\ll \circ (\!|(nil, e), (\succ \circ \mathrm{I}{\times}hd)\&g|\!)_{\mathbb{N}},$$

where for all $\oplus$ and $h$, $\oplus\&h = \langle \oplus, h \circ \gg \rangle$. Now the recurrence relations for $G$ are straightforward: $G.0 = tl^0.a = a$, and $G.(n+1) = tl^{n+1}.a = tl.(G.n)$. So, by the unique extension property for $\mathbb{N}$-catamorphisms, $G = (\!|a, tl|\!)_{\mathbb{N}}$ and therefore

$$F = \ll \circ (\!|(nil, a), (\succ \circ \mathrm{I}{\times}hd)\&tl|\!)_{\mathbb{N}}.$$

From the previous section, we know that the function $F \in A*\!\longleftarrow\!\mathbb{N}$ induces an infinite list of the initial segments of $a$, which is given by

$$\begin{array}{ll} & F.(\sigma/.0) \\ = & \qquad \{ \text{ property 3.3.1 } \} \\ & \ll\!\infty.(((\succ \circ \mathrm{I}{\times}hd)\&tl)/.(nil, a)) \end{array}$$

By abstracting on $nil$ and $a$, we have found the initial segments function,

$$\ll\!\infty \circ ((\succ \circ \mathrm{I}{\times}hd)\&tl)/ \in A*\infty\!\longleftarrow\!A*{\times}A\infty$$

which, given $l \in A*$ and $a \in A\infty$ as arguments, returns the infinite list of the initial segments of $a$, each prefixed by the list $l$. Let us immediately generalise this by giving a name to functions of this form:

**Definition 3.3.2 (accumulation)** Let $\oplus \in B\!\longleftarrow\!B{\times}A$; we define the *accumulation* of $\oplus$, denoted by $[\![\oplus]\!]$, to be the function:

$$\ll\!\infty \circ \vec{\oplus}/ \in B\infty\!\longleftarrow\!B{\times}A\infty$$

where $\vec{\oplus} \triangleq (\oplus \circ \mathrm{I}{\times}hd)\&tl$.
□

The accumulation $[\![\oplus]\!]$ takes $b \in B$ and infinite list $a_1, a_2, a_3, \ldots$ into the infinite list

$$b,\ b \oplus a_1,\ (b \oplus a_1) \oplus a_2,\ ((b \oplus a_1) \oplus a_2) \oplus a_3,\ \ldots$$

(it is, in fact, the "scan" function of Bird and Wadler [10]). In this notation, the initial segments function becomes $[\![\succ]\!]$. Since accumulations have been introduced as a generalisation of this function, we might expect snoc list promotion to be in some way applicable. A first step is the following distributivity property:



**Lemma 3.3.3** $f \circ \oplus = \otimes \circ f{\times}\mathrm{I} \quad \Rightarrow \quad f{\times}\mathrm{I} \circ \vec{\oplus} = \vec{\otimes} \circ f{\times}\mathrm{I}.$

**Proof:**

$$\begin{array}{ll} & f{\times}\mathrm{I} \circ \vec{\oplus} \\ = & \qquad \{ \triangleq \vec{\oplus}, \text{ calculus } \} \\ & \langle f \circ \oplus \circ \mathrm{I}{\times}hd,\ tl \circ \gg \rangle \\ = & \qquad \{ \text{ assumption: } f \circ \oplus = \otimes \circ f{\times}\mathrm{I} \ \} \\ & \langle \otimes \circ f{\times}hd,\ tl \circ \gg \rangle \\ = & \qquad \{ \text{ calculus } \} \\ & \langle \otimes \circ \mathrm{I}{\times}hd \circ f{\times}\mathrm{I},\ tl \circ \gg \circ f{\times}\mathrm{I} \rangle \\ = & \qquad \{ \text{ calculus, } \triangleq \vec{\otimes} \ \} \\ & \vec{\otimes} \circ f{\times}\mathrm{I} \end{array}$$

□

The importance of this lemma is that the distributivity property in the consequent is just what we need to apply $\infty$-promotion. We have almost proven

**Property 3.3.4** $f \circ \oplus = \otimes \circ f{\times}\mathrm{I} \quad \Rightarrow \quad f\infty \circ [\![\oplus]\!] = [\![\otimes]\!] \circ f{\times}\mathrm{I}.$

**Proof:**

$$\begin{array}{ll} & [\![\otimes]\!] \circ f{\times}\mathrm{I} \\ = & \qquad \{ \text{ definition 3.3.2 } \} \\ & \ll\!\infty \circ \vec{\oplus}/ \circ f{\times}\mathrm{I} \\ = & \qquad \{ \text{ lemma 3.3.3, promotion } \} \\ & \ll\!\infty \circ (f{\times}\mathrm{I})\infty \circ \vec{\otimes}/ \\ = & \qquad \{ \infty \text{ is a functor, calculus } \} \\ & f\infty \circ \ll\!\infty \circ \vec{\otimes}/ \\ = & \qquad \{ \text{ definition 3.3.2 } \} \\ & f\infty \circ [\![\otimes]\!] \end{array}$$

□

Accumulations were introduced by generalising the initial segment function; this generalisation reappears in the following corollary.

**Corollary 3.3.5** For all snoc list catamorphisms $(\!|e, \otimes|\!)_{\bullet}$,

$$(\!|e, \otimes|\!)_{\bullet}\infty \circ [\![\succ]\!] = [\![\otimes]\!] \circ (\!|e, \otimes|\!)_{\bullet} \times \mathrm{I}.$$

**Proof:** Immediate from the preceding property and the evaluation rule for snoc list catamorphisms:

$$(\!|e, \otimes|\!)_{\bullet} \circ \succ = \otimes \circ (\!|e, \otimes|\!)_{\bullet} \times \mathrm{I}.$$

□

This corollary, just like property 3.3.1 in the previous section, represents a considerable increase in efficiency. Assuming that $\otimes$ can be evaluated in constant time, the time required to compute the first $n$ elements of the list produced by the left-hand side program is quadratically proportional to $n$, whereas the right-hand side program is linear. It can be implemented as an on-line algorithm in which the next result is output in constant time after reading a new input.

As an example, we give a program to compute the maximum sum of the segments of a given infinite list. The segments of an infinite list are enumerated by mapping the "tails" function to the initial segments of the list:

$$tls\infty \circ [\![\succ]\!]$$

where $tls = (\!|nil, \oplus|\!)_{\bullet}$ and $\oplus$ is defined by

$$l \oplus a = (\succ a)*.l \succ (nil \succ a).$$

A program to compute the maximum segment sum is:

$$(\uparrow\!/ \circ +\!/*)\infty \circ tls\infty \circ [\![\succ]\!].$$

We have taken the notational liberty of combining snoc and join list catamorphisms in this program. The liberty is, however, purely notational, for the two types are isomorphic. Moreover, every join list catamorphism can be expressed as a snoc list catamorphism: in particular, the two reductions $\uparrow\!/$ and $+\!/$ are isomorphically the same as the snoc list catamorphisms $(\!|-\infty, \uparrow|\!)_{\bullet}$ and $(\!|0, +|\!)_{\bullet}$, respectively. A first step in transforming this program might be to apply corollary 3.3.5:

$$\begin{array}{ll} & (\uparrow\!/ \circ +\!/*)\infty \circ tls\infty \circ [\![\succ]\!] \\ = & \qquad \{ \triangleq tls, \text{ corollary 3.3.5} \} \\ & (\uparrow\!/ \circ +\!/*)\infty \circ [\![\oplus]\!] \circ tls \times \mathrm{I} \end{array}$$



In order to proceed further, we look to see whether we can distribute $\uparrow\!/ \circ +\!/*$ through $\oplus$, with a view to applying property 3.3.4. That is, we try to find a $\otimes$ such that

$$\uparrow\!/ \circ +\!/* \circ \oplus = \otimes \circ (\uparrow\!/ \circ +\!/*) \times \mathrm{I}.$$

$\otimes$ is found by calculating: for all $l$ and $a$,

$$\begin{array}{ll} & (\uparrow\!/ \circ +\!/*).(l \oplus a) \\ = & \qquad \{ \triangleq \oplus \} \\ & (\uparrow\!/ \circ +\!/*).((\succ a)*.l \succ (nil \succ a)) \\ = & \qquad \{ \text{evaluation of catamorphisms: see (2.33)} \} \\ & (\uparrow\!/ \circ +\!/* \circ (\succ a)*).l \uparrow a \\ = & \qquad \{ \text{map distribution} \} \\ & (\uparrow\!/ \circ (+\!/ \circ (\succ a))*).l \uparrow a \\ = & \qquad \{ \text{evaluation of catamorphisms, map distribution} \} \\ & (\uparrow\!/ \circ (+a)* \circ +\!/*).l \uparrow a \\ = & \qquad \{ + \text{ distributes over } \uparrow \} \\ & ((+a) \circ \uparrow\!/ \circ +\!/*).l \uparrow a \\ = & \qquad \{ + \text{ distributes over } \uparrow \} \\ & ((\uparrow\!/ \circ +\!/*).l \uparrow 0) + a \end{array}$$

Thus we have found $\otimes$ defined by

$$l \otimes a = (l \uparrow 0) + a$$

and we can continue our program transformation by applying property 3.3.4:

$$\begin{array}{ll} & (\uparrow\!/ \circ +\!/*)\infty \circ [\![\oplus]\!] \circ tls \times \mathrm{I} \\ = & \qquad \{ \text{property 3.3.4} \} \\ & [\![\otimes]\!] \circ (\uparrow\!/ \circ +\!/* \circ tls) \times \mathrm{I} \end{array}$$

We note in passing that since the distributivity property in the antecedent of property 3.3.4 is the same as that for snoc list promotion, we might have first used the distributivity of $\oplus$ and $\otimes$ and snoc list promotion to give

$$\uparrow\!/ \circ +\!/* \circ tls = (\!|-\infty, \otimes|\!)_{\bullet}$$

and then used corollary 3.3.5:

$$(\!|-\infty, \otimes|\!)_{\bullet}{}^{\infty} \circ [\!|\succ|\!] \;=\; [\!|\oslash|\!] \circ (\!|-\infty, \otimes|\!)_{\bullet} \times \mathrm{I}$$

to obtain the same result. In any case, assuming that $\uparrow$ and $+$ can be evaluated in constant time, it follows that $\otimes$ can be evaluated in constant time, so that the accumulation $[\!|\otimes|\!]$ represents an on-line algorithm for computing the maximal sum of segments of an infinite list.

# Chapter 4

# Relational Catamorphisms

The techniques for the construction and transformation of functions that we have considered in the previous chapters exploit very general algebraic properties of the data types concerned. These techniques are derived mainly from unique extension properties, which in turn take their shape from the underlying algebraic structure induced by the functors by which the type is defined. Our aim has been to exploit the structure that informs the type as a whole. In many programming problems, however, one is interested in the restriction of a function to a particular subtype. For example, a simple algorithm to search a tree for an occurrence of a given item is to examine each node of the tree in turn; if this algorithm is restricted to trees which are ordered, then a considerable increase in efficiency may be gained by eliminating from the search all subtrees whose elements are all greater than, or all less than, the item sought. We desire, then, some means by which we can exploit the *logical* structure of subtypes, a means by which we can effectively introduce properties into our calculations.

In order to be effective, such a mechanism should satisfy two criteria. First, it should allow properties to be formulated in such a way that they may be combined homogeneously with the notation that is used for programs; and second, it should be amenable to manipulation, so that programs and properties may be *usefully* combined. Dijkstra's *wp*-calculus is a well-attested example of an effective mechanism: the *wp*-operator provides the bridge between programs and properties, and the algebraic properties of this operator, in combination with the predicate and quantifier calculi, allow of elegant manipulations. A further connection here between programs and properties is provided by the "guards" of the guarded command language, see for example Hesselink [22,23].

In this chapter we investigate the use of guards as a means of extending the "catamorphism calculus." Guards are restrictions of the identity function to subtypes.

Their manipulability is witnessed in, among other places, Hesselink (op. cit.), and Manes and Arbib [30]. They satisfy, therefore, our second criterion. As to the first criterion, we note that catamorphisms are total functions, so the question arises: can catamorphisms be easily combined with partial functions such as guards? A positive answer is given in Backhouse's paper [2], where the notion of catamorphism is extended to incorporate relations, thereby relegating the notions of totality and partiality — and, indeed, functionality — to the status of special cases. (Mention should also be made here of the work of de Moor [14], in which, via a slightly different approach, catamorphisms on join lists are extended to relations.) The ramifications of this generalisation are far-reaching, but for the purposes of this chapter the important consequences are that relational catamorphisms enjoy a unique extension property of which that of the functional catamorphisms of the previous chapters is an instance, and that, in a relational setting, guards and catamorphisms may be homogeneously combined.

In section 4.1 we introduce the notation for, and certain properties of, the relational calculus, and we use these to express and prove an induction principle for initial Hagino types. In Martin-Löf's type theory, a type's elimination rule provides a schema for the construction of recursive functions on that type; these functions are of the same form as Meertens' paramorphisms [33]. The elimination rule also provides an induction principle for the type; so in proving an induction principle for Hagino types, we show that the Hagino paradigm of type definition parallels that of Martin-Löf's type theory.

The induction principle is also necessary for the proof of the unique extension property of relational catamorphisms. This and other properties, taken from Backhouse [2], are stated in section 4.2. Section 4.3 concludes the chapter with some examples of the use of guards.

## 4.1 Relations

In the previous chapters we have spoken blithely of types and functions between types, without imposing any interpretation on the notion of type; whether types be, for example, sets, ordered domains, or types in the sense of Martin-Löf's type theory. We have, implicitly, assumed that types can be viewed as collections of objects, but our continued emphasis on calculation has led to our concentrating on properties of functions, and we have kept objects very much in the background, often writing $0 \in \mathbb{N} \longleftarrow \mathbb{1}$ rather than $0 \in \mathbb{N}$, for example. In this chapter the notion of types as collections of objects becomes more prominent, and we adopt a somewhat more set-



theoretical tenor. We shall be concerned with subtypes, collections of objects which enjoy a given property, and we shall speak of the intersection and union of relations defined on types and subtypes. That said, our interest still lies chiefly in being able to calculate — now with relations instead of with functions — so that what is important is the algebraic properties of the operations involved, rather than their set-theoretic interpretation.

We write $R \in A \sim B$ to denote that $R$ is a relation between objects of type $A$ and objects of type $B$. If $R \in A \sim B$, then

$$a \,(\!|R|\!)\, b$$

means that $a \in A$ stands in relation $R$ to $b \in B$. Intersection, union, reverse (we write $R\cup \in B \sim A$ for the reverse of relation $R \in A \sim B$), composition and containment of relations are defined in the usual way: for all $a \in A$, $b \in B$ and $c \in C$,

$$\begin{aligned}
a \,(\!|R \cap S|\!)\, b &\equiv a \,(\!|R|\!)\, b \wedge a \,(\!|S|\!)\, b \\
a \,(\!|R \cup S|\!)\, b &\equiv a \,(\!|R|\!)\, b \vee a \,(\!|S|\!)\, b \\
b \,(\!|R\cup|\!)\, a &\equiv a \,(\!|R|\!)\, b \\
a \,(\!|R \circ T|\!)\, c &\equiv \exists(y \in B : a \,(\!|R|\!)\, y : y \,(\!|T|\!)\, c) \\
R \supseteq S &\equiv \forall(x \in A, y \in B :: x \,(\!|R|\!)\, y \Leftarrow x \,(\!|S|\!)\, y)
\end{aligned}$$

where $R, S \in A \sim B$ and $T \in B \sim C$. We shall make use of the fact that intersection, union, reverse and particularly composition are monotonic with respect to containment. A useful property of reverse is that it distributes over composition:

$$(R \circ T)\cup = T\cup \circ R\cup.$$

Note the reversal of $R$ and $T$. Furthermore, the reverse operator is its own inverse.

For all types $A$ and $B$, every function $f \in A \longleftarrow B$ induces a relation which we also denote by $f \in A \sim B$, and which is defined by: for all $a \in A$ and $b \in B$,

$$a \,(\!|f|\!)\, b \;\equiv\; a = f.b.$$

In this way we regard relations as a generalisation of functions. Note that composition of functions has become a special case of composition of relations. The equality relation on a type is simply the identity function I; obviously, I is also the unit of relational composition. The notion of functionality can be defined using the above operations:

**Definition 4.1.1** Let $R \in A \sim B$. We say that $R$ is *functional* iff

$$\mathrm{I} \supseteq R \circ R\cup.$$

Moreover, we say $R$ is *total* iff

$$R\cup \circ R \supseteq \mathrm{I}.$$

Finally, $R$ is *injective* iff $R\cup$ is functional, and $R$ is *surjective* iff $R\cup$ is total (note that $R\cup\cup = R$).
□

Functional and total relations enjoy many special properties, amongst which the following are useful:

**Property 4.1.2** If $R$ is functional, then for all $S$ and $T$,

$$S \supseteq R \circ T \quad \Leftarrow \quad R\cup \circ S \supseteq T \tag{4.1}$$

and if $R$ is total, then for all $S$ and $T$,

$$S \circ R \supseteq T \quad \Leftarrow \quad S \supseteq T \circ R\cup. \tag{4.2}$$

□

We shall use lower-case letters $f$, $g$, $h$,... for functions, whether or not they be total, and upper-case letters $R$, $S$, $T$,... for arbitrary relations.

Now, the point of considering relations is that they play an important role in the process of specifying and deriving programs. One can begin by specifying the desired relation that the program is to establish between input and output, and then "find" the program by refining this relation to such an extent that the result may be implemented on a computer: in this connection the containment ($\supseteq$) may be read as "is refined by." Alternatively, one might begin by specifying properties required of the input and of the output. Such an approach underlies the assertional programming techniques often associated with imperative programming, in which a typical task is the finding of a program $f$ which satisfies the Hoare triple

$$\{p\}\, f\, \{q\}$$

for some given predicates $p$ and $q$. The essential unity of these alternative approaches is seen in the following correspondence of relations and predicates: for every predicate $p$ on a type $A$ we shall construct a relation $p? \in A \sim A$, and for every relation $R \in A \sim A$ a predicate $R!$ on $A$ such that $p?! = p$ and, furthermore, if $\mathrm{I} \supseteq R$ then $R!? = R$. We call relations of the form $p?$ "guards":



**Definition 4.1.3 (guards)** For predicate $p$ on a type $A$, define the *guard* $p? \in A \sim A$ by:

$$a \,(\!|p?|\!)\, b \quad \equiv \quad a = b \ \wedge \ p.b.$$

□

Thus $p?$ is the restriction of the identity on $A$ to those elements satisfying $p$; or, in other words, $p?$ is the identity function on the subtype $\{x \in A | p.x\}$. This subtype is completely characterised by the predicate $p$, so we may allow ourselves the liberty of writing $p?$ for $\{x \in A \,|\, p.x\}$. Accordingly, if function $f \in B \leftarrow\!\!- A$ takes arguments satisfying predicate $p$ to results satisfying predicate $q$:

$$f \in \{y \in B \,|\, q.y\} \leftarrow\!\!- \{x \in A \,|\, p.x\},$$

then we may simply write

$$f \in q? \leftarrow\!\!- p?$$

which is equivalent to the containment

$$q? \circ f \ \supseteq \ f \circ p?.$$

We shall presently see that guards play the role of assertions in a program text, so that the above type of $f$ can also be interpreted as saying that $f$ satisfies

$$\{p\}\, f\, \{q\}.$$

The aim of establishing a correspondence between predicates and relations contained in the identity is that subtypes are then characterised by either; that is, subtypes, predicates and relations contained in the identity are in one to one correspondence with each other, so that we obtain the typing rule:

$$f \in S \leftarrow\!\!- R \quad \equiv \quad S \circ f \ \supseteq \ f \circ \mathrm{I}_A \tag{4.3}$$

where $\mathrm{I}_B \supseteq S$ and $\mathrm{I}_A \supseteq R$.

To return to the correspondence between relations and predicates, given relation $R \in A \sim A$, define the predicate $R!$ on $A$ by

$$R!.a \quad \equiv \quad a \,(\!|R|\!)\, a.$$

It is obvious that $p?! = p$; just as obvious is that

$$R!? \ = \ R \cap \mathrm{I} \tag{4.4}$$

from which it follows that if $\mathrm{I}_A \supseteq R$ then indeed $R!? = R$.

The importance of all this is that we can move freely between considering subtypes characterised by predicates and subtypes characterised by relations contained in the identity. We shall put this freedom to use by deriving an induction principle for initial Hagino types. The induction principle is somewhat unusual in that it is stated in terms of relations rather than predicates. Usually, induction is used to show that a predicate holds for all objects of a type. For the natural numbers, the induction principle is

$$\forall(x \in \mathbb{N} :: p.x) \quad \Leftarrow \quad p.0 \ \wedge \ \forall(x \in \mathbb{N} : p.x : p.(\sigma.x)).$$

In the notation just introduced, the conclusion $\forall(x \in \mathbb{N} :: p.x)$ is

$$p? \ \supseteq \ \mathrm{I}_{\mathbb{N}}.$$

As to the antecedent, the conjunct $\forall(x \in \mathbb{N} : p.x : p.(\sigma.x))$ states that the constructor $\sigma$ preserves the property $p$; that is, $\sigma \in p? \longleftarrow p?$, or, equivalently by (4.3),

$$p? \circ \sigma \ \supseteq \ \sigma \circ p?.$$

The conjunct $p.0$ can be put in a similar form. By definition 4.1.3 we have that $p.0$ is equivalent to

$$0 \ (\!|p?|\!) \ 0.$$

If we regard $0$ as a constant function of type $\mathbb{N} \longleftarrow \mathbb{1}$, this is in turn equivalent to

$$p? \circ 0 \ \supseteq \ 0.$$

The induction principle thus becomes

$$p? \ \supseteq \ \mathrm{I}_{\mathbb{N}} \quad \Leftarrow \quad (p? \circ 0 \ \supseteq \ 0) \ \wedge \ (p? \circ \sigma \ \supseteq \ \sigma \circ p?).$$

A pleasant side-effect of the use of guards here is that we have eliminated the universal quantifications with their attendant bound variables. Moreover, we obtain a more general induction principle by generalising the relation $p?$ to an arbitrary relation $R \in \mathbb{N} \sim \mathbb{N}$, which gives

$$R \supseteq \mathrm{I}_{\mathbb{N}} \quad \Leftarrow \quad (R \circ 0 \ \supseteq \ 0) \ \wedge \ (R \circ \sigma \ \supseteq \ \sigma \circ R).$$

By expressing the induction principle in this form, we see an interesting correspondence with unique extension properties: if we replace the containments by equalities, we obtain an instance of the uniqueness property of the identity catamorphism $(\!|0, \sigma|\!)$.



Our task now is to prove that this generalised induction principle follows from the unique extension property for an arbitrary initial Hagino type. In order to do so, we must first introduce Backhouse's definition of "relators" (an extension of the notion of functor, given by Backhouse in [2]). Since we shall be using guards in the proof of the induction principle, it is convenient to state here some of their basic properties: the proofs of these properties are straightforward and therefore omitted.

**Property 4.1.4 (idempotence)** $p? \circ p? \ = \ p?.$
□

**Property 4.1.5 (symmetry)** $p?^{\cup} \ = \ p?.$
□

**Property 4.1.6 (monotonicity)** $\forall(x :: p.x \Leftarrow q.x) \quad \equiv \quad p? \ \supseteq \ q?.$
□

Finally, a most useful property is the following, which has an interesting analogue in the trading law for filters (see equation (3.22) of chapter 3).

**Property 4.1.7 (precondition)** If $f$ is a function, then

$$p? \circ f \ = \ f \circ (p \circ f)?.$$

□

We are now ready to return to Hagino types. We begin by giving Backhouse's definition of relators: the definition extends the definition of functors by requiring that relators map types to types and relations to relations.

**Definition 4.1.8 (relators)** A *relator* is a mapping $\mathsf{F}$ from types to types with a corresponding action on relations, which satisfies the following properties

- for all types $A$, $A\mathsf{F}$ is a type;
- for all relations $R \in A \sim B$, there is a relation $R\mathsf{F} \in A\mathsf{F} \sim B\mathsf{F}$;
- $\mathrm{I}\mathsf{F} = \mathrm{I}$;
- $(R \circ S)\mathsf{F} \ = \ R\mathsf{F} \circ S\mathsf{F}$;

- F is monotonic: $R \supseteq S \Rightarrow RF \supseteq SF$; and
- F respects reverse: $R^{\cup}F = RF^{\cup}$.

Not every functor is a relator, but all the functors that we have used so far — the most important examples are constant functors, cartesian product and disjoint sum — are indeed also relators.
□

We need the following property, which says that relators commute with $!?$.

Lemma 4.1.9 $RF!? \supseteq R!?F$.

Proof:

$$
\begin{array}{ll}
 & RF!? \\
= & \quad \{ \text{(4.4)} \} \\
 & RF \cap \mathrm{I} \\
= & \quad \{ \text{F preserves identity} \} \\
 & RF \cap \mathrm{I}F \\
\supseteq & \quad \{ \text{F is monotonic} \} \\
 & (R \cap \mathrm{I})F \\
= & \quad \{ \text{(4.4)} \} \\
 & R!?F
\end{array}
$$

□

To conclude this section, we state the generalised induction principle for an arbitrary initial Hagino type.

**Property 4.1.10 (induction principle)** Let $\mathcal{T} = \mu(\tau : \mathsf{F})$; then for all relations $R \in \mathcal{T} \sim \mathcal{T}$:

$$R \supseteq \mathrm{I}_{\mathcal{T}} \Leftarrow R \circ \tau \supseteq \tau \circ RF.$$

**Proof:** It is sufficient to show that the condition implies that $\mathrm{I}_{\mathcal{T}} \in R!?\longleftarrow\mathcal{T}$.



$$
\begin{array}{ll}
 & \mathrm{I}_{\mathcal{T}} \in R!?\longleftarrow\mathcal{T} \\
\equiv & \quad \{ \text{property 2.1.4} \} \\
 & (\!|\tau|\!) \in R!?\longleftarrow\mathcal{T} \\
\Leftarrow & \quad \{ \text{uep for functional catamorphisms} \} \\
 & \tau \in R!?\longleftarrow R!?F \\
\equiv & \quad \{ \text{(4.3)} \} \\
 & R!? \circ \tau \supseteq \tau \circ R!?F \\
\Leftarrow & \quad \{ \text{lemma 4.1.9} \} \\
 & R!? \circ \tau \supseteq \tau \circ RF!? \\
\equiv & \quad \{ \text{(4.4)} \} \\
 & (R \cap \mathrm{I}) \circ \tau \supseteq \tau \circ (RF \cap \mathrm{I}) \\
\equiv & \quad \{ \tau \text{ is bijective, so distributes over } \cap \} \\
 & (R \circ \tau) \cap \tau \supseteq (\tau \circ RF) \cap \tau \\
\Leftarrow & \quad \{ \text{monotonicity of } \cap \} \\
 & R \circ \tau \supseteq \tau \circ RF
\end{array}
$$

□

## 4.2 Properties of Relational Catamorphisms

We summarise here the main results from Backhouse's paper, [2]. Backhouse starts from scratch, as it were, by defining relational catamorphisms as the intersection of a class of relations. This definition generalises the notion of functional catamorphisms insofar as relations are more general than functions. What makes his approach so attractive is that not only do relational catamorphisms enjoy a unique extension property identical to that of their functional counterparts, but they also allow the promotion theorem to be generalised in three ways: by considering equality and the two directions of set inclusion.

For the remainder of this section we shall consider an arbitrary Hagino type $\mathcal{T} = \mu(\tau : \mathsf{F})$. Backhouse's definition of relational catamorphisms is as follows:

**Definition 4.2.1** Given relation $R \in A \sim AF$, define the relational catamorphism $(\!|R|\!) \in A \sim \mathcal{T}$ by $(\!|R|\!) \triangleq \bigcap(T : T \circ \tau \supseteq R \circ TF : T)$.
□

**Corollary 4.2.2** For all relations $T \in A \sim \mathcal{T}$,

$$T \supseteq (\!|R|\!) \quad \Leftarrow \quad T \circ \tau \supseteq R \circ T_F.$$

□

The same notation is used for relational as for functional catamorphisms in anticipation of property 4.2.3 below, which gives a unique extension property for relational catamorphisms. It follows from the unique extension property that functional catamorphisms are special cases of relational catamorphisms. Whereas here the starting point has been functional catamorphisms and induction, it would therefore also have been possible to begin with the unique extension property for relational catamorphisms, thence deriving properties of functional catamorphisms as special cases.

For the proofs of the following five properties, the reader is referred to Backhouse [2].

**Property 4.2.3 (unique extension property)** For relation $R \in A \sim A_F$, the relational catamorphism $(\!|R|\!) \in A \sim \mathcal{T}$ enjoys the following uniqueness property. For all $T \in A \sim \mathcal{T}$,

$$T = (\!|R|\!) \quad \equiv \quad T \circ \tau = R \circ T_F.$$

□

**Corollary 4.2.4** $(\!|R|\!) \circ \tau = R \circ (\!|R|\!)_F$.
□

**Property 4.2.5 (1st Promotion Theorem)**

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

□

**Property 4.2.6 (2nd Promotion Theorem)**

$$T \circ (\!|S|\!) \supseteq (\!|R|\!) \quad \Leftarrow \quad T \circ S \supseteq R \circ T_F.$$

□



**Property 4.2.7 (3rd Promotion Theorem)**

$$T \circ (\!|S|\!) \subseteq (\!|R|\!) \quad \Leftarrow \quad T \circ S \subseteq R \circ T_F.$$

□

It follows from the unique extension property that functional catamorphisms are special cases of relational catamorphisms; it is however very important that the catamorphism operator $(\!|.|\!)$ preserves fundamental properties such as totality and functionality. The preservation of totality is proven by Backhouse in [2]; we present here a proof of the preservation of functionality and a proof that catamorphisms preserve total injections, a result that we make use of in chapter 5 below. The preservation of functionality is important since we shall be using the precondition property of guards (property 4.1.7 above) in connection with catamorphisms; recall that this property was only applicable to functions. To show that catamorphisms preserve functionality, we require the notion of weakest postspecifications, introduced by Hoare and He in [24].

**Definition 4.2.8 (weakest postspecification)** For relations $P \in C \sim A$ and $Q \in B \sim A$, the *weakest postspecification* $P/Q \in C \sim B$ is defined by the property that for all $R$,

$$P/Q \supseteq R \quad \equiv \quad P \supseteq R \circ Q$$

□

Weakest postspecifications enjoy the following cancellation property.

**Property 4.2.9** $P \supseteq P/Q \circ Q$.
**Proof:** $P/Q \supseteq P/Q$, hence by definition 4.2.8, $P \supseteq P/Q \circ Q$.
□

In the above proof and below, we let / bind more tightly than composition; we also let $\cup$ bind more tightly than /.

**Property 4.2.10 (preservation of functionality)** For all functions $f$, the catamorphism $(\!|f|\!)$ is a function.
**Proof:**

$$
\begin{array}{ll}
 & \mathrm{I} \supseteq (\!|f|\!) \circ (\!|f|\!)^{\cup} \\
\equiv & \quad \{ \text{ defn. 4.2.8 } \} \\
 & \mathrm{I}/(\!|f|\!)^{\cup} \supseteq (\!|f|\!) \\
\Leftarrow & \quad \{ \text{ corollary 4.2.2 } \} \\
 & \mathrm{I}/(\!|f|\!)^{\cup} \circ \tau \supseteq f \circ (\mathrm{I}/(\!|f|\!)^{\cup})^{\mathsf{F}} \\
\Leftarrow & \quad \{ \text{ totality of } \tau, (4.2) \} \\
 & \mathrm{I}/(\!|f|\!)^{\cup} \supseteq f \circ (\mathrm{I}/(\!|f|\!)^{\cup})^{\mathsf{F}} \circ \tau^{\cup} \\
\equiv & \quad \{ \text{ defn. 4.2.8 } \} \\
 & \mathrm{I} \supseteq f \circ (\mathrm{I}/(\!|f|\!)^{\cup})^{\mathsf{F}} \circ \tau^{\cup} \circ (\!|f|\!)^{\cup} \\
\equiv & \quad \{ \text{ property of } \cup \} \\
 & \mathrm{I} \supseteq f \circ (\mathrm{I}/(\!|f|\!)^{\cup})^{\mathsf{F}} \circ (\tau \circ (\!|f|\!))^{\cup} \\
\equiv & \quad \{ \text{ corollary 4.2.4; property of } \cup \} \\
 & \mathrm{I} \supseteq f \circ (\mathrm{I}/(\!|f|\!)^{\cup})^{\mathsf{F}} \circ (\!|f|\!)^{\mathsf{F}\cup} \circ f^{\cup} \\
\equiv & \quad \{ \mathsf{F} \text{ is a relator } \} \\
 & \mathrm{I} \supseteq f \circ (\mathrm{I}/(\!|f|\!)^{\cup} \circ (\!|f|\!)^{\cup})^{\mathsf{F}} \circ f^{\cup} \\
\Leftarrow & \quad \{ \text{ property 4.2.9; relators preserve identity } \} \\
 & \mathrm{I} \supseteq f \circ f^{\cup}
\end{array}
$$

□

A further property, one which we shall need in the next chapter, states that catamorphisms preserve total injections:

**Property 4.2.11**

$$(\!|R|\!)^{\cup} \circ (\!|R|\!) = \mathrm{I} \quad \Leftarrow \quad R^{\cup} \circ R = \mathrm{I}.$$

**Proof:**

$$
\begin{array}{ll}
 & (\!|R|\!)^{\cup} \circ (\!|R|\!) = \mathrm{I} \\
\equiv & \quad \{ \text{ corollary 2.1.4 } \} \\
 & (\!|R|\!)^{\cup} \circ (\!|R|\!) = (\!|\tau|\!) \\
\Leftarrow & \quad \{ \text{ promotion: property 4.2.5 } \} \\
 & (\!|R|\!)^{\cup} \circ R = \tau \circ (\!|R|\!)^{\mathsf{F}\cup}
\end{array}
$$



$$
\begin{array}{ll}
\equiv & \quad \{ \tau \text{ is bijective } \} \\
 & \tau^{\cup} \circ (\!|R|\!)^{\cup} \circ R = (\!|R|\!)^{\mathsf{F}\cup} \\
\equiv & \quad \{ \text{ property of } \cup \} \\
 & ((\!|R|\!) \circ \tau)^{\cup} \circ R = (\!|R|\!)^{\mathsf{F}\cup} \\
\equiv & \quad \{ \text{ corollary 4.2.4 } \} \\
 & (R \circ (\!|R|\!)^{\mathsf{F}})^{\cup} \circ R = (\!|R|\!)^{\mathsf{F}\cup} \\
\equiv & \quad \{ \text{ property of } \cup \} \\
 & (\!|R|\!)^{\mathsf{F}\cup} \circ R^{\cup} \circ R = (\!|R|\!)^{\mathsf{F}\cup} \\
\Leftarrow & \quad \{ \text{ relators, monotonicity of composition } \} \\
 & R^{\cup} \circ R = \mathrm{I}
\end{array}
$$

□

## 4.3 Examples

We give two examples of the use of guards and relational catamorphisms. The first example is concerned with constructing and transforming programs defined on subtypes: we shall attempt to exploit the logical structure of the subtype itself in the process of construction and transformation. In the second example we show that guards can be used as a means of introducing vital contextual information into calculations.

### 4.3.1 Transforming Programs on Subtypes

Subtypes in Martin-Löf's type theory are problematic. Given $a \in \{x \in A | p.x\}$, it is not, in general, possible to assert that $p.a$ holds: that information is — sometimes irretrievably — "hidden" in the subtype. The need to keep explicit all information that might later be necessary, or to explicitly pass such information as parameters, can lead to programs that are at best verbose and at worst needlessly inefficient: this problem was discussed by Chisholm and the present author in [29]. Another, related problem treated in that paper concerned the construction of recursive functions on subtypes. As mentioned earlier, recursive functions in Martin-Löf's type theory are constructed by means of an elimination rule which also encapsulates the induction principle for the type concerned. For this reason, the functions so constructed are total on the entire type; moreover, the constitution of the theory renders it difficult to reason inductively about subtypes — in particular, if the predicate $p$ which characterises the subtype

is recursively defined, one might expect it to "distribute," as it were, through the elimination rule, so that the assumption that $p$ held would be valid in each of the premises of the elimination rule. For example, if predicate $p$ on the natural numbers is such that

$$p.x \Leftarrow p.(\sigma.x)$$

then the following induction schema for $\{n \in \mathbb{N} | p.n\}$ is valid:

$$\forall(x \in \{n \in \mathbb{N} | p.n\} :: q.x) \Leftarrow (p.0 \Rightarrow q.0) \wedge \forall(x : p.(\sigma.x) \wedge q.x : q.(\sigma.x)).$$

In [29] we investigated a type of polymorphic objects and showed that this allowed "hidden" information to be distributed through elimination rules in Martin-Löf's type theory, thereby allowing such induction schemata as that above to be proven valid within the theory. This in turn facilitated the construction of recursive functions on subtypes.

A similar construction is possible in the present setting, where subtypes are identified with guards. If the predicate which characterises subtype can be expressed as a catamorphism, the calculational properties of guards and catamorphisms can be combined to manipulate functions defined on that subtype.

Consider the following genus of binary trees which consist either of a leaf or of an internal label and two subtrees:

$$A\dagger \triangleq \mu(\eta : (4\ll), ./.\backslash. : \mathsf{F})$$

where the relator $\mathsf{F}$ is defined by

$$\begin{aligned} X\mathsf{F} &= X \times A \times X \\ R\mathsf{F} &= R \times 1_A \times R \end{aligned}$$

for types $X$ and relations $R$.

Thus, for all $a \in A$, we have the leaf $\eta.a \in A\dagger$, and for all $l, r \in A\dagger$ and $m \in A$, the node $l/m\backslash r \in A\dagger$. The unique extension property for the type is as follows. For all $f \in P \longleftarrow A$ and $g \in P \longleftarrow P \times A \times P$, there is a catamorphism

$$(\!|f,g|\!) \in P \longleftarrow A\dagger$$

such that for all $h$,

$$(4.5) \qquad h = (\!|f,g|\!) \quad \equiv \quad h \circ \eta = f \ \wedge \ h \circ ./.\backslash. = g \circ h \times \mathrm{I} \times h.$$



Property 4.2.3 above tells us that this unique extension property also holds for relations and relational catamorphisms; the statement above is in terms of functions, as in the remainder of this section we shall be considering catamorphisms which are functional, though not necessarily total. From the unique extension property we obtain the rules for the evaluation of catamorphisms:

$$(4.6) \qquad (\!|f,g|\!) \circ \eta = f$$

$$(4.7) \qquad (\!|f,g|\!) \circ ./.\backslash. = g \circ (\!|f,g|\!) \times \mathrm{I} \times (\!|f,g|\!).$$

A simple example of a catamorphism on the type is the search predicate which, for given $x \in A$ and $t \in A\dagger$ determines whether $x$ occurs in $t$. For $x \in A$, we define the search predicate $in.x$ by

$$in.x \triangleq (\!|(x=), k|\!) \in Bool \longleftarrow A\dagger$$

where for all $b, c \in Bool$ and $m \in A$,

$$k.\langle b, m, c\rangle = b \vee (x = m) \vee c.$$

For $t \in A\dagger$, the evaluation of $in.x.t$ requires time linearly proportional to the number of nodes in $t$. It is well known that if the tree $t$ is ordered, there is a search algorithm which is linear in the maximum depth of the tree: we shall attempt to derive this latter algorithm by transforming the catamorphism $in.x$. We begin by specifying the subtype of ordered trees, by constructing a predicate $Ord$ which holds on ordered trees. Clearly, all leafs are ordered; i.e.,

$$(4.8) \qquad Ord \circ \eta = tt$$

where $tt$ is the constant function everywhere true. A node $l/m\backslash r$ is ordered if both subtrees $l$ and $r$ are ordered, and all labels in $l$ are strictly smaller than $m$, and all labels in $r$ are strictly greater than $m$:

$$(4.9) \qquad Ord.(l/m\backslash r) \equiv Ord.l \wedge Ord.r \wedge \forall(z : in.z.l : z < m) \wedge \forall(z : in.z.r : z > m).$$

From these recursive equations, we can express the predicate $Ord$ as a paramorphism; however, since we are interested in the subtype of ordered trees, we shall instead try to express the guard $Ord?$ as a relational catamorphism (since guards are contained in the identity, this catamorphism will be functional — though it will not be total).

From the unique extension property (4.5) we have that $Ord?$ is equal to a relational catamorphism $(\!|f, g|\!)$ iff

$$Ord? \circ \eta \;=\; f$$

and

$$Ord? \circ ./\backslash. \;=\; g \circ Ord?\times \mathrm{I} \times Ord?.$$

We can find $f$ straight away:

$$\begin{array}{ll} & Ord? \circ \eta \\ = & \quad \{ \text{ property 4.1.7 } \} \\ & \eta \circ (Ord \circ \eta)? \\ = & \quad \{ \text{ (4.8) } \} \\ & \eta \circ tt? \\ = & \quad \{ \; tt? = \mathrm{I} \; \} \\ & \eta \end{array}$$

Thus we take $f := \eta$. For the sake of brevity, define the predicate $q$ by

$$q.\langle l, m, r\rangle \;\equiv\; \forall(z : in.z.l : z < m) \;\wedge\; \forall(z : in.z.r : z > m).$$

Now, we find $g$ as follows:

$$\begin{array}{ll} & l/m\backslash r \; (\!|Ord? \circ ./\backslash.|\!) \; \langle l, m, r\rangle \\ \equiv & \quad \{ \text{ functions } \} \\ & l/m\backslash r \; (\!|Ord?|\!) \; l/m\backslash r \\ \equiv & \quad \{ \text{ definition 4.1.3 } \} \\ & Ord.(l/m\backslash r) \\ \equiv & \quad \{ \text{ (4.9), definition of } q \; \} \\ & q.\langle l, m, r\rangle \;\wedge\; Ord.l \;\wedge\; Ord.r \\ \equiv & \quad \{ \text{ functions, definition 4.1.3 } \} \\ & l/m\backslash r \; (\!|./\backslash.|\!) \; \langle l, m, r\rangle \;\wedge\; \langle l, m, r\rangle \; (\!|q?|\!) \; \langle l, m, r\rangle \;\wedge \\ & \langle l, m, r\rangle \; (\!|Ord?\times \mathrm{I} \times Ord?|\!) \; \langle l, m, r\rangle \\ \equiv & \quad \{ \text{ composition of functions } \} \\ & l/m\backslash r \; (\!|./\backslash. \circ q? \circ Ord?\times \mathrm{I} \times Ord?|\!) \; \langle l, m, r\rangle \end{array}$$



Thus we have found $g := ./\backslash. \circ q?$, and hence by the unique extension property

$$Ord? \;=\; (\!|\eta,\; ./\backslash. \circ q?|\!). \tag{4.10}$$

Our goal was to transform the function $in.x$ into an efficient search program on the subtype of ordered trees. We are seeking, therefore, a function $F$ such that $F$ is equal to $in.x$ on that subtype. That is,

$$F \circ Ord? \;=\; in.x \circ Ord?.$$

In finding this $F$, we shall use the following property:

**Property 4.3.1** For function $f$ and total $R$,

$$f \circ p? \;=\; R \circ p? \;\Leftarrow\; f \;\supseteq\; R \circ p?.$$

**Proof:** Since by monotonicity of composition and idempotency of guards,

$$f \circ p? \;\supseteq\; R \circ p? \;\Leftarrow\; f \;\supseteq\; R \circ p?$$

it suffices to show the other inclusion:

$$\begin{array}{ll} & R \circ p? \;\supseteq\; f \circ p? \\ \Leftarrow & \quad \{ \text{ monotonicity of } \circ\text{; idempotency of guards } \} \\ & R \;\supseteq\; f \circ p? \\ \Leftarrow & \quad \{ \; f \text{ is a function: (4.1) } \} \\ & f\cup \circ R \;\supseteq\; p? \\ \Leftarrow & \quad \{ \; R \text{ is total: (4.2) } \} \\ & f\cup \;\supseteq\; p? \circ R\cup \\ \equiv & \quad \{ \text{ property of } \cup\text{; symmetry of guards } \} \\ & f \;\supseteq\; R \circ p? \end{array}$$

□

The catamorphism $in.x$ is indeed total, so this property is applicable. Our goal is thus to find an $F$ satisfying

$$F \;\supseteq\; in.x \circ Ord?.$$

Our strategy will be to use promotion: in this case property 4.2.7. Our goal is then refined to the finding of a catamorphism $(\!|f,g|\!)$ such that

$$(\!|f,g|\!) \supseteq in.x \circ Ord?.$$

Using the above results and the promotion theorem to break this up into subgoals, we obtain:

$$\begin{array}{ll} & (\!|f,g|\!) \supseteq in.x \circ Ord? \\ \equiv & \quad \{\ (4.10)\ \} \\ & (\!|f,g|\!) \supseteq in.x \circ (\!|\eta, ./\backslash. \circ q?|\!) \\ \Leftarrow & \quad \{\ \text{property 4.2.7}\ \} \\ & f \supseteq in.x \circ \eta \ \wedge\ g \circ in.x \times \mathrm{I} \times in.x \supseteq in.x \circ ./\backslash. \circ q? \end{array}$$

Now, by the definition of $in.x$ and evaluation of catamorphisms (4.6), we have

$$(x=) \ =\ in.x \circ \eta$$

and so we have immediately found $f := (x=)$.

Before we calculate a suitable choice for $g$, we need to say a few words about conditional expressions. We shall use the notation

$$if(p_1 \rightarrow e_1 \,[\!]\, \cdots \,[\!]\, p_n \rightarrow e_n)$$

for the (possibly non-deterministic) program which evaluates to $e_i$ if test $p_i$ holds. We shall assume that the reader is sufficiently familiar with such constructions that we need not labour the point any further. The interested reader is referred to Hesselink [22,23] where it is shown how such conditional expressions can be modelled by using guards. The only property of conditional expressions that we shall make use of in our calculation of $g$ is this: that if the expressions $e_i$ are booleans, then the above conditional expression is equivalent to

$$(p_1 \wedge e_1) \ \vee \cdots \vee (p_n \wedge e_n).$$

This by way of preparation for the following calculation:

$$\begin{array}{ll} & in.x.(l/m\backslash r) \ \wedge\ q.\langle l,m,r\rangle \\ \equiv & \quad \{\ \text{definition of } in.x;\ (4.7)\ \} \\ & (in.x.l \ \vee\ x=m \ \vee\ in.x.r) \ \wedge\ q.\langle l,m,r\rangle \end{array}$$



$$\begin{array}{ll} \equiv & \quad \{\ \text{propositional calculus}\ \} \\ & (in.x.l \ \wedge\ q.\langle l,m,r\rangle) \ \vee\ (x=m \ \wedge\ q.\langle l,m,r\rangle) \ \vee \\ & (in.x.r \ \wedge\ q.\langle l,m,r\rangle) \\ \Rightarrow & \quad \{\ \text{definition of } q\text{, predicate calculus}\ \} \\ & (x<m \ \wedge\ in.x.l) \ \vee\ (x=m \ \wedge\ \text{true}) \ \vee\ (x>m \ \wedge\ in.x.r) \\ \equiv & \quad \{\ \text{conditionals}\ \} \\ & if(x<m \rightarrow in.x.l \,[\!]\, x=m \rightarrow \text{true} \,[\!]\, x>m \rightarrow in.x.r) \\ \equiv & \quad \{\ g \text{ defined below}\ \} \\ & (g \circ in.x \times \mathrm{I} \times in.x).\langle l,m,r\rangle \end{array}$$

In the last step we introduced a function $g \in Bool \longleftarrow Bool \times \mathbb{N} \times Bool$: a suitable definition of $g$ is for all $b,c \in Bool$ and $m \in \mathbb{N}$

$$g.\langle b,m,c\rangle \ =\ if(x<m \rightarrow b \,[\!]\, x=m \rightarrow \text{true} \,[\!]\, x>m \rightarrow c).$$

Clearly, with this definition, the last step of the above calculation is justified. What this calculation shows is that for all $y \in \mathbb{N}\!\uparrow \times \mathbb{N} \times \mathbb{N}\!\uparrow$,

$$(g \circ in.x \times \mathrm{I} \times in.x).y \quad \Leftarrow \quad (in.x \circ ./\backslash.).y \ \wedge\ q.y.$$

It is straightforward to show that this implies that

$$g \circ in.x \times \mathrm{I} \times in.x \ \supseteq\ in.x \circ ./\backslash. \circ q?$$

and so we have found a suitable $g$ and hence a catamorphism $(\!|f,g|\!)$ which is equivalent to $in.x$ on the subtype of ordered trees. Moreover, while $in.x$ in the worst case searches every node of a tree, the catamorphism that we have derived always searches in at most one subtree of every node, which means that it requires time proportional to the greatest depth of the tree.

### 4.3.2 Introducing Context

In the previous section we addressed the problem of transforming functions when these were restricted to a subtype. We found guards useful in two respects: first, that they gave an explicit formulation of the subtype to which the function was restricted, a formulation which, moreover, had the advantage of being itself functional and which could therefore be composed with the function in question; and second, that they

lent themselves to algebraic manipulation, above all when combined with catamorphisms. In this section we shall show how guards can be used to introduce contextual information into calculations. The example we give comprises part of a solution to the longest ascending segment problem, which comes from Dijkstra and Feijen [15]. In [1], Backhouse shows that segment problems can be solved in linear time if they can be expressed in a certain form, and if a certain distributivity property obtains. Given this distributivity property, a form of Horner's rule can be applied to yield a linear function which solves the segment problem. The longest ascending segment problem can be expressed in the required form, but the required distributivity property obtains only in the range of a given function. This is, in fact, sufficient, but we need some means of including this contextual information in our calculations in order that Horner's rule be applicable. We shall use guards to specify the contextual information, and then use the algebraic properties of guards to propagate this information in such a way that we make use of the distributivity property.

The problem is stated in terms of non-empty snoc lists: we begin with a brief revision of the type structure.

**Definition 4.3.2 (non-empty snoc lists)** For a type $A$, the type of non-empty snoc lists over $A$ is defined by

$$A+ \;\triangleq\; \mu(\eta : (A\ll), \succ\!+ : (\times A)).$$

The type therefore has two constructors: the singleton constructor $\eta \in A+\longleftarrow A$, and the cons operator $\succ\!+ \in A+\longleftarrow A+\times A$. The unique extension property states that for every $f \in B\longleftarrow A$ and $\oplus \in B\longleftarrow B\times A$, there is a catamorphism $(\!|f,\oplus|\!) \in B\longleftarrow A+$ such that for all $h$,

$$h = (\!|f,\oplus|\!) \;\equiv\; h\circ\eta = f \;\wedge\; h\circ\succ\!+ = \oplus\circ h\times \mathrm{I}$$

from which we obtain the evaluation rules for catamorphisms:

$$(\!|f,\oplus|\!)\circ\eta = f \tag{4.11}$$

$$(\!|f,\oplus|\!)\circ\succ\!+ = \oplus\circ(\!|f,\oplus|\!)\times \mathrm{I}. \tag{4.12}$$

□

The promotion theorem for the type is:



**Theorem 4.3.3 (promotion)**

$$g\circ(\!|f,\oplus|\!) = (\!|g\circ f,\otimes|\!) \;\Leftarrow\; g\circ\oplus = \otimes\circ g\times \mathrm{I}.$$

□

Again, we have stated the unique extension property and the promotion theorem in terms of functions, since in this section we consider only (not necessarily total) functions.

The map function is defined in the usual way (see definition 2.3.1 of chapter 2):

**Definition 4.3.4 (map)** For $f \in B\longleftarrow A$, define

$$f+ \;\triangleq\; (\!|\eta\circ f, \succ\!+\circ \mathrm{I}\times f|\!) \in B+\longleftarrow A+.$$

□

Although the type is not free in the sense of section 2.3.2 of chapter 2, it is possible to construct catamorphisms that closely resemble reductions (in fact, these are reductions in the sense of Verwer [40]): for the purposes of this section, we shall refer to such catamorphisms also as reductions, and use the same notation.

**Definition 4.3.5 (reduction)** For $\oplus \in A\longleftarrow A\times A$, define

$$\oplus/ \;\triangleq\; (\!|\mathrm{I},\oplus|\!) \in A\longleftarrow A+.$$

□

Of the properties of maps given in section 2.3.1 of chapter 2, we shall need the following:

**Property 4.3.6** $\oplus/\circ f+ = (\!|f, \oplus\circ \mathrm{I}\times f|\!)$.
□

And map distribution:

**Property 4.3.7** $f+\circ g+ = (f\circ g)+$.
□

In solving the longest ascending segment problem, the following subgoal occurs.

$$(4.13) \qquad \Uparrow\!/ \circ F{+} \circ tls \;=\; (\!|\diamond 1, \oslash|\!) \;\in\; \mathbb{Z}{\times}\mathbb{N} \longleftarrow \mathbb{Z}{+}.$$

where, for $m, m' \in \mathbb{Z}$, $l, l' \in \mathbb{N}$, $x \in \mathbb{Z}{+}$ and $y \in \mathbb{Z}{\times}\mathbb{N}$:

$$\begin{aligned}
(m,l) \Uparrow (m',l') &= \text{if } l \geq l' \text{ then } (m,l) \text{ else } (m',l') \text{ fi} \\
F &= (\!|\diamond 1, \otimes|\!) \\
(\diamond 1).m &= (m,1) \\
(m,l) \otimes m' &= (m', \text{if } m \leq m' \text{ then } l+1 \text{ else } 1 \text{ fi}) \\
tls &= (\!|\eta \circ \eta, \odot|\!) \\
x \odot m &= ((>\!\!\rightarrow m){+}.x) \rightarrowtail \eta.m \\
y \otimes m &= (y \otimes m) \Uparrow (m,1)
\end{aligned}$$

The difficulty in proving equation (4.13) is that we require that $\otimes$ distributes backwards through $\Uparrow$; in that case the equality follows by Horner's rule, which is the essence of the "aggregated segment sum" theorem of Backhouse [1]. However, the distributivity property does not hold in general. In an investigation of the aggregated segment sum theorem, Zwiggelaar [42] finds several examples of segment problems, among which the longest ascending segment problem, in which the distributivity property required to apply the theorem is similarly frustrated. In each of these examples, the distributivity property does in fact hold, but only in certain contexts: in the present case, the crucial observation made by Zwiggelaar is that distributivity holds when the first components of the pairs are equal, i.e., for all $m$, $l$, $l'$ and $a$:

$$(4.14) \qquad ((m,l) \Uparrow (m,l')) \otimes a \;=\; ((m,l) \otimes a) \Uparrow ((m,l') \otimes a).$$

Now it is the case that in the lists of pairs in the range of the function $F{+} \circ tls$ all the first components are equal, and Zwiggelaar's inductive proof of (4.13) makes use of this property. The remainder of this section is concerned with the construction of a calculational proof in which we can make use of the fact that we are working in the context of the range of the above function in order to apply promotion, and so avoid having recourse to induction.

We begin by deriving some lemmas on conditional distributivity.

**Definition 4.3.8 (invariance)** We say that predicate $p$ is $\oplus$-*invariant* if for all $x$ and $y$, $p.(x \oplus y) \Leftarrow p.x \wedge p.y$. This implication is equivalent to the equation:

$$p? \circ \oplus \circ p?{\times}p? \;=\; \oplus \circ p?{\times}p?.$$

□



The reader can easily check that equality of first components is $\Uparrow$-invariant: for any $m$, construct the predicate $(=\!m \circ \ll)$ and we have for all $x$ and $y$:

$$(4.15) \qquad \ll.(x \Uparrow y) = m \;\Leftarrow\; \ll.x = m \wedge \ll.y = m.$$

**Property 4.3.9** If $p$ is $\oplus$-invariant, then

$$\oplus\!/ \circ (p?){+} \;=\; p? \circ (\!|p?, \oplus \circ p?{\times}p?|\!).$$

**Proof:**

$$\begin{array}{ll}
 & \oplus\!/ \circ (p?){+} \;=\; p? \circ (\!|p?, \oplus \circ p?{\times}p?|\!) \\
\equiv & \quad \{ \text{ property 4.3.6 } \} \\
 & (\!|p?, \oplus \circ \mathrm{I}{\times}p?|\!) \;=\; p? \circ (\!|p?, \oplus \circ p?{\times}p?|\!) \\
\Leftarrow & \quad \{ \text{ promotion; idempotency of guards } \} \\
 & p? \circ \oplus \circ p?{\times}p? \;=\; \oplus \circ \mathrm{I}{\times}p? \circ p?{\times}\mathrm{I} \\
\equiv & \quad \{ \text{ relators; defn. 4.3.8 } \} \\
 & p \text{ is } \oplus\text{-invariant}
\end{array}$$

□

**Property 4.3.10 (conditional distributivity)** Suppose $p$ is $\oplus$-invariant and $f \in A \longleftarrow A$ distributes over $\oplus$ on condition $p$:

$$f \circ \oplus \circ p?{\times}p? \;=\; \oplus \circ f{\times}f \circ p?{\times}p?$$

(or, equivalently, $f.(x \oplus y) \;=\; f.x \oplus f.y \;\Leftarrow\; p.x \wedge p.y$); then

$$f \circ \oplus\!/ \circ (p?){+} \;=\; \oplus\!/ \circ f{+} \circ (p?){+}.$$

**Proof:**

$$\begin{array}{ll}
 & f \circ \oplus\!/ \circ (p?){+} \;=\; \oplus\!/ \circ f{+} \circ (p?){+} \\
\equiv & \quad \{ \text{ property 4.3.9; properties 4.3.7, 4.3.6 } \} \\
 & f \circ p? \circ (\!|p?, \oplus \circ p?{\times}p?|\!) \;=\; (\!|f \circ p?, \oplus \circ \mathrm{I}{\times}(f \circ p?)|\!) \\
\Leftarrow & \quad \{ \text{ promotion; idempotency of guards } \} \\
 & f \circ p? \circ \oplus \circ p?{\times}p? \;=\; \oplus \circ \mathrm{I}{\times}(f \circ p?) \circ (f \circ p?){\times}\mathrm{I} \\
\equiv & \quad \{ \text{ relators; } p \text{ is } \oplus\text{-invariant } \} \\
 & f \circ \oplus \circ p?{\times}p? \;=\; \oplus \circ f{\times}f \circ p?{\times}p?
\end{array}$$

□

With reference to the problem in hand, (4.14) gives the following conditional distribution.

$$\begin{array}{l} (\circledcirc a) \circ \Uparrow \circ (=m \circ \ll)?\times(=m \circ \ll)? \\ = \\ \Uparrow \circ (\circledcirc a)\times(\circledcirc a) \circ (=m \circ \ll)?\times(=m \circ \ll)? \end{array}$$

and we already have the invariance property (4.15), so property 4.3.10 gives:

$$(4.16) \qquad (\circledcirc a) \circ \Uparrow\!\!/ \circ (=m \circ \ll)?+ \;=\; \Uparrow\!\!/ \circ (\circledcirc a)+ \circ (=m \circ \ll)?+$$

We return now to the proof of (4.13): in fact, we shall use promotion to prove:

$$(4.17) \qquad \Uparrow\!\!/ \circ F+ \circ tls \circ (\!|\eta, \succ\!+|\!) = (\!|\diamond 1, \oslash|\!).$$

This is equivalent to (4.13) since $(\!|\eta, \succ\!+|\!)$ is the identity catamorphism, by property 2.1.4 of chapter 2. Promotion then yields the following subgoals:

$$(4.18) \qquad \Uparrow\!\!/ \circ F+ \circ tls \circ \eta \;=\; \diamond 1$$

$$(4.19) \qquad \Uparrow\!\!/ \circ F+ \circ tls \circ \succ\!+ \;=\; \oslash \circ (\Uparrow\!\!/ \circ F+ \circ tls)\times \mathrm{I}$$

The former is straightforward calculation using (4.11); we concentrate on the latter. We shall need the following lemmas. For predicate $p$, the predicate $\wedge\!\!/ \circ p+$ tests whether all elements of a given list satisfy $p$.

**Lemma 4.3.11** $(p?)+ \;=\; \wedge\!\!/ \circ p+.$

**Proof:** Straightforward application of the unique extension property.
□

We introduce the notation $l_x$ for $\gg\!/.x$:

**Lemma 4.3.12** For all $x \in A+$,

$$\wedge\!\!/ \circ (=l_x \circ \gg\!/)+ \circ tls \;=\; (=l_x) \circ \gg\!/.$$

**Proof:** Using the unique extension property, the right-hand side can be expressed as a catamorphism; the equality then follows by a straightforward application of promotion.
□



**Lemma 4.3.13** For all $x \in A+$,

$$tls \circ (=x)? \;=\; ((=l_x \circ \gg\!/)?)+ \circ tls \circ (=x)?.$$

**Proof:**

$$\begin{array}{ll} & ((=l_x \circ \gg\!/)?)+ \circ tls \\ = & \quad \{ \text{ lemma 4.3.11 } \} \\ & (\wedge\!\!/ \circ (=l_x \circ \gg\!/)+)? \circ tls \\ = & \quad \{ \text{ property 4.1.7 } \} \\ & tls \circ (\wedge\!\!/ \circ (=l_x \circ \gg\!/)+ \circ tls)? \\ = & \quad \{ \text{ lemma 4.3.12 } \} \\ & tls \circ (=l_x \circ \gg\!/)? \\ \supseteq & \quad \{ \; l_x = l_y \Leftarrow x = y; \text{ property 4.1.6 } \} \\ & tls \circ (=x)? \end{array}$$

By property 4.3.1, the inclusion proved by this calculation is sufficient to establish the equality.
□

**Lemma 4.3.14** $\ll \circ F \;=\; \gg\!/.$

**Proof:** Straightforward, using the definition of $F$ and promotion.
□

Finally, the proof of (4.19) is given below: we use guards to introduce contextual information, and then use the lemmas above to push this information leftwards through the expression until we can apply conditional distributivity. The guards which are so propagated always hold in their particular context, playing the role of comments in a program text: since the guards always hold, they can simply be omitted when no longer required.

$$\begin{array}{ll} & (\Uparrow\!\!/ \circ F+ \circ tls).(x \succ\!+ a) \\ = & \quad \{ \text{ evaluation, using (4.12) and } \triangleq \odot \; \} \\ & (\Uparrow\!\!/ \circ (F \circ (\succ\!+a))+ \circ tls).x \;\Uparrow\; (a, 1) \\ = & \quad \{ \text{ (4.12), } \triangleq F\text{; introduce context: } x = x \; \} \end{array}$$

$$
\begin{array}{ll}
 & (\Uparrow/ \circ (\circledcirc a)+ \circ F+ \circ tls \circ (=x)?).x \ \Uparrow \ (a,1) \\
= & \quad \{ \text{ lemma 4.3.13; property 4.3.7 } \} \\
 & (\Uparrow/ \circ (\circledcirc a)+ \circ (F \circ (=l_x \circ \gg/)?)+ \circ tls).x \ \Uparrow \ (a,1) \\
= & \quad \{ \text{ lemma 4.3.14 } \} \\
 & (\Uparrow/ \circ (\circledcirc a)+ \circ (F \circ (=l_x \circ \ll \circ F)?)+ \circ tls).x \ \Uparrow \ (a,1) \\
= & \quad \{ \text{ properties 4.1.7 and 4.3.7 } \} \\
 & (\Uparrow/ \circ (\circledcirc a)+ \circ (=l_x \circ \ll)?+ \circ F+ \circ tls).x \ \Uparrow \ (a,1) \\
= & \quad \{ (4.16) \} \\
 & ((\circledcirc a) \circ \Uparrow/ \circ F+ \circ tls).x \ \Uparrow \ (a,1) \\
= & \quad \{ \triangleq \circledcirc \} \\
 & (\Uparrow/ \circ F+ \circ tls).x \circledcirc a
\end{array}
$$

# Chapter 5

# Fixed Points of Functors

We have deferred until now any discussion of the existence of Hagino types. This is not because the topic is unimportant, but rather because we wished to emphasise in the foregoing chapters the effectiveness with which Hagino types can be used. Of course, there is little point in dilating upon such effectiveness if Hagino types can only be shown to exist under severely restricted conditions: it turns out, however, that the conditions on existence are very mild. In this chapter we give a brief account of these conditions — essentially, that the existence of a Hagino type is guaranteed if the functors by which the type is defined are, in a certain sense, continuous — and we prove that functors induced by parameterised Hagino types are continuous if their defining functors are continuous.

The types $\mu(\tau : \mathsf{F})$ and $\nu(\delta : \mathsf{F})$ can be constructed as the least and greatest fixed points respectively of the functor $\mathsf{F}$. The literature on fixed points is extensive, but Bos and Hemerik [11] and Manes and Arbib [30] give particularly lucid accounts. It is well known from domain theory that continuous functions have fixed points; the thrust of the two works mentioned above is that the notions of domain and continuity admit of a very elegant generalisation in category theory. For us, the beauty and economy of this generalisation reinforce the argument that the applications of category theory to computing science deserve further study. Since our aim here is to give merely a brief summary of this generalisation, a summary, moreover, tailored to fit our present purposes, the interested reader is urged to consult one or both of the above-mentioned works.

Briefly, then, the notion of ordering on a domain is subsumed by the notion of function space. That is, the existence of a function $A \longleftarrow B$ replaces the statement that $A \sqsupseteq B$; what is surprising is that we need not require that the function of type $A \longleftarrow B$ be injective. Accordingly, the domain theoretical notion of a least element

(usually denoted by "$\perp$") is replaced by the notion of initial object: a type $\perp$ such that for all types $A$ there is exactly one function (usually denoted by "!") in $A \longleftarrow \perp$ (if we regard types as sets, the empty set $\emptyset$ is an initial object; if we take Hagino types as starting point, then $\mu(\tau : \mathsf{I})$ is an initial object). The domain theoretic notion of an ascending chain of objects $d_i$,

$$\cdots \sqsupseteq d_3 \sqsupseteq d_2 \sqsupseteq d_1 \sqsupseteq d_0,$$

becomes a composition of functions $d_i \in D_{i+1} \longleftarrow D_i$ (also called a chain):

$$\cdots \xleftarrow{d_3} D_3 \xleftarrow{d_2} D_2 \xleftarrow{d_1} D_1 \xleftarrow{d_0} D_0.$$

The family of functions $d_i \in D_{i+1} \longleftarrow D_i$ is indexed by $i \in \mathbb{N}$, so we are not considering arbitrary chains (the standard terminology is "$\mathbb{N}$-chain" or "$\omega$-chain," "$\omega$-continuity," and so forth; we shall simply refer to "chain," "continuity," etc.).

The notion of upper bound is generalised as follows. An upper bound of the above chain is a pair $(A, \alpha)$, where $A$ is a type and $\alpha$ is a family of functions $\alpha_i \in A \longleftarrow D_i$ such that for all $i$,

$$\alpha_{i+1} \circ d_i = \alpha_i.$$

Recall that the existence of the functions $\alpha_i \in A \longleftarrow D_i$ is to be interpreted as the statement that $A \sqsupseteq D_i$ for each $i$. Correspondingly, a least upper bound is an upper bound $(A, \alpha)$ such that for all other upper bounds $(B, \beta)$ there is a unique function $f \in B \longleftarrow A$ (i.e., $B \sqsupseteq A$) such that for all $i$,

$$f \circ \alpha_i = \beta_i.$$

The notion corresponding to continuity is referred to as *co*continuity (the terminology is unfortunate, but well-established). A functor is cocontinuous if it distributes over least upper bounds; that is, $\mathsf{F}$ is cocontinuous if whenever $(A, \alpha)$ is the least upper bound of the chain

$$\cdots \xleftarrow{d_3} D_3 \xleftarrow{d_2} D_2 \xleftarrow{d_1} D_1 \xleftarrow{d_0} D_0,$$

then $(A\mathsf{F}, \alpha\mathsf{F})$ is the least upper bound of the chain

$$\cdots \xleftarrow{d_3\mathsf{F}} D_3\mathsf{F} \xleftarrow{d_2\mathsf{F}} D_2\mathsf{F} \xleftarrow{d_1\mathsf{F}} D_1\mathsf{F} \xleftarrow{d_0\mathsf{F}} D_0\mathsf{F}.$$

This notion of cocontinuity is sufficient to establish that, given an initial object $\perp$, and given that every chain has a least upper bound, if $\mathsf{F}$ is cocontinuous, then the type $\mu(\tau : \mathsf{F})$ exists, and can be constructed as the least upper bound of

$$\cdots \xleftarrow{!\mathsf{FFF}} \perp\mathsf{FFF} \xleftarrow{!\mathsf{FF}} \perp\mathsf{FF} \xleftarrow{!\mathsf{F}} \perp\mathsf{F} \xleftarrow{!} \perp.$$



The assumptions that there is an initial object and that every chain has a least upper bound correspond to the domain theoretical assumptions that the domain contains a least element, and that every ascending chain has a least upper bound. The proof that these conditions are sufficient to establish the existence of initial Hagino types originates with Smyth and Plotkin [39], and can also be found in Bos and Hemerik, and in Manes and Arbib (op. cit.). Manes and Arbib also address the question of the existence of types with congruences, such as the type of join lists.

The prefix "co-" in cocontinuity suggests that this construction can be dualised, giving the dual notions of terminal object, descending chains, (greatest) lower bounds and continuity of functors (these notions are defined in section 5.2 below). The theorem then becomes that if there is a terminal object, and if every descending chain has a greatest lower bound, then the type $\nu(\delta : \mathsf{F})$ exists. We shall refer to functors that are both continuous and cocontinuous as "bicontinuous."

The conditions on these theorems (existence of initial and terminal objects, and of least upper bounds and greatest lower bounds) are not too restrictive. When we consider types to be sets, the initial object is the empty set, the terminal object is any one-point set, and least upper bound and greatest lower bounds are, respectively, inductive and projective limits. An important result is that in the category of sets all polynomial functors are bicontinuous — a polynomial functor is a functor constructed according to the following rules:

- the identity functor and constant functors are polynomial functors;
- if $\mathsf{F}$ and $\mathsf{G}$ are polynomial functors, so too is their composition $\mathsf{FG}$;
- if $\mathsf{F}$ and $\mathsf{G}$ are polynomial functors, so too is their product, $\mathsf{F} \hat{\times} \mathsf{G}$, and their sum, $\mathsf{F} \hat{+} \mathsf{G}$.

(This result holds not only in the category of sets: Manes and Arbib, op. cit., give more examples of interesting categories in which polynomial functors are bicontinuous.) For example, the functor which defines the type of snoc lists, is polynomial — it can be written as $(\mathbb{1}\ll)\hat{+}(\mathsf{I}\hat{\times}(A\ll))$ — and so both its least and greatest fixed points exist (its least fixed point is the type of snoc lists; its greatest fixed point is a type that includes both finite and infinite lists).

Many of the Hagino types in the previous chapters were defined by polynomial functors; some, however, such as the type of rose trees (section 2.3.3), were not. The functor which defines the type of rose trees, $(A\ll)\hat{+}*$, uses the functor $*$, which was constructed from a parameterised Hagino type, the type of snoc lists. Similarly, the

type of infinite multiway trees of section 3.2 is defined by the functor $(A\ll)\widehat{\times}*$. The question then arises whether the functor $*$ is bicontinuous. One might ask, indeed, whether bicontinuity is enjoyed by the class of functors obtained by adding to the definition of polynomial functors the clause:

- if for all types $A$, the functor $(A\otimes)$ is polynomial, so too are the functors $\varpi$ and $\epsilon$ defined by: for all $A$, $A\varpi \triangleq \mu(\tau : (A\otimes))$ and $A\epsilon \triangleq \nu(\delta : (A\otimes))$.

In fact, we shall prove that if $\otimes$ is bicontinuous, then $\varpi$ is bicontinuous, and $\epsilon$ is continuous. Section 5.1 below contains the proof that $\varpi$ is cocontinuous — from which we obtain the dual statement that $\epsilon$ is continuous — and section 5.2 shows that $\varpi$ is continuous, though for this latter result we require further that $\otimes$ be a relator.

## 5.1 Cocontinuity

We begin by giving explicitly the definitions given informally above.

**Definition 5.1.1 (ascending chain)** An *ascending chain* is a family of functions $d_i \in D_{i+1} \longleftarrow D_i$ for $i \in \mathbb{N}$. We shall use the notation $(d : D)$ for such a family.
□

Corresponding to the domain theoretical notion of upper bound is that of cocone.

**Definition 5.1.2 (cocone)** A *cocone* of an ascending chain $(d:D)$ is a pair $(A, \alpha)$, where $A$ is an type and $\alpha$ is a family of functions $\alpha_i \in A \longleftarrow D_i$ for $i \in \mathbb{N}$ with the property that for all $i$,

$$\alpha_{i+1} \circ d_i = \alpha_i.$$

□

The notion corresponding to least upper bound is "colimit," characterised by a unique extension property:

**Definition 5.1.3 (colimit)** A *colimit* of an ascending chain $(d : D)$ is a cocone $(A, \alpha)$ with the property that for all cocones $(B, \beta)$ of $(d : D)$, there is a function $\beta/\alpha : B \longleftarrow A$, with the following uniqueness property: for all $f \in B \longleftarrow A$,

$$\beta/\alpha = f \quad \equiv \quad \forall(i :: \beta_i = f \circ \alpha_i). \tag{5.1}$$

□



In category-theoretical terminology, the colimit $(A, \alpha)$ is initial in the category of cocones of the chain $(d : D)$. The initiality of Hagino types allowed us to derive many useful properties; the same is true of the initiality of colimits. taking $f := \beta/\alpha$, the left-hand side of (5.1) becomes true, yielding:

$$\forall(i :: \beta_i = \beta/\alpha \circ \alpha_i). \tag{5.2}$$

(This "cancellation" property is the reason that we have chosen the notation $\beta/\alpha$; it resembles the weakest postspecification of Hoare and He [24] and the "over" operator of the Lambek calculus [26], for which similar cancelation properties obtain.) Another corollary of initiality is the following, which is obtained from the initiality of colimits in just the same way as the identity catamorphism law (corollary 2.1.4) is obtained from the initiality of Hagino types.

**Property 5.1.4** For all colimits $(A, \alpha)$, $\alpha/\alpha = \mathrm{I}_A$.
**Proof:** Since all colimits are cocones, there is a function $\alpha/\alpha \in A \longleftarrow A$. The equality is proven by:

$$\begin{array}{ll} & \alpha/\alpha = \mathrm{I}_A \\ \equiv & \quad \{ \text{ (5.1), with } \beta := \alpha \text{ and } f := \mathrm{I}_A \ \} \\ & \forall(i :: \alpha_i = \mathrm{I}_A \circ \alpha_i) \\ \equiv & \quad \{ \text{ identity } \} \\ & \text{true} \end{array}$$

□

As with catamorphisms and zygomorphisms, the existence of a uniqueness property suggests that we look for a promotion property. We find:

**Property 5.1.5 (colimit promotion)** For all ascending chains $(d:D)$ with colimit $(A, \alpha)$ and cocone $(C, \gamma)$, and for all functions $f : B \longleftarrow C$ and families of functions $\beta_i : B \longleftarrow D_i$,

$$\beta/\alpha = f \circ \gamma/\alpha \quad \Leftarrow \quad \forall(i :: \beta_i = f \circ \gamma_i).$$

**Proof:** We first show that $\beta/\alpha$ is well-defined under the assumption that the right-hand side holds; i.e., we show that $(B, \beta)$ is a cocone of $(d : D)$. This in turn follows by definition 5.1.2 from the following equality:

$$
\begin{array}{ll}
 & \beta_{i+1} \circ d_i \\
= & \quad \{ \text{ assuming the right-hand side } \} \\
 & f \circ \gamma_{i+1} \circ d_i \\
= & \quad \{ (C,\gamma) \text{ is a cocone } \} \\
 & f \circ \gamma_i \\
= & \quad \{ \text{ assuming the right-hand side } \} \\
 & \beta_i
\end{array}
$$

Hence $\beta/\alpha$ is well-defined, so we can complete the proof:

$$
\begin{array}{ll}
 & \beta/\alpha \;=\; f \circ \gamma/\alpha \\
\equiv & \quad \{ \text{ unique extension: (5.1), with } f := f \circ \gamma/\alpha \} \\
 & \forall(i :: \beta_i \;=\; f \circ \gamma/\alpha \circ \alpha_i) \\
\equiv & \quad \{ (5.2) \} \\
 & \forall(i :: \beta_i \;=\; f \circ \gamma_i)
\end{array}
$$

□

Before proving the main theorem, we give the definitions of cocontinuity for functors of one and of two arguments.

**Definition 5.1.6 (cocontinuous)** A functor $F$ is *cocontinuous* if for all colimits $(A,\alpha)$ of ascending chains $(d : D)$, $(AF, \alpha F)$ is the colimit of the ascending chain $(dF : DF)$.
□

**Definition 5.1.7 (cocontinuous)** A binary functor $\otimes$ is *cocontinuous* if for all colimits $(A,\alpha)$ of $(d : D)$ and $(B,\beta)$ of $(e : E)$, $(A \otimes B, \alpha \otimes \beta)$ is the colimit of $(d \otimes e : D \otimes E)$.
□

The following theorem assumes a category in which all ascending chains have colimits.

**Theorem 5.1.8 (cocontinuity of defined functors)** Let the functor $\varpi$ be defined by $A\varpi \triangleq \mu(\tau : A\otimes)$, where the bifunctor $\otimes$ is cocontinuous. Then $\varpi$ is cocontinuous.



**Proof:** Let $(A,\alpha)$ be the colimit of $(d : D)$ and $(B,\beta)$ the colimit of $(d\varpi : D\varpi)$. It suffices to show that $(A\varpi, \alpha\varpi)$ is isomorphic to $(B,\beta)$. We first construct morphisms $(A\varpi, \alpha\varpi) \longleftarrow (B,\beta)$ and $(B,\beta) \longleftarrow (A\varpi, \alpha\varpi)$ and then show that these constitute an isomorphism.

(a) Construction of $(A\varpi, \alpha\varpi) \longleftarrow (B,\beta)$.

Since $\varpi$ is a functor, $(A\varpi, \alpha\varpi)$ is a cocone of $(d\varpi : D\varpi)$. Since $(B,\beta)$ is the colimit of this chain, there is a unique function $\alpha\varpi/\beta \in A\varpi \longleftarrow B$ satisfying for all $i$

$$\alpha_i\varpi \;=\; \alpha\varpi/\beta \circ \beta_i. \tag{5.3}$$

(b) Construction of $(B,\beta) \longleftarrow (A\varpi, \alpha\varpi)$.

From the unique extension property for $A\varpi$, we know that we can construct a catamorphism $B \longleftarrow A\varpi$ from a function $B \longleftarrow A \otimes B$. Let us examine, then, the object $A \otimes B$. Since $\otimes$ is cocontinuous, $(A \otimes B, \alpha \otimes \beta)$ is the colimit of $(d \otimes d\varpi : D \otimes D\varpi)$ — but $(B, \beta \circ \tau)$ is a cocone of this chain because for all $i \in \mathbb{N}$,

$$
\begin{array}{ll}
 & \beta_{i+1} \circ \tau \circ d_i \otimes d_i\varpi \\
= & \quad \{ \text{ evaluation of maps: (2.16) } \} \\
 & \beta_{i+1} \circ d_i\varpi \circ \tau \\
= & \quad \{ (B,\beta) \text{ is cocone of } (d\varpi : D\varpi) \} \\
 & \beta_i \circ \tau
\end{array}
$$

Hence, by the colimit property of $(A \otimes B, \alpha \otimes \beta)$, definition 5.1.3, there is a unique function $(\beta \circ \tau)/(\alpha \otimes \beta) \in B \longleftarrow A \otimes B$ which satisfies for all $i$:

$$\beta_i \circ \tau \;=\; (\beta \circ \tau)/(\alpha \otimes \beta) \circ \alpha_i \otimes \beta_i \tag{5.4}$$

Thus we have constructed the desired function $(\!|(\beta \circ \tau)/(\alpha \otimes \beta)|\!) \in B \longleftarrow A\varpi$. Incidentally, we remark that equation (5.4) is a promotion property so that we can calculate:

$$
\begin{array}{ll}
 & \beta_i \\
= & \quad \{ (5.4), \text{ uep for catamorphisms } \} \\
 & (\!|(\beta \circ \tau)/(\alpha \otimes \beta) \circ \alpha_i \otimes \mathrm{I}|\!) \\
= & \quad \{ \text{ property 2.3.3 } \} \\
 & (\!|(\beta \circ \tau)/(\alpha \otimes \beta)|\!) \circ \alpha_i\varpi
\end{array}
$$

which proves

(5.5) $$\beta_i = (\!|(\beta\circ\tau)/(\alpha\otimes\beta)|\!) \circ \alpha_i\varpi.$$

**Proof of isomorphism.**

(a) $(\!|(\beta\circ\tau)/(\alpha\otimes\beta)|\!) \circ \alpha\varpi/\beta = \mathrm{I}_B$.

$$\begin{array}{ll} & (\!|(\beta\circ\tau)/(\alpha\otimes\beta)|\!) \circ \alpha\varpi/\beta = \mathrm{I}_B \\ \equiv & \quad \{ \text{ property 5.1.4 } \} \\ & (\!|(\beta\circ\tau)/(\alpha\otimes\beta)|\!) \circ \alpha\varpi/\beta = \beta/\beta \\ \Leftarrow & \quad \{ \text{ property 5.1.5 } \} \\ & \forall(i :: \beta_i = (\!|(\beta\circ\tau)/(\alpha\otimes\beta)|\!) \circ \alpha_i\varpi) \\ \equiv & \quad \{ (5.5) \} \\ & \text{true} \end{array}$$

(b) $\alpha\varpi/\beta \circ (\!|(\beta\circ\tau)/(\alpha\otimes\beta)|\!) = \mathrm{I}_{A\varpi}$.

We anticipate using $\varpi$-promotion in the proof of this equality, so we note that

$$\begin{array}{ll} & \alpha\varpi/\beta \circ \beta_i \circ \tau \\ = & \quad \{ (5.3) \} \\ & \alpha_i\varpi \circ \tau \\ = & \quad \{ \text{ evaluation of maps: (2.16) } \} \\ & \tau \circ \alpha_i\otimes\alpha_i\varpi \end{array}$$

and hence by property 5.1.5, with $\gamma := \tau \circ \alpha\otimes\alpha\varpi$, $\alpha := \alpha\otimes\beta$, $f := \alpha\varpi/\beta$ and $\beta := \beta\circ\tau$:

(5.6) $$\alpha\varpi/\beta \circ (\beta\circ\tau)/(\alpha\otimes\beta) = (\tau \circ \alpha\otimes\alpha\varpi)/(\alpha\otimes\beta).$$

And the proof is completed as follows.

$$\begin{array}{ll} & \alpha\varpi/\beta \circ (\!|(\beta\circ\tau)/(\alpha\otimes\beta)|\!) = \mathrm{I}_{A\varpi} \\ \equiv & \quad \{ \text{ identity } \} \\ & \alpha\varpi/\beta \circ (\!|(\beta\circ\tau)/(\alpha\otimes\beta)|\!) = (\!|\tau|\!) \\ \Leftarrow & \quad \{ \varpi\text{-promotion } \} \\ & \alpha\varpi/\beta \circ (\beta\circ\tau)/(\alpha\otimes\beta) = \tau \circ \mathrm{I}\otimes(\alpha\varpi/\beta) \end{array}$$



$$\begin{array}{ll} \equiv & \quad \{ (5.6) \} \\ & (\tau \circ \alpha\otimes\alpha\varpi)/(\alpha\otimes\beta) = \tau \circ \mathrm{I}\otimes(\alpha\varpi/\beta) \\ \equiv & \quad \{ \text{ unique extension property of colimits } \} \\ & \forall(i :: \tau \circ \alpha_i\otimes\alpha_i\varpi = \tau \circ \mathrm{I}\otimes(\alpha\varpi/\beta) \circ \alpha_i\otimes\beta_i) \\ \equiv & \quad \{ \text{ functors preserve composition } \} \\ & \forall(i :: \tau \circ \alpha_i\otimes\alpha_i\varpi = \tau \circ \alpha_i\otimes(\alpha\varpi/\beta \circ \beta_i)) \\ \equiv & \quad \{ (5.3) \} \\ & \text{true} \end{array}$$

□

## 5.2 Continuity

In this section we prove that functors induced by initial Hagino types are continuous. We begin by dualising the definitions of ascending chains, cocones and colimits; this is just a simple exercise in turning arrows around.

**Definition 5.2.1 (descending chain)** A *descending chain* is a family of functions $d_i \in D_i \longleftarrow D_{i+1}$ for $i \in \mathbb{N}$. We denote such a chain by $(d : D)$.

□

There is no danger of confusing this with the notation for ascending chains, since all chains in this section are descending.

Just as cocones corresponded to upper bounds, so the dual notion of cone corresponds to that of lower bound:

**Definition 5.2.2 (cone)** A *cone* of a descending chain $(d{:}D)$ is a pair $(L, \lambda)$, where $L$ is a type and $\lambda$ is a family of functions $\lambda_i \in D_i \longleftarrow L$ for $i \in \mathbb{N}$ with the property that for all $i$,

(5.7) $$d_i \circ \lambda_{i+1} = \lambda_i$$

□

The notion of greatest lower bound is replaced by that of limit:

**Definition 5.2.3 (limit)** A *limit* of a descending chain $(d{:}D)$ is a cone $(L, \lambda)$ with the property that for all cones $(K, \kappa)$ of $(d : D)$, there is a function $\lambda\backslash\kappa \in L \longleftarrow K$,

with the following uniqueness property: for all $f : L \longleftarrow K$,

$$\lambda\backslash\kappa = f \quad \equiv \quad \forall(i :: \lambda_i \circ f = \kappa_i). \tag{5.8}$$

□

From equation (5.8) we immediately obtain two useful identities. The first is the dual of equation (5.2) of the previous section:

$$\forall(i :: \lambda_i \circ \lambda\backslash\kappa = \kappa_i). \tag{5.9}$$

The second identity is the dual of property 5.1.4:

**Property 5.2.4** For all limits $(L, \lambda)$, $\lambda\backslash\lambda = \mathrm{I}_L$.
□

Again, the following definitions of continuity are the duals of the definitions of cocontinuity:

**Definition 5.2.5 (continuous)** A functor F is *continuous* if for all limits $(L, \lambda)$ of descending chains $(d : D)$, $(L_F, \lambda_F)$ is the limit of the descending chain $(dF : DF)$.
□

**Definition 5.2.6 (continuous)** A binary functor $\otimes$ is *continuous* if for all limits $(L, \lambda)$ of $(d : D)$ and $(K, \kappa)$ of $(e : E)$, $(L \otimes K, \lambda \otimes \kappa)$ is the limit of $(d \otimes e : D \otimes E)$.
□

We shall prove the following

**Theorem 5.2.7 (continuity of defined functors)** Let the functor $\varpi$ be defined by $A\varpi \triangleq \mu(\tau : A\otimes)$, where the bifunctor $\otimes$ is continuous and a relator. Then $\varpi$ is continuous.

**Proof:** Let $(L, \lambda)$ be the limit of $(d : D)$, and $(K, \kappa)$ the limit of $(d\varpi : D\varpi)$; it is sufficient to show that $(L\varpi, \lambda\varpi)$ is isomorphic to $(K, \kappa)$.

Since $(L\varpi, \lambda\varpi)$ is a cone of $(d\varpi, D\varpi)$, we have immediately a function $\kappa\backslash\lambda\varpi \in K \longleftarrow L\varpi$, which satisfies

$$\forall(i :: \kappa_i \circ \kappa\backslash\lambda\varpi = \lambda_i\varpi).$$



We must show that this function is bijective. We know more properties of catamorphisms than we do of functions of the form $\kappa\backslash\lambda\varpi$, so we begin by expressing this function as a catamorphism: in order to do so, we need to find a function of type $K \longleftarrow L \otimes K$. Using definition 5.2.2, we verify that $(L \otimes K, \tau \circ \lambda \otimes \kappa)$ is a cone of $(d\varpi : D\varpi)$ by:

$$\begin{array}{ll} & d_i\varpi \circ \tau \circ \lambda_{i+1} \otimes \kappa_{i+1} \\ = & \quad \{ \text{ evaluation of maps: (2.16) } \} \\ & \tau \circ d_i \otimes d_i\varpi \circ \lambda_{i+1} \otimes \kappa_{i+1} \\ = & \quad \{ \text{ functors preserve composition } \} \\ & \tau \circ (d_i \circ \lambda_{i+1}) \otimes (d_i\varpi \circ \kappa_{i+1}) \\ = & \quad \{ (L, \lambda) \text{ and } (K, \kappa) \text{ are cones } \} \\ & \tau \circ \lambda_i \otimes \kappa_i \end{array}$$

Thus $(L \otimes K, \tau \circ \lambda \otimes \kappa)$ is a cone, so the limit property of $(K, \kappa)$ yields the function

$$\kappa\backslash(\tau \circ \lambda \otimes \kappa) \in K \longleftarrow L \otimes K.$$

From the type of this function, we have the catamorphism

$$(\!|\kappa\backslash(\tau \circ \lambda \otimes \kappa)|\!) \in K \longleftarrow L\varpi.$$

We now prove

**Lemma 5.2.8** $\kappa\backslash\lambda\varpi = (\!|\kappa\backslash(\tau \circ \lambda \otimes \kappa)|\!)$.

**Proof:**

$$\begin{array}{ll} & \kappa\backslash\lambda\varpi = (\!|\kappa\backslash(\tau \circ \lambda \otimes \kappa)|\!) \\ \equiv & \quad \{ \text{ uniqueness: (5.8) } \} \\ & \forall(i :: \kappa_i \circ (\!|\kappa\backslash(\tau \circ \lambda \otimes \kappa)|\!) = \lambda_i\varpi) \\ \equiv & \quad \{ \text{ definition of maps: definition 2.3.1 } \} \\ & \forall(i :: \kappa_i \circ (\!|\kappa\backslash(\tau \circ \lambda \otimes \kappa)|\!) = (\!|\tau \circ \lambda_i \otimes \mathrm{I}|\!)) \\ \Leftarrow & \quad \{ \varpi\text{-promotion } \} \\ & \forall(i :: \kappa_i \circ \kappa\backslash(\tau \circ \lambda \otimes \kappa) = \tau \circ \lambda_i \otimes \mathrm{I} \circ \mathrm{I} \otimes \kappa_i) \\ \equiv & \quad \{ (5.9) \} \\ & \forall(i :: \tau \circ \lambda_i \otimes \kappa_i = \tau \circ \lambda_i \otimes \mathrm{I} \circ \mathrm{I} \otimes \kappa_i) \end{array}$$

$\equiv$ { functors preserve composition }

true

□

Now the point of expressing $\kappa\backslash\lambda\varpi$ as a catamorphism is that we know from the previous chapter that among the properties preserved by catamorphisms are injectivity and totality, and these properties are necessary to showing that $\kappa\backslash\lambda\varpi$ is bijective. We need one more lemma:

**Lemma 5.2.9** $\kappa\backslash(\tau \circ \lambda\otimes\kappa)$ is bijective.

**Proof:** We shall explicitly construct its inverse. By the continuity of $\otimes$, we have that $(L\otimes K, \lambda\otimes\kappa)$ is the limit of $(d\otimes d\varpi : D\otimes D\varpi)$. Remarking that $D_i \otimes D_i\varpi$ is isomorphic to $D_i\varpi$ leads to finding $(K, \tau\upsilon \circ \kappa)$ as a cone of $(d\otimes d\varpi : D\otimes D\varpi)$. We have

$d_i\otimes d_i\varpi \circ \tau\upsilon \circ \kappa_{i+1} = \tau\upsilon \circ \kappa_i$

$\equiv$ { $\tau$ is bijective }

$\tau \circ d_i\otimes d_i\varpi \circ \tau\upsilon \circ \kappa_{i+1} = \kappa_i$

$\equiv$ { evaluation of maps: (2.16) }

$d_i\varpi \circ \tau \circ \tau\upsilon \circ \kappa_{i+1} = \kappa_i$

$\equiv$ { $\tau$ is bijective }

$d_i\varpi \circ \kappa_{i+1} = \kappa_i$

$\equiv$ { $(K, \kappa)$ is a cone }

true

Thus by definition 5.2.2, $(K, \tau\upsilon \circ \kappa)$ is a cone, so we have the function

$$(\lambda\otimes\kappa)\backslash(\tau\upsilon \circ \kappa) \in L \otimes K \longleftarrow K.$$

To show that this is indeed the inverse of $\kappa\backslash(\tau \circ \lambda\otimes\kappa)$ we note first of all:

$(\lambda\otimes\kappa)\backslash(\tau\upsilon \circ \kappa) \circ \kappa\backslash(\tau \circ \lambda\otimes\kappa) = \mathrm{I}_{L\otimes K}$

$\equiv$ { property 5.2.4 }

$(\lambda\otimes\kappa)\backslash(\tau\upsilon \circ \kappa) \circ \kappa\backslash(\tau \circ \lambda\otimes\kappa) = (\lambda\otimes\kappa)\backslash(\lambda\otimes\kappa)$

$\equiv$ { uniqueness: (5.8) }

$\forall(i :: \lambda_i\otimes\kappa_i \circ (\lambda\otimes\kappa)\backslash(\tau\upsilon \circ \kappa) \circ \kappa\backslash(\tau \circ \lambda\otimes\kappa) = \lambda_i\otimes\kappa_i)$



$\equiv$ { (5.9) }

$\forall(i :: \tau\upsilon \circ \kappa_i \circ \kappa\backslash(\tau \circ \lambda\otimes\kappa) = \lambda_i\otimes\kappa_i)$

$\equiv$ { (5.9) }

$\forall(i :: \tau\upsilon \circ \tau \circ \lambda_i\otimes\kappa_i = \lambda_i\otimes\kappa_i)$

$\equiv$ { $\tau$ is bijective }

true

The proof that $\kappa\backslash(\tau \circ \lambda\otimes\kappa) \circ (\lambda\otimes\kappa)\backslash(\tau\upsilon \circ \kappa) = \mathrm{I}_K$ is symmetric to that above. Thus we have proven:

(5.10) $$(\kappa\backslash(\tau \circ \lambda\otimes\kappa))\upsilon = (\lambda\otimes\kappa)\backslash(\tau\upsilon \circ \kappa).$$

□

The above lemma proves that $\kappa\backslash(\tau \circ \lambda\otimes\kappa)$ is bijective; it is therefore injective and total, and so by property 4.2.11 of chapter 4, $(\!|\kappa\backslash(\tau \circ \lambda\otimes\kappa)|\!)$ is injective and total. Therefore, by lemma 5.2.8, we have proven one half of the bijection:

$$(\kappa\backslash\lambda\varpi)\upsilon \circ \kappa\backslash\lambda\varpi = \mathrm{I}_{L\varpi}.$$

(Note that whereas in the previous section we required only properties of functional catamorphisms, we are now appealing to properties of relational catamorphisms: this means that we must at this point strengthen our assumptions, and assume that $\otimes$ enjoys the additional properties of being a relator.)

We prove the other half of the bijection as follows:

$\kappa\backslash\lambda\varpi \circ (\kappa\backslash\lambda\varpi)\upsilon = \mathrm{I}_K$

$\equiv$ { property 5.2.4 }

$\kappa\backslash\lambda\varpi \circ (\kappa\backslash\lambda\varpi)\upsilon = \kappa\backslash\kappa$

$\equiv$ { (5.8) }

$\forall(i :: \kappa_i \circ \kappa\backslash\lambda\varpi \circ (\kappa\backslash\lambda\varpi)\upsilon = \kappa_i)$

$\equiv$ { (5.9) }

$\forall(i :: \lambda_i\varpi \circ (\kappa\backslash\lambda\varpi)\upsilon = \kappa_i)$

$\equiv$ { reverse; relators }

$\forall(i :: \kappa\backslash\lambda\varpi \circ \lambda_i\upsilon\varpi = \kappa_i\upsilon)$

$$
\begin{array}{ll}
 & \{ \text{ lemma 5.2.8 } \} \\
\forall(i :: (\!|\kappa\backslash(\tau \circ \lambda \circledcirc \kappa)|\!|) \circ \lambda_i \upsilon \varpi = \kappa_i \upsilon) & \\
\equiv & \{ \text{ property of maps: lemma 2.3.3 } \} \\
\forall(i :: (\!|\kappa\backslash(\tau \circ \lambda \circledcirc \kappa) \circ \lambda_i \upsilon \circledcirc 1|\!|) = \kappa_i \upsilon) & \\
\equiv & \{ \text{ uep for catamorphisms } \} \\
\forall(i :: \kappa_i \upsilon \circ \tau = \kappa\backslash(\tau \circ \lambda \circledcirc \kappa) \circ \lambda_i \upsilon \circledcirc \kappa_i \upsilon) & \\
\equiv & \{ \text{ reverse, relators } \} \\
\forall(i :: \tau \upsilon \circ \kappa_i = \lambda_i \circledcirc \kappa_i \circ (\kappa\backslash(\tau \circ \lambda \circledcirc \kappa))\upsilon) & \\
\equiv & \{ (5.10) \} \\
\forall(i :: \tau \upsilon \circ \kappa_i = \lambda_i \circledcirc \kappa_i \circ (\lambda \circledcirc \kappa)\backslash(\tau \upsilon \circ \kappa)) & \\
\equiv & \{ (5.9) \} \\
\text{true} &
\end{array}
$$

Thus we have proven the second half of the bijection, and the proof of the theorem is complete.
□

In the proof of the cocontinuity of $\varpi$ in section 5.1 above, the only properties we used were the cocontinuity of $\circledcirc$ and the unique extension property of Hagino types, so the proof is valid in any category with an initial object and in which colimits of ascending chains exist. For the same reason, the proof can easily be dualised to give a proof that if $\circledcirc$ is continuous, then $\xi$ is continuous, in a category with a terminal object and in which limits of descending chains exist. In the above proof of the continuity of $\varpi$, however, we found it necessary to use properties of relational catamorphisms in order to prove that $\kappa\backslash\lambda\varpi$ was injective and total. The proof is consequently only valid in a setting, such as the category of sets, which is sufficiently rich as to provide a model for the relational calculus. A further consequence is that we cannot dualise this proof to obtain a proof that $\xi$ is cocontinuous, as we do not yet have the mechanisms with which to dualise the properties of relational catamorphisms given in chapter 4. We return to this point in the next chapter.

# Chapter 6

# In Conclusion

The ambiguity in the title of this chapter is deliberate: a certain inconclusiveness in summarising the work presented in this thesis is unavoidable, for the work is still continuing. The least inconclusive chapters are 2 and 3, largely because these draw on a well-established body of research into initial algebra semantics (Ehrig and Mahr [18], Meseguer and Goguen [36] and Manes and Arbib [30] are good introductions and give further references). Our contribution here has been to apply systematically the unique extension properties of Hagino types to the deriving, proving and transforming of programs, in the spirit of the Bird-Meertens formalism, and always with an eye to exploiting the algebraic structure of the type.

Chapter 4 leaves the greatest scope for future work. In that chapter, we found it convenient to appeal to the set-theoretical interpretation of operations such as intersection and union, in order to be able to calculate with relations. The disadvantage of this is that we are thereby constrained to work in the category of sets, although it seems clear that a greater generality is possible. Work in progress by R. Backhouse, P. de Bruin, E. Voermans and the present author, at Groningen University, together with J. van der Woude of Eindhoven University, remedies this by constructing a calculus of relational catamorphisms on an axiomatic basis. This allows for a much neater presentation and, furthermore, makes explicit exactly what properties are required — thus clarifying, for example, the range of applicability of the proof of continuity of defined functors which we gave in chapter 5.

An important element missing from chapter 4 is the construction of relational catamorphisms for terminal Hagino types. The use of relational catamorphisms in the continuity proof of section 5.2 prevents the proof's being dualised to show the cocontinuity of functors derived from terminal Hagino types. However, the dualisation of the constructions of chapter 4 is not as straightforward as in the purely algebraic

setting of chapters 2 and 3, where it was simply a case of reversing arrows. For example, in the relational calculus, the dual notion of function is injection, and an injection need not itself be functional. This means that the duals of the results of chapter 4 are not always what one might at first expect. The induction principle, certainly, does not dualise in the way one would hope for. A useful form of "co-induction" does seem to hold for terminal types which are constructed as limits of descending chains in the manner described in chapter 5, but to reason from construction in this way seems unattractive, considering that it is not necessary for initial Hagino types.

In any case, the dualisation of relational catamorphisms to terminal types is an important area for future research, hopefully leading to a body of techniques which extends those developed in chapter 3 for specifying and calculating with programs on infinite structures.

Finally, a word on proofs. Throughout this thesis we have compared the Hagino paradigm of type definition with that of Martin-Löf's type theory. While the comparison has been by no means exhaustive, it became apparent that one of the main differences is that Martin-Löf's theory places the emphasis on induction and a natural deduction style of proof whereas the Hagino paradigm stresses unique extension properties and a calculational style of proof.

This is largely a matter of presentation, but style of presentation is an important issue in assessing the feasibility of a system as a programming logic. Certainly, the association of propositions with types and the close link between induction and the construction of recursive functions in Martin-Löf's theory provides an attractive unification of the activities of specifying, proving and deriving programs. In chapter 4 we gave some short examples of the use of guards and relational catamorphisms as a means of incorporating properties into calculations, but we really need to see more examples before we can judge how favourably these techniques compare in this respect with Martin-Löf's theory. As regards the developing of proofs, we feel that its calculational bias lends the Hagino paradigm a clear advantage. As the examples in this thesis illustrate, proof by unique extension or by promotion obviates to a great extent the need to keep track of assumptions and to go through the "ritual steps" of folding and unfolding in inductive proofs. To quote Goguen ([20], p.1):

> Once basic concepts have been formulated in a categorical language, it often seems that proofs "just happen": at each step there is a "natural" thing to try, and it works. ... It could almost be said that the purpose of category theory is to reduce all proofs to such simple calculations.

# Bibliography


[1] R.C. Backhouse. Aggregated segment sums. Dept. Comp. Sci., Groningen University, 1989.

[2] R.C. Backhouse. Naturality of homomorphisms. Lecture notes, International Summer School on Constructive Algorithmics, vol. 3, 1989.

[3] R.C. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. Technical Report CS 8606, Dept. Comp. Sci., Groningen University, 1986.

[4] R.C. Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS 8810, Dept. Comp. Sci., Groningen University, 1988.

[5] R.C. Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, June 1989.

[6] R.C. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[7] R.S. Bird. Constructive functional programming. International Summer School on Constructive Methods in Computing Science, Marktoberdorf, 24th July to 5th August 1988.

[8] 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.

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

[10] R.S. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[11] R.S. Bos and C. Hemerik. An introduction to the category theoretic solution of recursive domain equations. Technical Report Comp. Sci. Notes 88/15, Eindhoven University of Technology, 1985.

[12] P. Chisholm. *Investigations into Martin-Löf Type Theory as a Programming Logic*. Ph.D. thesis, Dept. Comp. Sci., Heriot-Watt University, Edinburgh, July 1988.

[13] P.J. de Bruin. Naturalness of polymorphism. Technical Report CS 8916, Dept. Comp. Sci., Groningen University, 1989.

[14] O. de Moor. Indeterminacy in optimization problems. Lecture Notes, International Summer School on Constructive Algorithmics, vol. 2, 1989.

[15] E.W. Dijkstra and W.H.J. Feijen. *Een Methode van Programmeren*. Academic Service, Den Haag, 1984. Also available as *A Method of Programming*, Addison-Wesley, Reading, Mass., 1988.

[16] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, 1990.

[17] P. Dybjer. An inversion principle for Martin-Löf's type theory. Dept. Comp. Sci., Univ. Göteborg, 1989.

[18] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag Berlin, 1985.

[19] P.J. Freyd, J.Y. Girard, A. Scedrov, and P.J. Scott. Semantic parametricity in polymorphic lambda calculus. In *Proc. 3rd Ann. Symp. on Logic in Computer Science*, pages 274–9. Computer Society Press, 1988.

[20] J. Goguen. A categorical manifesto. Technical Report PRG-72, Programming Research Group, Univ. Oxford, 1989.

[21] T. Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A. Poigne, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–57. Springer-Verlag LNCS 283, 1988.

[22] W.H. Hesselink. An algebraic calculus of commands. Technical Report CS 8808, Dept. Comp. Sci., Groningen University, 1988.



[23] W.H. Hesselink. Command algebras, recursion and program transformation. Technical Report CS 8812, Dept. Comp. Sci., Groningen University, 1988.

[24] C.A.R. Hoare and Jifeng He. The weakest prespecification. *Fundamenta Informaticae*, 9:51–84, 217–252, 1986.

[25] R. Hoogerwoord. *The Design of Functional Programs: A Calculational Approach*. Ph.D. thesis, Eindhoven University of Technology, 1989.

[26] J. Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65:154–170, 1958.

[27] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

[28] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[29] G. Malcolm and P. Chisholm. Polymorphism and information loss in Martin-Löf's type theory. Technical Report CS 8814, Dept. Comp. Sci., Groningen University, 1988.

[30] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1986.

[31] P. Martin Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118. North-Holland, 1975.

[32] L. Meertens. First steps towards the theory of rose trees. draft report, CWI, Amsterdam 1988.

[33] L. Meertens. Paramorphisms. To appear in *Formal Aspects of Computing*, 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. Constructing a calculus of programs. In J.L.A. van de Snepscheut, editor, *Conference on the Mathematics of Program Construction*, pages 66–90. Springer-Verlag LNCS 375, 1989.

[36] J. Meseguer and J.A. Goguen. Initiality, induction and computability. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–542. Cambridge University Press, 1985.

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

[38] B.A. Sijtsma. *Verification and Derivation of Infinite-List Programs*. Ph.D. thesis, Dept. Comp. Sci., Groningen University, 1988.

[39] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–83, 1982.

[40] N. Verwer. Homomorphisms, factorisation and promotion. Technical Report RUU-CS-90-5, Dept. Comp. Sci., Utrecht University, 1990.

[41] P. Wadler. Theorems for free! Draft report, Dept. Comp. Science, University of Glasgow, March 1989.

[42] F. Zwiggelaar. A new Horner's rule. Afstudeerverslag, Dept. Comp. Sci., Groningen University, 1989.

# Samenvatting

## Algebraische Data Typen en Programma Transformatie

Een computer programma verwerkt gegevens. Deze gegevens bezitten een of andere struktuur, die in zekere mate de struktuur van het programma bepaalt. Dit principe heeft geleid tot het ontwerpen van funktionele programmeertalen die toelaten dat men zijn eigen data strukturen definiëert. Zulke programmeertalen kennen meestal ook *pattern-matching*, waardoor het schrijven van rekursieve funkties volgens de struktuur van de gegeven data vergemakkelijkt wordt. De rol van data strukturen bij het schrijven van programma's vormde het uitgangspunt voor het onderzoek dat tot dit proefschrift heeft geleid.

Een theorie van data strukturen voor de informatika hoort aan zekere eisen te voldoen. Zij moet een notatie voor het definiëren van nieuwe data strukturen aanbieden, alsmede een leidraad voor het schrijven van rekursieve funkties die op de gedefiniëerde typen opereren. Vooral is het van belang dat de notatie voor deze funkties goed formeel hanteerbaar is. Het is immers dikwijls het gemakkelijkst om een evident korrekt programma te schrijven terwijl de efficiëntie van dit programma veel te wensen overlaat. Daarom is het nuttig programma's te kunnen transformeren op zo'n manier dat de korrektheid gehandhaafd wordt en de efficiëntie verbeterd. Kortom, een theorie van data strukturen moet ook een calculus voor het transformeren van programma's bevatten.

In dit proefschrift wordt een notatie voor het definiëren van data typen dmv zogenaamde initiële algebra's besproken. Deze notatie berust op een unieke extensie eigenschap. Deze vormt zowel een manier om rekursieve funkties te schrijven als de basis voor een calculus voor programma transformatie. De unieke extensie eigenschap en een gevolg daarvan, de zogenaamde „promotie stelling", lenen zich goed voor het uitvoeren van korte en fraaie programma transformaties. De notatie en enkele voorbeelden worden in het tweede hoofdstuk uiteengezet. Het derde hoofdstuk behandelt een eenvoudige uitbreiding van de notatie om ook data typen met oneindige elementen te kunnen definiëren, bv het type van oneindige rijen. Het vierde hoofdstuk gaat over het gebruik van relaties ipv funkties bij het afleiden en transformeren van programma's die op subtypen opereren. Hier speelt zowel de algebraische als de logische struktuur van het data type een belangrijke rol. Het vijfde hoofdstuk behandelt het bestaan van

de gedefiniëerde typen. Geparametriseerde typen worden in het tweede en het derde hoofdstuk besproken: daar wordt getoond dat ze een manier vormen om nieuwe typen te definiëren. Er wordt bewezen dat de zo gedefiniëerde typen bestaan. Slotopmerkingen zijn in het laatste hoofdstuk te vinden.