$\beta \dagger id ; abide_\dagger ; \alpha \ddagger \alpha ; id \ddagger \alpha^\cup ; id \ddagger (id \dagger f) \;=\; id \dagger (id \ddagger f) ; id \dagger \beta^\cup ; \beta \dagger \beta ; abide_\dagger ; \alpha \ddagger id$

$\equiv\; \beta \dagger id ; (exl \dagger exl) \vartriangle (exr \dagger exr) ; \alpha \times (id \dagger f) \;=\; \beta \dagger (id \times f) ; (exl \dagger exl) \vartriangle (exr \dagger exr) ; \alpha \times id \;\Leftarrow\; id ; exl = id \times f ; exl \;\wedge$

$id ; exr ; f = id \times f ; exr \qquad \psi ; id \ddagger (\alpha^\cup ; id \dagger f ; \varphi) ; \beta^\cup \;=\; \alpha^\cup ; id \dagger (\psi ; id \ddagger f ; \beta^\cup) ; \varphi$

$\equiv\; \alpha^\cup ; id \dagger \psi ; \beta \dagger id ; abide_\dagger ; \alpha \ddagger \alpha ; id \ddagger (\alpha^\cup ; id \dagger f ; \varphi) ; \beta^\cup \;=\; \alpha^\cup ; id \dagger (\psi ; id \ddagger f ; \beta^\cup) ; \beta \dagger \beta ; abide_\dagger ; \alpha \ddagger id ; id \ddagger \varphi ; \beta^\cup \;\Leftarrow$

Law and Order in Algorithmics

Maarten M. Fokkinga

4. Beschouw de eenvoudig getypeerde lambda-calculus met producten en sommen, waarbij een type-schema is opgebouwd uit variabelen middels constructoren $\rightarrow, \times, +$. De vraag of een type-schema τ precies n inwoners heeft, is beslisbaar. (Hierbij is een 'inwoner' van τ een equivalentieklasse van termen getypeerd met τ, en de equivalentie wordt voortgebracht door de gebruikelijke β- en η-regel en soortgelijke regels voor product en som.)

5. In het 26 bladzijden tellende hoofdstuk "IX: Subobjects, Quotient Objects, Factorization" van onderstaand boek komt het woord *quotient* alleen voor in de titel (en tweemaal in een oefening, in de combinaties 'quotient map' en 'quotient semigroup').

   H. Herrlich and G.E. Strecker. *Category Theory.* Allyn and Bacon Inc., Boston, 1973. Second edition published by Heldermann Verlag Berlin, 1979.

6. De kwaliteit van werkstukken voor theoretisch geörienteerde vakken in de Informatica zal aanzienlijk toenemen indien de studenten beter zijn geschoold in het leveren van wiskundige bewijzen. Bij die scholing dient ook geoefend te worden in het opsporen van logische fouten in incorrecte 'bewijzen' van ware beweringen.

7. In de NS dubbeldekkers is er bij optrekken en afremmen een hinderlijke tocht doordat de klapdeuren te gemakkelijk open gaan.

8. De bewegwijzering van de fietspaden in Amsterdam ZO is onvoldoende. Veel fietsers zouden al geholpen zijn als er bij ieder kruispunt op een eenvoudig paaltje het noorden wordt aangegeven.

9. De uitleg die ik in mijn proefschrift geef aan de titel *Law and Order in Algorithmics*, is bij de voor de hand liggende nederlandse vertaling niet mogelijk, maar wel bij *Orde(ning) en Wet in Algoritmiek.*

Enschede, 13 februari 1992 Maarten Fokkinga

STELLINGEN

behorende bij het proefschrift

Law and Order in Algorithmics

1. Met abstractie, in de formele betekenis van 'lambda-abstractie' ofwel 'parametrisatie', kan in computerprogramma's vorm gegeven worden aan abstractie in de informele betekenis van 'weglaten van aspecten' en 'verbergen van representatie-keuzen'.

   M.M. Fokkinga. Programming language concepts: the lambda calculus approach. In P.R.J. Asveld and A. Nijholt, editors, *Essays on Concepts, Formalisms, and Proofs*, volume 42 of *CWI Tracts*, pages 129–162. CWI, Amsterdam, 1987.

2. Het door Ehrig en Mahr gebruikte formalisme kan sterk vereenvoudigd worden in díe uiteenzettingen waar het niet gaat om de syntactische presentatie van specificaties en algebras.

   H. Ehrig and B. Mahr. *Fundamentals of Equational Specification 1: equations and initial semantics.* Springer Verlag, 1985.

   Hoofdstuk 5 van dit proefschrift.

3. Lambek en Scott noteren de compositie van zowel morfismen als ook functoren door ze naast elkaar te plaatsen. Het gebruik van een operatie-symbool met een grote scheidingskracht voor de compositie van morfismen vermindert het aantal nodige haakjes en verbetert daardoor de leesbaarheid van de formules.

   J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in advanced mathematics.* Cambridge University Press, 1986.

Law and Order in Algorithmics

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus
prof. dr. ir. J.H.A. de Smit
volgens besluit van het College van Dekanen
in het openbaar te verdedigen
op donderdag 13 februari 1992 te 16:00 uur

door

Martinus Maria Fokkinga

geboren op 25 december 1948 te Nijmegen

Dit proefschrift is goedgekeurd door de promotoren
prof. L.G.L.T. Meertens en
prof. dr. ir. L.A.M. Verbeek

Contents

Acknowledgements

This work could not have been completed without the help by my wife Ans. For over three years she has given invaluable mental and organisational support.

I also wish to thank several others that have contributed directly or indirectly to this work:

Lambert Meertens for inviting me to participate in his research project at the CWI (Centrum voor Wiskunde en Informatica) in Amsterdam;

the board of the Faculteit Informatica, headed by prof. dr. ir. A.J.W. Duijvestijn, for approving a three years secondment at the CWI;

my colleagues in the research group at the CWI, namely Lambert Meertens, Johan Jeuring and Jaap van der Woude, for teaching me a lot, and taking the trouble of commenting on various drafts of this thesis;

Erik Meijer for inviting me to jointly write a paper, out of which this thesis has grown, and for always having useful references and funny ideas;

the lectures at the Ameland meetings (one summer school and two workshops), in particular Roland Backhouse, and Grant Malcolm, for their inspiring lectures;

the members of the weekly meetings in Utrecht, for discussing my ideas and providing useful feedback: in addition to the people mentioned above, also Henk Doornbos, Frans Rietman, Doaitse Swierstra, Nico Verwer; and

Ross Paterson for several suggestions concerning Chapter 5.

The three years were not easy, but I've learned a lot, notably the limits (of myself and within categories). I wish to thank Jan Kuper for his mental support at a distance that I've appreciated very much, and Jaap van der Woude, once more, for his assistance in times when I was about to give up.

Chapter 1

Introduction

1a The theme of this text

There are various ways in which computer programs may be produced. In this text we are concerned with just one of them: the transformational method. In this method a specification is assumed to be given, and a program is derived by a stepwise transformation of the specification until an expression is obtained that is a satisfactory computer program. More specifically, we are concerned with that transformational approach to program construction in which the transformation steps are very similar to, if not essentially *the same* as, in high school algebra: in each step a part of the expression is replaced by an operationally different but semantically equivalent part, like the replacement of $(a+b)(a-b)$ by a^2-b^2 or the other way around.

Needless to say that our concern is but one, small aspect of the production of computer programs. In particular, it may be much harder to obtain a satisfactory formal specification of the informal requirements, than to transform the specification into a program.

1 **Algorithm.** Computer programs are quite complicated entities; there are many aspects that make the production of programs a difficult task. To get some grip on the production, it is wise to deal with the aspects in isolation, or one after the other, as much as possible. In this text we abstract from almost *all* aspects of a program except its input-output behaviour; what remains is mathematically known as a *function.* We shall speak of **algorithm** rather than function in order to allow for some aspects that do not make sense for functions, and to keep in mind the intended interpretation as a computer program. An algorithm is a computer program of which several aspects have been abstracted from.

We can now explain the words *algorithmics*, *law* and *order* from the title.

2 **Algorithmics.** By definition **algorithmics** is: the theory and practice of *calculating with algorithms.* Compare this with arithmetic: the theory and practice of calculating with numbers. To illustrate the latter, suppose a number x is specified by

$$ax^2 + bx + c \quad = \quad 0\,.$$

It is possible to calculate, rather than guess or 'just invent', numbers x that satisfy the specification, for instance $x = (-b + \sqrt{(b^2 - 4ac)})/2a$. To illustrate the former, suppose a list producing algorithm f is specified as follows, using the notation of Bird and Wadler [11].

$$y \in f x \quad \equiv \quad y - -x = [\,] \;\wedge\; x - -y = [\,],$$

that is, $f x$ yields a list containing all permutations of the list x. It is possible to calculate, rather than guess or 'just invent', algorithms f that satisfy the specification, for instance

$$\begin{array}{lcl} f\,[\,] & = & [[\,]] \\ f\,(x\!: y) & = & \mathit{concat}\;(\mathit{map}\;(\mathit{interleave}\;x)\;(f\,y)), \end{array}$$

where we don't bother to detail *interleave* any further.

In both algorithmics and arithmetic the calculations are based on calculation rules that describe properties of algorithms and numbers, respectively. In both algorithmics and arithmetic theorems may be used to take bigger steps in a calculation. In both algorithmics and arithmetic there is ample opportunity for machine assistance. In both algorithmics and arithmetic one cannot delegate the whole calculation of a solution for a specification to a machine, and human creativity (invention, eurekas) is needed. There seems to be no difference between algorithmics and arithmetic except for the entities that are dealt with: algorithms and numbers respectively. Yet this is a significant difference because algorithms are so much more complicated than numbers. In comparison with the theoretical and practical results of arithmetic, one might say that algorithmics is just in its infancy.

The further development of the theory part of algorithmics is a major theme throughout the entire text.

3 Law. In arithmetic there is a property saying that addition is associative; it is frequently used as a step in a calculation (usually implicitly). We call such a property a **law**; laws form the justifications for the steps in a calculation. A similar law relevant for algorithmics is the associativity of sequential composition (where the output of the first algorithm is the input for the second one).

The systematic use and production (invention, derivation) of such laws is a major theme throughout the entire text; not only within algorithmics but also —as a side effect— within category theory. Moreover, in Chapter 5 we develop, or rather propose and investigate, a semantic characterisation of the notion of law.

4 Order. Two algorithms f and g may be placed in an order relation, $f \sqsupseteq g$, according to the criterion whether f produces the same output as g does, for each input where g produces something. Such an order models the phenomenon in reality that one program f may differ from another program g only in that g does not terminate for some inputs for which f does.

Such an order —and its consequences for algorithmics— is the central theme of the investigations in Chapter 6.

* * *

So much for an explanation of the words in the title. Algebra and category theory are topics that also play a rôle in this text.

5 Algebra. The word algebra is used in this text in two meanings. In its technical meaning an *algebra* is a collection of operations together with a set on which the operations act. In a nontechnical sense *algebra* is the art of manipulating with formulas as in high school algebra: a formula F is broken up into its semantic relevant constituents and the pieces are assembled together into a different but semantic equivalent formula F', thus yielding the equality $F = F'$.

The choice of notation may greatly influence the ability to isolate the semantic relevant constituents. Consider for example the following five line law, in the notation of Bird and Wadler [10].

$$\forall(x, y :: \; h(g(x, y)) \; = \; g'(x, hy)) \qquad \Rightarrow \qquad h \circ f = f'$$

where

$$\begin{array}{lcl} f\;[\,] & = & e \\ f\;(x\!:\, y) & = & g(x, fy) \\ f'\;[\,] & = & h\,e \\ f'\;(x\!:\, y) & = & g'(x, f'y)\,. \end{array}$$

In the notation of the sequel this law takes just one line:

$$h \circ g = g' \;\circ\; id \times h \quad \Rightarrow \quad h \circ (\!|e, g|\!) = (\!|h\,e, g'|\!)\,.$$

The *dummies* x, y have been eliminated, and the *recursion pattern* has been captured by a single operation $(\!|\;|\!)$ so that f together with its entire definition is replaced by $(\!|e, g|\!)$. In $(\!|e, g|\!)$ the three semantic relevant constituents are clear: e, g, and $(\!|\;|\!)$.

Algebraic calculation is a major concern in this text.

6 Category Theory. Category theory is the study that seeks to unify concepts and constructions in various fields of mathematics. It achieves this goal by the use of a language, a formalism, which allows for various interpretations and, conversely, in which many diverse concepts and constructions can be expressed after suitable abstraction. The language has two beneficial aspects. First, there is an elegant style of expressing and proof (equational reasoning); for our intended interpretation in algorithmics this happens to be reasoning at the function level, without a need for introducing arguments explicitly. Second, the language often suggests or eases a far-reaching generalisation.

Because of these two reasons we shall use the framework of category theory.

7 Related research. Transformational programming is by no means new; Partsch [57] gives an extensive overview of current approaches. The more specific approach of algorithmics, namely an algebraic calculational style, has already been mentioned as early as 1969 by Burstall and Landin [13] and in 1978 by Backus [6]. *Algorithmics*, as defined above, was first formulated by Meertens [47], and was around the same time further developed for a particular datatype by Bird [9, 10]. Algorithmics is now being explored and extended by a number of research groups, in the Netherlands and Great Britain [3, 5, 34, 56, 33, 48, 36, 48, 53, 55, 54, 22, 21, 19, 18, 70, 76].

Bird [9, 10] identified several laws for the datatype of lists, and put them to use in actual calculations of algorithms. It was well-know that datatypes like lists could be described categorically, using the notion of *initiality*. For example, Spivey [69] derived several of Bird's laws for the datatype of lists from the categorical description. Moreover, Hagino [30, 29] showed that the dual of initiality, namely finality, can be used to define cartesian product and infinite datatypes like streams. Subsequently, Malcolm [42, 43] showed that the category theoretic approach lends itself well to actual *calculation* of algorithms.

An early attempt to exploit initiality for actual programming has been made by Aiello et al. [1] in 1978; apparently their work escaped the attention of computing scientists. The importance of initiality as a *principle of proof* has already been observed by Lehmann and Smyth [40]. Also Goguen [26] observes that initiality allows proofs by induction to be formulated without induction, and Backhouse [4] and Meertens [49] show the advantage of this for a *practical calculus* of algorithms: the induction-less proof steps are more compact and purely calculational.

There are many more aspects to algorithmics than covered in this text. For example, we do not give any large-scale calculation of an algorithm; Jeuring [35] and Jones [36] give nontrivial calculations. We do not present any high-level algorithmics theorem that is problem oriented rather than datatype oriented; in [21] we attempt at such a theorem. We do not say anything about indeterminacy and underspecification; Backhouse et al. [5], De Moor [53] and our case study [20] address this topic. We do not say anything about machine-assistance, nor about the design of a notation that is geared towards use in actual calculation. And so on.

8 Our contribution. This text is a continuation of the line set out by Malcolm [42, 43]. In Chapter 3 we give a categorical description of algorithmics that is heavily inspired by Malcolm and Hagino [30, 29]. That chapter contains no new results; what is new is the motivation for considering dialgebras, and several examples. We systematise the production and use of laws that govern the steps in a calculation; this occurs throughout the text. We apply the systematisation also to category theory itself, thus offering an alternative to the conventional style of proof in category theory, diagram chasing; this is done in Chapter 2.

Some small-scale contributions to algorithmics occur in Chapters 4–6. In Chapter 4 we produce several derived laws, and show them in action to prove some problems that puzzled me for a long time. In Chapter 5 we propose and investigate a semantic characterisation of the notion of law, thus offering a tool to investigate 'laws' in the same abstract way in which, say, 'algebras' may be investigated. Finally, in Chapter 6 we investigate the extension of the theory (production and use of laws) to a situation where a datatype is not initial with respect to the entire universe of discourse (though it is initial in a subset of that universe). This is relevant for programming with arbitrary recursive definitions, as in functional programming languages.

1b Preliminaries

9 What you should know about category theory. With the exception of Sections 2d–2f, there occur in this text a few concepts of category theory. The concepts are: *category*, *isomorphism*, *duality*, *functor* and *naturality*. If you are not familiar with these, it suffices to read the appendix.

From here onwards you are assumed to know these five concepts,
as well as the notions of cartesian product and disjoint union of sets.

Initiality and *finality* is explained in Section 2b; you should read that section anyway since it explains our way of exploiting and proving initiality properties.

10 The format of a calculation. We present a calculation in the way we have actually derived it (or would like to have derived it). The task of a calculation is to find a definition for some, possibly none, unknowns and to prove an equation or equivalence that contains the unknowns. In general we start with the main task and *reduce* it step by step to simpler tasks, until we finally arrive at **true**. In each step we apply a known *fact*, or *define* an unknown possibly in terms of new unknows, or, in order to proceed, *assume* that some property holds. In the end, all the definitions made along the way constitute a construction of the unknowns, and the assumptions remain as premises that imply the validity of the start equation or equivalence.

Sometimes a calculation can more elegantly be conducted and presented as a transformation between the left hand side of the equality (or equivalence) and the right hand side, using equalities (or equivalences) only. In such a case we usually start with the more complicated side, and transform it step by step to the simpler one.

This style of conducting and reading proofs requires some exercising to get used to; once mastered it turns out to be an effective way of working. Dijkstra and Scholten [16] discuss this style in detail, and attribute the calculational format to Feijen, and van Gasteren [25].

11 Notation. Here is the default typing of frequently occurring one-letter variables.

$\mathcal{A}, \mathcal{B}, \mathcal{C}$	categories
$A, \ldots, Z$	functors
M	sumtype and prodtype functors (formerly map functors)
T	transformer (special functor, Chapter 5 only)
$\dagger, \ddagger$	bifunctors (infix notation)
$a, b, c, \ldots$	objects
$e, f, g, h, j, \ldots$	morphisms
$\varphi, \psi, \chi, \ldots$	algebras (special morphisms)
α, β	initial and final algebras
$\gamma, \delta, \varepsilon, \eta$	natural transformations (γ, δ cones and cocones)
x, y, z	various entities (mostly morphisms but also objects)
$\mathcal{F}$	mapping on morphisms, not necessarily a functor.

One-letter variables B, C, K, I, L, S, U have a fixed meaning; I, L, S are explained below, U is the underlying functor introduced in Chapter 3, C, K occur in Chapter 2 only, and B in the examples of Chapter 5.

Formula $f\colon a \to_{\mathcal{A}} b$ means that f is a morphism in $\mathcal{A}$ with source a and target b ($\mathrm{src}_{\mathcal{A}}\, f = a$ and $\mathrm{tgt}_{\mathcal{A}}\, f = b$). We denote composition arrows of the "base" category (and all the categories that inherit its composition) in diagrammatic order: if $f\colon a \to b$ and $g\colon b \to c$ then $f \,;\, g\colon a \to c$. Composition of functors and other mappings is denoted by juxtaposition: $(FG)f = F(Gf)$. If $\dagger$ is a bifunctor (like $\times$ and $+$) and F, G are functors, then $F \dagger G$ denotes the functor defined by $(F \dagger G)x = Fx \dagger Gx$ for all objects and morphisms x. In particular, $I\!I = I \times I$; it maps each x onto $x \times x$. Following common practice we omit the subscript of a natural transformation if it can be derived or is clear from the context. Also, we assume in each formula that the free variables are typed in such a way that the formula makes sense, that is, the targets and sources match at each composition and objects and morphisms are in the appropriate category.

Product and Sum. In case you want to skip Section 2c where the categorical product and sum (coproduct) are discussed, we list here the notation that we use for them. The product functor is denoted $\times$, the *extractions* (projections) are denoted $\mathit{exl}, \mathit{exr}$ or, for three components, $\mathit{ex}_{3,0}, \mathit{ex}_{3,1}, \mathit{ex}_{3,2}$, and we write $f \vartriangle g$ (f split g) for what is commonly denoted $\langle f, g \rangle$. The notation for the sum suggests the duality: bifunctor $+$ and *injections* $\mathit{inl}, \mathit{inr}$ or $\mathit{in}_{n,i}$, and $f \triangledown g$ (f junc g) for $[f, g]$. (For product categories the extraction functors are denoted $\mathit{Exl}, \mathit{Exr}$ while the symbol $\vartriangle$ also denotes the tupling (pairing) of functors.)

Default category. The declaration that a category is the **default category** means that it is this category that should be mentioned whenever the notions or notations in the text require that some category be mentioned. We shall only use $\mathcal{S}et$ and identifier $\mathcal{C}$ as the default category. So, in particular, $\to_{\mathcal{C}}$ is often abbreviated to just $\to$ if $\mathcal{C}$ has been declared the default category.

Omitting objects. A functor is mainly a mapping of morphisms; its action on objects can be derived since $Fa = \mathrm{tgt}\, F\mathit{id}_a$ by one of the functor axioms. In the same vein, we shall define concepts in terms of morphisms as much as possible, and suppress the rôle of objects when it can be derived from the context.

Parsing. Juxtaposition associates to the *right*, so that $U\mu Fa = U(\mu(Fa))$, and binds **stronger** than any binary operation symbol, so that $Fa\dagger = (Fa)\dagger$. Binary operation symbol $;$ binds the **weakest** of all operation symbols in a term denoting a morphism. As usual, $\times$ has priority over $+$.

12 Naturals, lists, streams. We shall frequently use naturals, cons lists, cons′ lists, and streams in examples, assuming that you know these concepts. Here is some informal explanation; the default category is $\mathcal{S}et$.

A distinguished one-element set is denoted 1. Function $!_a\colon a \to 1$ is the unique function from a to 1. Constants, like the number zero, will be modeled by functions with 1 as source, thus $zero\colon 1 \to nat$. The sole member of 1 is sometimes written $()$, so that $zero() \in nat$ and $zero$ is called a *nullary* function.

For the **naturals** we use several known operations.

$zero$	:	$1 \to nat$	zero, considered as a function from 1
$succ$	:	$nat \to nat$	the successor function
add	:	$\mathit{II}\, nat \to nat$	addition.

The set nat consists of all natural numbers. Functions on nat may be defined by induction on the $zero, succ$-structure of their argument.

For lists we distinguish between several variants.

The datatype of **cons lists** over a has as carrier the set La that consists of finite lists only. There are two functions nil and $cons$.

$$\begin{array}{lll} nil & : & 1 \to La \\ cons & : & a \times La \to La\,. \end{array}$$

Depending on the context, nil and $cons$ are fixed for one specific set a, or they are considered to be polymorphic, that is, having the indicated type for each set a. In a very few cases a subscript will make this explicit. Each element from La can be written as a finite expression

$$cons(x_0,\ cons(x_1,\ \ldots\ cons(x_{n-1},\ nil)))\,.$$

So, functions over La can be defined by induction on the $nil, cons$ structure of their argument. For example, definitions of $size\colon La \to nat$ and $isempty\colon La \to La + La$ read

$$\begin{array}{lll} nil\ ;\ size & = & zero \\ cons\ ;\ size & = & id \times size\ ;\ add \end{array}$$

and

$$\begin{array}{lll} nil\ ;\ isempty & = & nil\ ;\ inl \\ cons\ ;\ isempty & = & cons\ ;\ inr\,. \end{array}$$

Function $isempty$ sends its argument unaffected to the left/right component of its result type according to whether it is/isn't the empty list. A boolean result may be obtained by post-composing *isempty* with $true \triangledown false$, see paragraph 11 or Section 2c for the *case* construct $\triangledown$. For each function $f\colon a \to b$ the so-called *map* f for cons lists, denoted Lf, is defined by

$$\begin{array}{lcl} nil_a \,;\, Lf & = & nil_b \\ cons_a \,;\, Lf & = & f \times Lf \,;\, cons_b\,. \end{array}$$

If L were a functor, these equations assert that *nil* and *cons* are natural transformations:

$$\begin{array}{lcl} nil & : & \underline{1} \dot{\to} L \\ cons & : & I \times L \dot{\to} L\,. \end{array}$$

We shall see that L really is a functor.

The datatype of **streams** over a has as carrier the set Sa that consists of infinite lists only. There are two functions to destruct a stream into a head in a and a tail that is a stream over a again.

$$\begin{array}{lcl} hd & : & Sa \to a \\ tl & : & Sa \to Sa\,. \end{array}$$

A function yielding a stream can be defined by inductively describing what its result is, in terms of applications of hd and tl. For example, the lists of naturals is defined as follows.

$$\begin{array}{lcl} from & : & nat \to S\,nat \\ from \,;\, hd & = & id \\ from \,;\, tl & = & succ \,;\, from \\ nats & : & 1 \to S\,nat \\ nats & = & zero \,;\, from \end{array}$$

By induction on n one can prove that

$$nats \,;\, tl^n \,;\, hd \quad = \quad zero \,;\, succ^n\,.$$

These functions act on infinite datastructures and the evaluation of *nats* on a computing engine requires an infinite amount of time. Yet these functions are *total*; for each argument the result is well-defined. For each function $f\colon a \to b$ the so-called *map* f for streams, denoted Sf, is defined by

$$\begin{array}{lcl} Sf \,;\, hd_b & = & hd_a \,;\, f \\ Sf \,;\, tl_b & = & tl_a \,;\, Sf\,. \end{array}$$

If S were a functor, these equations assert that hd and tl are natural transformations:

$$\begin{array}{lcl} hd & : & S \dot{\to} I \\ tl & : & S \dot{\to} S\,. \end{array}$$

We shall see that S really is a functor.

The datatype of **cons′ lists** over a has as carrier the set $L'a$ that consists of all *finite and infinite* lists, called cons′ lists. There are several relevant functions.

$$\begin{array}{lcl} nil' & : & 1 \to L'a \\ cons' & : & a \times L'a \to L'a \\ destruct' & : & L'a \to 1 + a \times L'a \\ isempty' & : & L'a \to L'a + L'a \end{array}$$

with

$$\begin{array}{lcl} nil' \,;\, destruct' & = & inl \\ cons' \,;\, destruct' & = & inr \\ nil' \,;\, isempty' & = & nil' \,;\, inl \\ cons' \,;\, isempty' & = & cons' \,;\, inr\,. \end{array}$$

Since cons′ lists are possibly infinite, ‘definitions’ by induction on the nil', $cons'$-structure of cons′ lists is in general not possible; that would give *partially defined* functions, and these do not exist in our intended universe of discourse $\mathcal{S}et$. For example, consider the following equations with “unknown $size'$”.

$$\begin{array}{lcl} nil' \,;\, size' & = & zero \\ cons' \,;\, size' & = & id \times size' \,;\, add \end{array}$$

These do **not** define a total function $size'$: $L'a \to nat$, in contrast to the situation for cons lists. (Notice also the difference with the usual datatype of lists of nonstrict functional programming languages: next to finite and infinite lists, it comprises also partially defined lists.)

Chapter 2

Categories algebraically

'Diagram chasing' is an established proof technique in Category Theory. *Algebraic calculation* is a good alternative; made possible thanks to a *notation* for various unique arrows and a suitable *formulation* of initiality, and the properties brought forward by initiality.

2a Introduction

Category Theory [41] is a field of mathematics that seeks to discuss and unify many concepts occurring in mathematics. In the last decade it has proved useful for computing science as well; this may be evident from the rapidly growing number of conferences and publications with 'Category Theory' and 'Computer Science' in their title, for example [31, 7, 61, 62, 28, 65]. Not only is category theory helpful to formalise and prove results for theoretical aspects of computing science, like lambda calculus theory, denotational semantics, and fundamentals of algebraic specification, but also to formalise and prove results for practical aspects like language design and implementation (e.g., Hagino [30, 29], Reynolds [64] and Cousineau et al [14]) and program derivation (e.g., Malcolm [42, 43], Meijer [50], Paterson [59], and this text).

In this chapter we pave the way for a style of proof that is an alternative to the conventional one in Category Theory: *calculation* instead of *diagram chasing*. In effect, it is a form of Functional Programming. Let us explain the key-words.

1 **Category.** Roughly said, a category is just a collection of arrows with the closure property that "composition of two arrows f and g with $\text{target}(f) = \text{source}(g)$, is an arrow again." Thus, a mathematical structure, when studied categorically, has to be modeled as a system of arrows. This may pose serious problems to the newcomer; Arbib

and Manes [2] teach how to think in terms of arrows. The prominent rôle of arrows invites to use pictures containing (a lot of) arrows, so-called diagrams, as a tool in categorical proofs. The conventional style of proof is diagram chasing (explained below); we offer an alternative: algebraic calculation. To do so, we give a systematic treatment of the calculation properties brought forward by *initiality*, and show them in action on a variety of examples. Initiality is a categorical concept by which many mathematical constructions can be characterised.

2 Diagrams and diagram chasing. The basic task in a categorical proof is to show the existence of an arrow, or to show the equality of two arrows, when some other arrows and objects are given. There are several reasons why diagrams may be helpful, and one has to face all of them when judging the relative merits of an alternative style of proof. Let us consider (all?) four of these reasons.

- *Typing.* A picture may clearly indicate which arrows have a common source or target, much more so than a linear listing of the arrows with the source and target given for each of them.

  Remark. The need for a survey of the sources and targets of the arrows is partly caused by the notation $f\colon a \rightarrow b$ and $g\colon b \rightarrow c$ to indicate the source and target (called *typing*), and the notation $g \circ f$ for their composition. We choose the notation $f \mathbin{;} g$ for composition, so that $f \mathbin{;} g\colon a \rightarrow c$ falls out naturally. (An alternative would be to use the notation $f\colon a \leftarrow b$ and $g\colon b \leftarrow c$ so that $f \circ g\colon a \leftarrow c$.)
  Though a consistent notation obviates in some cases the need for a survey of the common sources and targets, we do **not** claim that it does so in all cases. Pictures are helpful in presenting and viewing the data in an organised way.

- *Naming.* Initiality means that for certain pairs of source and target there is precisely one arrow in between. A picture is a suitable tool to indicate such an arrow, typically by a dashed line, and to attach a name to it for use in the text. Without pictures one usually introduces such an arrow by a phrase like "Let f be the unique arrow from *this* to *that* that exists on account of the initiality of *such-and-so*."

  Remark. We shall use a standard notation for various 'unique' arrows; the notation will clearly suggest the source and target, as well as some other properties. Thanks to the availability of a notation there is no need to interrupt an argument or calculation for a verbose introduction of such an arrow: you can just denote it.

- *Commuting diagrams.* Equality of arrows can be indicated pictorially if, by convention, in the picture any two (composite) arrows with the same source and the same target are equal. Thus $f \mathbin{;} g = h$ appears as a triangle, and $f \mathbin{;} g = h \mathbin{;} j$ as a quadrangle. This convention is called *commutativity of the diagram.* A commuting composite diagram is a very economical way of showing several equalities simultaneously without duplication of subterms that denote arrows. Moreover, the diagram may present additional information, like sources and targets.

Remark. I can hardly believe that a triangle with edges labelled f, g, h is clearer than the formula $f ; g = h$, and a square with edges f, g, h, j is clearer than $f ; g = h ; j$. In fact, even complicated formulas like $f_0 ; f_1 ; \dots ; f_{m-1} = g_0 ; g_1 ; \dots ; g_{n-1}$ are not more (and no more) understandable by drawing the left-hand side and right-hand side as a stretched, possibly wriggled quadrangle. In almost any diagram there is one equation of interest (the theorem) and the other equations in the diagram are just auxiliary, for use in the proof only; in that case there is no need to display them all at once (if an alternative proof does the job). Similarly, the information about source and target of each term occurring in a proof is often not helpful for understanding the main equation or verifying the proof steps, as we will see.

- *Diagram chasing.* Pasting several commuting diagrams together along common arrows gives a commuting diagram as result. It is an easy, visual, reliable style of proving equality of arrows. Apart from the purpose to give an overview, as mentioned above, this is the main reason why composite diagrams appear that are more complicated than the simple polygons.
  It is particularly easy to extend a diagram with an arrow; in a calculation one would have to *copy* the equation obtained thus far, and transform that a little.

  Remark. This use of diagrams may be quite helpful when conducting a proof on a blackboard, with an eraser at hand. (Also, it lends itself well for presentations with an overhead projector, using overlays.) However, in the final picture the history is completely lost. It is then just a puzzle, called *diagram chasing*, to find out what arrows exist for what reason, and what subdiagrams commute on what grounds. Moreover, it is even much harder to read off from the final diagram for what reason a certain arrow is the only one possible that makes a certain subdiagram commute. It is all this implicit information that is so clearly present in the calculations below.

So far for the reasons to use pictures and diagram chasing, and our objections to some of these.

3 Calculation with initiality. There is no problem in presenting the pasting of two diagrams as a calculation. For example, pasting 'squares'

$$\text{(a)}\quad \varphi ; f \;=\; p ; \psi \qquad\text{and}\qquad \text{(b)}\quad \psi ; g \;=\; q ; \chi$$

along 'edge' ψ yields $\varphi ; f ; g = p ; q ; \chi$. This is rendered in a one or two step calculation as follows.

$\varphi ; f ; g = p ; q ; \chi$

$\equiv$ in left hand side (a): $\varphi ; f = p ; \psi$,
in right hand side (b)$^{\cup}$: $q ; \chi = \psi ; g$

$p ; \psi ; g = p ; \psi ; g$

$\equiv$ reflexivity of equality

true.

More important is a formalisation of initiality that lends itself to such a calculational, equational reasoning. By definition, a is initial if for each target b there is precisely one arrow from a to b. Formally, a is initial if, for all b,

$$\exists(x :: \; x\colon a \to b \;\; \wedge \;\; \forall(y :: \; y\colon a \to b \;\; \Rightarrow \;\; y = x)) .$$

It is the presence of the existential quantifier (and the universal one in its scope) that hinders equational reasoning. An equivalent formalisation of initiality of a reads: there exists a function $\mathcal{F}$ such that, for all b and x,

$$x\colon a \to b \quad \equiv \quad x = \mathcal{F}b .$$

Indeed, substituting $x = \mathcal{F}b$ gives $\mathcal{F}b\colon a \to b$ (there is at least one arrow), and the implies part of the equivalence gives that there is at most one arrow. We shall see that this formalisation is the key to calculational reasoning. The use of equivalences to characterise initiality (and more generally, universality) has been thoroughly advocated by Hoare [31]. As far as we know, Malcolm [42] was the first to use this style of reasoning in a formal way for the derivation of functional programs over initial algebras.

4 Functional programming. In this text all arrows (in the sequel called morphisms) in the "base" category may be interpreted as typed total functions; there is simply no axiom for the category under consideration, that prohibits this interpretation. Therefore one may interpret our activity as functional programming, though for specifications that are a bit unusual. The combinations and transformations of morphisms (functions) are fully in the spirit of Backus [6] and Meertens [47]. One should note that nowhere in this text a morphism (function) is applied to an argument, except in examples; it is just by composing functions in various ways that new functions are formed and equalities are proved. The absence of restrictions on combining functions (except for typing constraints) has often been claimed to be a major benefit of functional programming, for example by Backus [6] and Hughes [32].

5 Historical remark. Originally our interest was in the development of a calculus for the derivation of algorithms from a specification, as proposed by Meertens [47] and Bird [9, 10]. Category theory provides a suitable medium to formalise the notion of datatype, as shown by Lehmann and Smith [40], Manes and Arbib [45], and many others. Malcolm [42] showed that *actual calculations* of algorithms can be rendered in a categorical style. It is from here a small step to apply the calculational style of algorithm derivation more generally to category theory itself.

The overall acceptance of diagram chasing is presumably the cause that this style of deriving categorical properties is relatively unknown. Indeed, only recently books and papers on category theory have appeared in which equational reasoning is explicitly strived for; for example by Lambek and Scott [37], Hoare [31].

6 Overview. The remainder of the chapter is organised as follows. In the next section we define some lesser known categorical concepts and discuss initiality. Then we specialise the laws for initiality to products and sums in Section 2c, to coequalisers and kernel pairs in Section 2d, and to colimits in general in Section 2e. Each of these sections contains one or more examples of a calculation for the derivation of a well-known result. We conclude with a worked-out example in Section 2f: the construction of an induced congruence. Many more examples of categorical calculations occur in the remainder of the text.

Sections 2b, 2c are essential for the following chapters; Sections 2d, 2e, and 2f may be skipped without loss of continuity (but 2f depends on all preceding sections). Sections 2d, 2e, and 2f assume more familiarity with categorical concepts, and are intended as a case study in the calculational approach to category theory.

The proof of the pudding is in the eating: the categorician should compare our algebraic calculations with the usual pictorial proofs, and pay attention to the precision, conciseness, and clarity with which various steps in the proofs are stated, and to the absence of verbose or pictorial introductions of various unique arrows.

2b Preliminaries and Initiality

Throughout the chapter $\mathcal{C}$ is the default category.

7 Categories built upon $\mathcal{C}$. Often an interesting construction in $\mathcal{C}$ can be characterised by initiality in a category $\mathcal{A}$ built upon $\mathcal{C}$. We say $\mathcal{A}$ is **built upon $\mathcal{C}$** if: each morphism of $\mathcal{A}$ is a -special- morphism in $\mathcal{C}$ and $\mathcal{A}$'s composition and identities are that of $\mathcal{C}$. So, $\mathcal{A}$ is fully determined by defining its objects and morphisms. Moreover, 'built upon' is a reflexive and transitive relation. Here are some examples that we'll meet in the sequel; skip the description upon first reading. (The categorician may recognise $\bigvee(D)$ as the category of *cocones* for the *diagram* D. Dually, the category of *cones* for D is denoted $\bigwedge(D)$. I owe these notations, and those for D below, to Jaap van der Woude.)

Category $\bigvee(\vec{a})$, where $\vec{a}$ is an n-tuple of objects in $\mathcal{C}$. An object in $\bigvee(\vec{a})$ is: an n-tuple of morphisms in $\mathcal{C}$ with a common target and the objects $\vec{a}$ as sources, as suggested by the symbol $\bigvee$ for the case $n = 2$. Let $\vec{f}$ and $\vec{g}$ be such objects; then a morphism from $\vec{f}$ to $\vec{g}$ in $\bigvee(\vec{a})$ is: a morphism x in $\mathcal{C}$ satisfying $f_i \,;\, x = g_i$ for each index i of the n-tuple. It follows that $x\colon \operatorname{tgt} \vec{f} \to \operatorname{tgt} \vec{g}$. (As a special case, category $\bigvee(a)$ is known as the co-slice category 'under a', usually denoted $a/\mathcal{C}$.) An object in $\bigvee(a, a)$ is a *parallel pair* with source a.

Category $\bigvee(f \| g)$, where f and g are morphisms in $\mathcal{C}$ with a common source and a common target. An object in $\bigvee(f \| g)$ is: a morphism p for which $f \,;\, p = g \,;\, p$. Let p and q be such objects; then a morphism from p to q in $\bigvee(f \| g)$ is: a morphism x in $\mathcal{C}$ satisfying $p \,;\, x = q$. (So, $\bigvee(f \| g)$ is a full subcategory of $\bigvee(a)$ where $a = \operatorname{tgt} f = \operatorname{tgt} g$.)

Category $\bigvee(f \ulcorner g)$, where f, g are morphisms in $\mathcal{C}$ with a common source. An object in $\bigvee(f \ulcorner g)$ is: a tuple (h, j) satisfying $f \,;\, h = g \,;\, j$. Let (h, j) and (k, l) be two objects; then a morphism from (h, j) to (k, l) in $\bigvee(f \ulcorner g)$ is: a morphism x in $\mathcal{C}$ satisfying $h \mathbin{;} x = k$ and $j \mathbin{;} x = l$. (This category is used to define the pushout of f, g.)

Category $\mathcal{A}lg(F)$, where F is an endofunctor on $\mathcal{C}$. An object φ of $\mathcal{A}lg(F)$ is: a morphism $\varphi\colon Fa \rightarrow a$ in $\mathcal{C}$, for some a called the *carrier* of φ. Let φ and ψ be such objects; then a morphism from φ to ψ in $\mathcal{A}lg(F)$ is: a morphism x in $\mathcal{C}$ satisfying $\varphi \mathbin{;} x = Fx \mathbin{;} \psi$. It follows that x: carrier $\varphi \rightarrow$ carrier ψ, and carrier $\varphi = \mathrm{tgt}\, \varphi$. An object φ is called an F-*algebra*, and a morphism x is called an F-*homomorphism*. We shall explain this in more detail in the chapters to come.

8 Initiality. Let $\mathcal{A}$ be a category, and a an object in $\mathcal{A}$. Then a is **initial** in $\mathcal{A}$ if:

$$x\colon a \rightarrow_{\mathcal{A}} b \qquad \equiv \qquad x = (\!| a \rightarrow b |\!)_{\mathcal{A}} \qquad\qquad \text{CHARN}$$

Here $(\!| a \rightarrow b |\!)_{\mathcal{A}}$ is just a notation, a name, for a morphism (depending on $\mathcal{A}, a$ and b), and all free variables in the line are understood to be universally quantified, except those that have been introduced in the immediate context ($\mathcal{A}$ and a in this case). 'CHARN' is mnemonic for Characterisation. The $\Rightarrow$ part of CHARN says that each morphism x with a as its source, is uniquely determined by its target b (if it exists at all). From the $\Leftarrow$ part, taking $x := (\!| a \rightarrow b |\!)_{\mathcal{A}}$, it follows that for each b there is a morphism from a to b. Thus $(\!| \ |\!)$ is a standard name for the unique morphisms from a. Often there is a more specific notation that better suggests the resulting properties (see the following sections).

Of course, when $\mathcal{A}$ is clear from the context we write $(\!| a \rightarrow b |\!)$ rather than $(\!| a \rightarrow b |\!)_{\mathcal{A}}$. It often happens that one initial object in $\mathcal{A}$ is fixed, and in that case $(\!| b |\!)$ abbreviates $(\!| a \rightarrow b |\!)$. The usual notation for $(\!| b |\!)_{\mathcal{A}}$ is $!_b$ or $¡_b$. The !-notation doesn't work well for categories built upon $\mathcal{A}$ since the notation of a and b may become too large for a subscript. In the "base" category the notation $\mathit{0}$ denotes an initial object.

Finality is dual to initiality; an object a is **final** if: for each object b there exists precisely one morphism from b to a. The default notation for this unique morphism is $[\!(b \rightarrow a)\!]_{\mathcal{A}}$, and the characterisation reads

$$x\colon b \rightarrow_{\mathcal{A}} a \qquad \equiv \qquad x = [\!(b \rightarrow a)\!]_{\mathcal{A}}.$$

In the "base" category the notation $\mathit{1}$ denotes a final object.

9 Corollaries. Here are some consequences of CHARN. A substitution for x such that the right-hand side becomes true yields SELF, and a substitution for b, x such that the left-hand side becomes true yields ID:

$$(\!| a \rightarrow b |\!)_{\mathcal{A}}\colon a \rightarrow_{\mathcal{A}} b \qquad\qquad \text{SELF}$$

$$id_a = (\!| a \rightarrow a |\!)_{\mathcal{A}} \qquad\qquad \text{ID}$$

Next we have the Uniqueness and Fusion property (still assuming a initial in $\mathcal{A}$):

$$x, y\colon a \rightarrow_{\mathcal{A}} b \qquad \Rightarrow \qquad x = y \qquad\qquad \text{UNIQ}$$

$$x\colon b \to_{\mathcal{A}} c \quad \Rightarrow \quad (\!|a \to b|\!)_{\mathcal{A}} \,;\, x = (\!|a \to c|\!)_{\mathcal{A}} \qquad \text{FUSION}$$

The 'proof' of UNIQ is left to the reader. For FUSION we argue (suppressing $\mathcal{A}$ and a):

$$\begin{array}{ll} & (\!|b|\!) \,;\, x = (\!|c|\!) \\ \equiv & \quad \text{CHARN}[\,b, x := c, (\!|b|\!) \,;\, x\,] \\ & (\!|b|\!) \,;\, x\colon a \to c \\ \Leftarrow & \quad \text{composition} \\ & (\!|b|\!)\colon a \to b \quad \wedge \quad x\colon b \to c \\ \equiv & \quad \text{SELF, and premise} \\ & \textbf{true}. \end{array}$$

These five laws become much more interesting in case category $\mathcal{A}$ is built upon another one, and $\to_{\mathcal{A}}$ is expressed as one or more equations in the underlying category. In particular the importance of law FUSION cannot be over-emphasised; we shall use it quite often. If the statement $x\colon b \to_{\mathcal{A}} c$ boils down to the equation $c = b \,;\, x$ (which is the case when $\mathcal{A} = \bigvee(a)$), law FUSION can be formulated as an unconditional equation (by substituting $c := b \,;\, x$ in the consequent, giving $(\!|b|\!) \,;\, x = (\!|b \,;\, x|\!)$). In the case of initial algebras UNIQ captures the pattern of proofs by induction that two functions x and y are equal; in several other cases UNIQ asserts that a collection of morphisms is jointly epic.

10 Well-formedness condition. In general, when $\mathcal{A}$ is built upon another category, $\mathcal{C}$ say, the well-formedness condition for the notation $(\!|b|\!)$ is that b (viewed as a composite entity in the underlying category $\mathcal{C}$) is an object in $\mathcal{A}$; this is not a purely syntactic condition.

$$b \text{ in } \mathcal{A} \quad \Rightarrow \quad (\!|a \to b|\!)_{\mathcal{A}} \text{ is a morphism in } \mathcal{C} \qquad \text{TYPE}$$

In the sequel we adhere to the (dangerous?) convention that in each law the free variables are quantified in such a way that the well-formedness condition, the premise of TYPE, is met.

11 Application. Here is a first example of the use of these laws: proving that an initial object is unique up to a unique isomorphism. Suppose that both a and b are initial. We claim that $(\!|a \to b|\!)$ and $(\!|b \to a|\!)$ establish the isomorphism and are unique in doing so. By TYPE and SELF they have the correct typing. We shall show

$$x = (\!|a \to b|\!) \;\wedge\; y = (\!|b \to a|\!) \qquad \equiv \qquad x \,;\, y = id_a \;\wedge\; y \,;\, x = id_b\,,$$

that is, both compositions of $(\!|a \to b|\!)$ and $(\!|b \to a|\!)$ are the identity, and conversely the identities can be factored only in this way. We prove both implications of the equivalence at once.

$$\begin{array}{ll} & x = (\!|a \to b|\!) \quad \wedge \quad y = (\!|b \to a|\!) \\ \equiv & \quad \text{CHARN} \end{array}$$

$$
\begin{array}{ll}
 & x\colon a \to b \ \wedge\ y\colon b \to a \\
\equiv & \quad \text{composition} \\
 & x \,;\, y\colon a \to a \ \wedge\ y \,;\, x\colon b \to b \\
\equiv & \quad \text{CHARN} \\
 & x \,;\, y = (\!|a \to a|\!) \ \wedge\ y \,;\, x = (\!|b \to b|\!) \\
\equiv & \quad \text{ID} \\
 & x \,;\, y = id_a \ \wedge\ y \,;\, x = id_b.
\end{array}
$$

The equality $(\!|a \to b|\!) \,;\, (\!|b \to a|\!) = id_a$ can be proved alternatively using ID, FUSION, and SELF in that order. (This gives a nice proof of the weaker claim that initial objects are isomorphic.)

2c Products and Sums

Products and sums are categorical concepts that, specialised to category $\mathcal{S}et$, yield the well-known notions of cartesian product and disjoint union. (In other categories products and sums may get a different interpretation.)

12 Disjoint union. As an introduction to the definition of the categorical sum, we present here a categorical description of the disjoint union. Let $\mathcal{C}$ be $\mathcal{S}et$. The disjoint union of sets a and b is a set $a + b$ with several operations associated with it. There are the injections

$$
\begin{array}{l}
inl\colon a \to a + b \\
inr\colon b \to a + b,
\end{array}
$$

and there may be a predicate that tests whether an element in $a + b$ is $inl(x)$ or $inr(y)$ for some $x \in a$ or some $y \in b$. Using the predicate one can define an operation that in programming languages is known as a **case** construct, and vice versa. The case construct of f and g is denoted $f \triangledown g$ and has the following typing and semantics.

$$f \triangledown g\colon a + b \to c \qquad \text{for } f\colon a \to c \text{ and } g\colon b \to c$$

and

$$
\begin{array}{lcll}
x \,;\, inl \,;\, f \triangledown g & = & x \,;\, f & \text{for each } x\colon 1 \to a \\
y \,;\, inr \,;\, f \triangledown g & = & y \,;\, g & \text{for each } y\colon 1 \to b.
\end{array}
$$

Here we have used "nullary" functions x from the one-point set to a to indicate an element in the set a. By extensionality the two equations read

$$inl \,;\, f \triangledown g \;=\; f \qquad \text{and} \qquad inr \,;\, f \triangledown g \;=\; g\,.$$

Moreover, $f \triangledown g$ is the only solution for x in

$$inl \,;\, x \;=\; f \qquad \text{and} \qquad inr \,;\, x \;=\; g\,.$$

This is an important observation; it holds for each representation of disjoint union! Indeed, a 'disjoint union'-like concept for which the claim does not hold, is normally not considered to be a proper 'disjoint union' of a and b. In summary, we call $\mathit{inl}\colon a \to d$ and $\mathit{inr}\colon b \to d$ together with their target d a disjoint union of a and b if and only if for each $f\colon a \to c$ and $g\colon b \to c$ there is precisely one function, henceforth denoted $f \triangledown g$, such that

$$\mathit{inl} \mathbin{;} x = f \ \wedge\ \mathit{inr} \mathbin{;} x = g \quad \equiv \quad x = f \triangledown g\,.$$

This is an entirely categorical formulation. In addition, the form of the equivalence suggests to look for a characterisation by means of initiality (or finality). The entities inl and inr have a common target and have a and b as source respectively. Thus they are objects in the category $\bigvee(a,b)$, and, indeed, the x and $f \triangledown g$ in the equivalence above are morphisms in this category.

This completes the introduction to the definition below. Since there are categories in which the objects are not sets, the categorical construct is called *sum* rather than disjoint union.

13 Sum. Let $\mathcal{C}$ be arbitrary, the default category, and let a, b be objects. A **sum** of a and b is: an initial object in $\bigvee(a,b)$; it may or may not exist. Let $\mathit{inl}, \mathit{inr}$ be a sum of a and b; their common target is denoted $a + b$. We abbreviate $(\!|\mathit{inl}, \mathit{inr} \to f, g|\!)_{\bigvee(a,b)}$ to just $f \triangledown g$, not mentioning the dependency on a, b and $\mathit{inl}, \mathit{inr}$. (The usual categorical notation for $f \triangledown g$ is $[f, g]$.)

$$f\colon a \to c \ \wedge\ g\colon b \to c \quad \Rightarrow \quad f \triangledown g\colon a + b \to c \qquad \triangledown\text{-TYPE}$$

Working out $\to_{\bigvee(a,b)}$ in terms of equations in $\mathcal{C}$, morphisms $\mathit{inl}, \mathit{inr}$ and operation $\triangledown$ are determined ("up to isomorphism") by law CHARN, and consequently also satisfy the other laws.

$$\begin{array}{llll}
\mathit{inl} \mathbin{;} x = f \ \wedge\ \mathit{inr} \mathbin{;} x = g & \equiv & x = f \triangledown g & \triangledown\text{-CHARN} \\
\mathit{inl} \mathbin{;} f \triangledown g = f \ \wedge\ \mathit{inr} \mathbin{;} f \triangledown g = g & & & \triangledown\text{-SELF} \\
\mathit{inl} \triangledown \mathit{inr} = \mathit{id} & & & \triangledown\text{-ID} \\
\mathit{inl} \mathbin{;} x = \mathit{inl} \mathbin{;} y \ \wedge\ \mathit{inr} \mathbin{;} x = \mathit{inr} \mathbin{;} y & \Rightarrow & x = y \quad (\text{"jointly epic"}) & \triangledown\text{-UNIQ} \\
f \mathbin{;} x = h \ \wedge\ g \mathbin{;} x = j & \Rightarrow & f \triangledown g \mathbin{;} x = h \triangledown j & \triangledown\text{-FUSION}
\end{array}$$

Law FUSION may be simplified to an unconditional law by substituting $h, j := f \mathbin{;} x,\ g \mathbin{;} x$,

$$f \triangledown g \mathbin{;} x = (f \mathbin{;} x) \triangledown (g \mathbin{;} x) \qquad \triangledown\text{-FUSION}$$

Similar simplifications will be done tacitly in the sequel. Notice that for given $f\colon a + b \to c$ the equation $x \triangledown y = f$ *defines* x and y, since the equation equivales by $\triangledown$-CHARN the two equations $x = \mathit{inl} \mathbin{;} f$ and $y = \mathit{inr} \mathbin{;} f$. We shall quite often use this form of definition.

14 Products. Products are, by definition, dual to sums. Let exl, exr be a product of a and b, supposing one exists; its common source is denoted $a \times b$. We abbreviate $[\![f,g \leftarrow exl, exr]\!]_{\wedge(a,b)}$ to just $f \vartriangle g$. (The usual categorical notation is $\langle f,g\rangle$).

$$f\colon c \to a \;\wedge\; g\colon c \to b \quad\Rightarrow\quad f \vartriangle g\colon c \to a \times b \qquad \vartriangle\text{-TYPE}$$

The laws for exl, exr and $\vartriangle$ work out as follows:

$$x \mathbin{;} exl = f \;\wedge\; x \mathbin{;} exr = g \quad\equiv\quad x = f \vartriangle g \qquad \vartriangle\text{-CHARN}$$
$$f \vartriangle g \mathbin{;} exl = f \;\wedge\; f \vartriangle g \mathbin{;} exr = g \qquad \vartriangle\text{-SELF}$$
$$exl \vartriangle exr = id \qquad \vartriangle\text{-ID}$$
$$x \mathbin{;} exl = y \mathbin{;} exl \;\wedge\; x \mathbin{;} exr = y \mathbin{;} exr \quad\Rightarrow\quad x = y \quad (\text{“jointly monic”}) \qquad \vartriangle\text{-UNIQ}$$
$$x \mathbin{;} f \vartriangle g = (x \mathbin{;} f) \vartriangle (x \mathbin{;} g) \qquad \vartriangle\text{-FUSION}$$

15 Application. As a first application we show that $inl_{a,a}$ is monic (and by symmetry $inr_{a,a}$ too, and dually each of $exl_{a,a}$ and $exr_{a,a}$ is epic):

$$\begin{array}{ll}
 & x = y \\
\equiv & \qquad \text{aiming at “ } ; inl \text{” after } x \text{ and } y\text{, use } \triangledown\text{-SELF}[f := id] \\
 & x \mathbin{;} inl \mathbin{;} id \triangledown g = y \mathbin{;} inl \mathbin{;} id \triangledown g \\
\Leftarrow & \qquad \text{Leibniz} \\
 & x \mathbin{;} inl = y \mathbin{;} inl
\end{array}$$

as desired. The choice for g is immaterial; id_a certainly does the job.

As a second application we show that $\triangledown$ and $\vartriangle$ abide. Two binary operations $\oplus$ and $\ominus$ **abide** with each other if: for all values a, b, c, d

$$(a \oplus b) \ominus (c \oplus d) \quad=\quad (a \ominus c) \oplus (b \ominus d)\,.$$

Writing $a \oplus b$ as $a \mid b$ and $a \ominus b$ as $\frac{a}{b}$, the equation reads

$$\frac{a \mid b}{c \mid d} \quad=\quad \frac{a}{c} \Big| \frac{b}{d}\,.$$

The term *abide* has been coined by Bird [10] and comes from “above-beside.” In category theory this property is called the ‘middle exchange rule.’

$$\begin{array}{ll}
 & (f \triangledown g) \vartriangle (h \triangledown j) = (f \vartriangle h) \triangledown (g \vartriangle j) \\
\equiv & \qquad \triangledown\text{-CHARN}\,[x, f, g := \text{lhs},\;\; f \vartriangle h,\;\; g \vartriangle j] \\
 & inl \mathbin{;} (f \triangledown g) \vartriangle (h \triangledown j) = f \vartriangle h \quad\wedge\quad inr \mathbin{;} (f \triangledown g) \vartriangle (h \triangledown j) = g \vartriangle j \\
\equiv & \qquad \vartriangle\text{-FUSION (at two places)} \\
 & (inl \mathbin{;} f \triangledown g) \vartriangle (inl \mathbin{;} h \triangledown j) = f \vartriangle h \quad\wedge\quad (inr \mathbin{;} f \triangledown g) \vartriangle (inr \mathbin{;} h \triangledown j) = g \vartriangle j \\
\equiv & \qquad \triangledown\text{-SELF (at four places)}
\end{array}$$

$$\begin{array}{ll} & f \vartriangle h = f \vartriangle h \;\; \wedge \;\; g \vartriangle j = g \vartriangle j \\ \equiv & \quad \text{equality} \\ & \textsf{true}. \end{array}$$

16 More laws for product and sum. For later use we define, for $f\colon\ a \to b$ and $g\colon\ c \to d$,

$$\begin{array}{lcllcl} f+g & = & (f \mathbin{;} \mathit{inl}) \triangledown (g \mathbin{;} \mathit{inr}) & : & a+c & \to & b+d \\ f\times g & = & (\mathit{exl} \mathbin{;} f) \vartriangle (\mathit{exr} \mathbin{;} g) & : & a\times c & \to & b\times d\,. \end{array}$$

These $+$ and $\times$ are bifunctors: $\mathit{id}+\mathit{id} = \mathit{id}$ and $f+g \mathbin{;} h+j = (f \mathbin{;} h)+(g \mathbin{;} j)$, and similarly for $\times$. Throughout the text we shall use several properties of product and sum. These are referred to by the hint 'product' or 'sum'. Here is a list.

$$\begin{array}{rcl@{\qquad}rcl}
f\times g \mathbin{;} \mathit{exl} & = & \mathit{exl} \mathbin{;} f & \mathit{inl} \mathbin{;} f+g & = & f \mathbin{;} \mathit{inl} \\
f \vartriangle g \mathbin{;} \mathit{exl} & = & f & \mathit{inl} \mathbin{;} f \triangledown g & = & f \\
f\times g \mathbin{;} \mathit{exr} & = & \mathit{exr} \mathbin{;} g & \mathit{inr} \mathbin{;} f+g & = & g \mathbin{;} \mathit{inr} \\
f \vartriangle g \mathbin{;} \mathit{exr} & = & g & \mathit{inr} \mathbin{;} f \triangledown g & = & g \\
f \mathbin{;} g \vartriangle h & = & (f \mathbin{;} g) \vartriangle (f \mathbin{;} h) & f \triangledown g \mathbin{;} h & = & (f \mathbin{;} h) \triangledown (g \mathbin{;} h) \\
\mathit{exl} \vartriangle \mathit{exr} & = & \mathit{id} & \mathit{inl} \triangledown \mathit{inr} & = & \mathit{id} \\
(h \mathbin{;} \mathit{exl}) \vartriangle (h \mathbin{;} \mathit{exr}) & = & h & (\mathit{inl} \mathbin{;} h) \triangledown (\mathit{inr} \mathbin{;} h) & = & h \\
f \vartriangle g \mathbin{;} h\times j & = & (f \mathbin{;} h) \vartriangle (g \mathbin{;} j) & f+g \mathbin{;} h \triangledown j & = & (f \mathbin{;} h) \triangledown (g \mathbin{;} j) \\
f\times g \mathbin{;} h\times j & = & (f \mathbin{;} h)\times(g \mathbin{;} j) & f+g \mathbin{;} h+j & = & (f \mathbin{;} h)+(g \mathbin{;} j) \\
f \vartriangle g = h \vartriangle j & \equiv & f=h \wedge g=j & f \triangledown g = h \triangledown j & \equiv & f=h \wedge g=j
\end{array}$$

Some of these are just the laws presented before.

2d Coequalisers and Kernel pairs

The equivalence relation induced by a given relation is an important concept in mathematics. Related to this is the lesser known concept of kernel pair. Both will be used in the construction of the congruence relation induced by a given relation, in Section 2f. Let us therefore present the algebraic properties of these concepts; they are categorically known as coequaliser and kernel pair.

17 Induced equivalence — coequaliser. Let $\mathcal{C}$ be $\mathcal{S}et$, the default category, and fix for the following discussion an object a and a parallel pair (f,g) with a as common source. The pair (f,g) is (or represents) a relation on a, namely the one that contains all pairs (fx, gx). We shall now describe the equivalence relation induced by (f,g).

Each function p with source a is (or represents) an equivalence on a, namely the one that contains all pairs (y,z) for which $py = pz$. An equivalence p on a is called **proper** if: p is a surjective function. Properness of p means that the target of p is precisely the set of equivalence classes (and does not contain unreachable junk).

$$\begin{tikzcd} \cdot \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & a \arrow[r, "p"] \arrow[dr, "q"'] & \cdot \arrow[d, "x"] \\ & & \cdot \end{tikzcd}$$

Observe that for an equivalence p

$$\begin{array}{ll} & p \text{ includes the relation } (f,g) \\ \equiv & \qquad \text{representation above, set theory} \\ & f \,;\, p \;=\; g \,;\, p \\ \equiv & \qquad \text{definition } \bigvee(f\|g) \\ & p \text{ is an object in } \bigvee(f\|g). \end{array}$$

And, for equivalences p, q

$$\begin{array}{ll} & p \text{ is included in } q \\ \equiv & \qquad \text{representation of equivalences, set theory} \\ & p \,;\, x \;=\; q \quad \text{for some function } x\colon\ \mathrm{tgt}p \to \mathrm{tgt}q \\ \equiv & \qquad \text{definition } \bigvee(f\|g) \\ & x\colon\ p \to q \text{ in } \bigvee(f\|g) \quad \text{for some } x. \end{array}$$

The equivalence on a **induced** by (f,g) is: a least, proper, equivalence including relation (f,g); least meaning being included in each equivalence that also includes (f,g). For an explicit expression of this notion in $\mathcal{S}et$, let

$$\begin{array}{lcl} R_{f,g} & = & \{(fx, gx) \mid\ x \in \mathrm{src} f (= \mathrm{src}\, g)\} \\ P & = & \bigcup n ::\ (R_{f,g} \cup R_{f,g}{}^{\cup})^n \end{array}$$

where for normal relations S and T on a

$$\begin{array}{lcl} S \,;\, T & = & \text{the usual composition of } S \text{ and } T \\ S^n & = & \text{the usual } n\text{-fold composition of } S \\ S \cup T & = & \text{the usual union of } S \text{ and } T \\ S^{\cup} & = & \text{the usual reverse of } S \end{array}$$

and

$$\begin{array}{lcl} a/{\cong} & = & \text{the usual set of } \cong\text{-equivalence classes of } a\,. \end{array}$$

Then the equivalence p induced by (f,g) is the function

$$\begin{array}{lcl} p & : & a \to a/P \\ p(x) & = & \text{the } P\text{-equivalence class in } a/P \text{ of } x\,. \end{array}$$

Alternatively, the induced equivalence can be expressed by initiality as follows. Let p be the equivalence induced by (f,g), and let u be such that $u \mathbin{;} p = id$ which exists since p is surjective. Let q be an arbitrary equivalence on a that also includes relation (f,g). Then, the initiality statement

$$x\colon p \to q \text{ in } \bigvee(f\|g) \qquad \equiv \qquad x = \text{some expression not involving x}$$

is established as follows.

$$\begin{array}{lll}
 & & x\colon p \to q \text{ in } \bigvee(f\|g) \\
 & \equiv & \quad \text{definition } \bigvee(f\|g) \\
 & & p \mathbin{;} x = q \\
(*) & \equiv & \quad \text{below: } q = p \mathbin{;} u \mathbin{;} q \\
 & & p \mathbin{;} x = p \mathbin{;} u \mathbin{;} q \\
 & \equiv & \quad \text{surjectivity } p \\
 & & x = u \mathbin{;} q.
\end{array}$$

For step $(*)$ we argue by extensionality. For each $x\colon \imath \to a$ (an element in a considered as a nullary function)

$$\begin{array}{ll}
 & x \mathbin{;} q = x \mathbin{;} p \mathbin{;} u \mathbin{;} q \\
\Leftarrow & \quad \text{equivalence } q \text{ includes relation } (f,g) \\
 & x \vartriangle (x \mathbin{;} p \mathbin{;} u) \in (R_{f,g} \cup R_{f,g}{}^{\cup})^n \quad \text{for some } n \\
= & \quad \text{above observation } p \approx P = \bigcup n :: (R_{f,g} \cup R_{f,g}{}^{\cup})^n \\
 & x \mathbin{;} p = x \mathbin{;} p \mathbin{;} u \mathbin{;} p \\
= & \quad \text{property } u \mathbin{;} p = id \\
 & \textsf{true}.
\end{array}$$

Thus p is initial in $\bigvee(f\|g)$.

Abstracting from $\mathcal{S}et$ and the application here, the initial object in $\bigvee(f\|g)$ is called coequaliser since in categories different from $\mathcal{S}et$ the terminology of relation, equivalence, and inclusion may not be appropriate. We shall present the properties of coequalisers in a way suitable for algebraic calculation.

18 Laws for coequalisers. Let $\mathcal{C}$ be arbitrary, the default category, and let (f,g) be a parallel pair. A **coequaliser** of (f,g) is: an initial object in $\bigvee(f\|g)$. Let p be a coequaliser of (f,g), supposing one exists. We write $p\backslash_{f,g}q$ or simply $p\backslash q$ instead of $(\!|p \to q|\!)_{\bigvee(f\|g)}$ since, as we shall explain, the fraction notation better suggests the calculational properties.

$$f \mathbin{;} q = g \mathbin{;} q \quad \Rightarrow \quad p\backslash q\colon \operatorname{tgt} p \to \operatorname{tgt} q \qquad \backslash\text{-TYPE}$$

Then the laws for p and $\backslash$ work out as follows.

$$p \mathbin{;} x = q \qquad \equiv \qquad x = p\backslash q \qquad \backslash\text{-CHARN}$$

$$
\begin{array}{llcll}
 & p \mathbin{;} p\backslash q = q & & & \backslash\text{-SELF} \\
 & id = p\backslash p & & & \backslash\text{-ID} \\
 & p \mathbin{;} x = q \;\wedge\; p \mathbin{;} y = q & \Rightarrow & x = y & \backslash\text{-UNIQ} \\
\text{i.e.,} & p \mathbin{;} x = p \mathbin{;} y & \Rightarrow & x = y & (p \text{ epic}) \\
 & q \mathbin{;} x = r & \Rightarrow & p\backslash q \mathbin{;} x = p\backslash r & \backslash\text{-FUSION} \\
\text{i.e.,} & p\backslash q \mathbin{;} x = p\backslash(q \mathbin{;} x) & & &
\end{array}
$$

In accordance with the convention explained in paragraph 10 we have omitted in laws \-CHARN, \-SELF and \-FUSION the well-formedness condition that q is an object in $\mathsf{V}(f\|g)$; the notation $...\backslash q$ is only senseful if $f \mathbin{;} q = g \mathbin{;} q$, like in arithmetic where the notation m/n is only senseful if n differs from 0. Notice also how \-FUSION simplifies to an unconditional fusion law. Similarly law \-UNIQ simplifies to the assertion that each coequaliser is epic.

Now that we have presented the laws the choice of notation may be evident: the usual manipulation of *cancelling adjacent factors in the denominator and nominator* is valid when composition is interpreted as multiplication and \ is interpreted as a fraction. (See also law \-COMPOSE below.) This may also help you to remember that there is only "post-fusion" here; the equation $x \mathbin{;} p\backslash q = (x \mathbin{;} p)\backslash q$ is not meaningful and not valid in general.

19 Additional laws. The following law confirms the choice of notation once more.

$$p\backslash q \mathbin{;} q\backslash r = p\backslash r \qquad \backslash\text{-COMPOSE}$$

Here is one way to prove it.

$$
\begin{array}{ll}
 & p\backslash q \mathbin{;} q\backslash r \\
= & \qquad \backslash\text{-FUSION} \\
 & p\backslash(q \mathbin{;} q\backslash r) \\
= & \qquad \backslash\text{-SELF} \\
 & p\backslash r.
\end{array}
$$

An interesting aspect is that the omitted subscripts to \ may differ: e.g., $p\backslash_{f,g} q$ and $q\backslash_{h,j} r$, and q is not necessarily a coequaliser of f, g. Rephrased in the standard notation, law \-COMPOSE reads:

$$(\!|a \to b|\!)_{\mathcal{A}} \mathbin{;} (\!|b \to c|\!)_{\mathcal{B}} = (\!|a \to c|\!)_{\mathcal{A}} \qquad \text{COMPOSE}$$

where $\mathcal{A}$ and $\mathcal{B}$ are full subcategories of some category $\mathcal{C}$ and objects b, c are in both $\mathcal{A}$ and $\mathcal{B}$; in our case $\mathcal{A} = \mathsf{V}(f\|g)$, $\mathcal{B} = \mathsf{V}(h\|j)$, and $\mathcal{C} = \mathsf{V}(d)$ where d is the common target of f, g, h, j. Then the proof runs as follows.

$$
\begin{array}{ll}
 & (\!|a \to b|\!)_{\mathcal{A}} \mathbin{;} (\!|b \to c|\!)_{\mathcal{B}} = (\!|a \to c|\!)_{\mathcal{A}} \\
\Leftarrow & \qquad \text{FUSION}
\end{array}
$$

$\llparenthesis b \rightharpoonup c \rrparenthesis_{\mathcal{B}}\colon\ b \rightarrow_{\mathcal{A}} c$

$\equiv$ both $\mathcal{A}$ and $\mathcal{B}$ are full subcategories of $\mathcal{C}$, each containing both b and c

$\llparenthesis b \rightharpoonup c \rrparenthesis_{\mathcal{B}}\colon\ b \rightarrow_{\mathcal{B}} c$

$\equiv$ SELF

true.

Another law that we shall use below has to do with functors. As before, let p be a coequaliser. Then

$$F(p\backslash q) = Fp\backslash Fq \qquad \backslash\text{-FCTR}$$

The implicit well-formedness condition here is that Fp is a coequaliser. Clearly, this condition is valid when F preserves coequalisers. The proof of the law reads:

$F(p\backslash q) = Fp\backslash Fq$

$\equiv$ $\backslash$-CHARN

$Fp\,;F(p\backslash q) = Fq$

$\equiv$ functor

$F(p\,;p\backslash q) = Fq$

$\equiv$ $\backslash$-SELF

true.

20 Induced relation — kernel pair. Above we have dealt with a categorical description of the equivalence p on a induced by a given relation (f,g): the coequaliser of (f,g). Now we consider inducing in the opposite direction. A relation (f,g) on a is called **proper** if: both function f and function g are injective. Let a set a and an equivalence p on a be fixed for this discussion.

$$\begin{tikzcd}
{} \arrow[d, "x"'] \arrow[rd, shift left, "{d,e}"] \arrow[rd, shift right] & & \\
{} \arrow[r, shift left] \arrow[r, shift right, "{f,g}"'] & a \arrow[r, "p"] & {}
\end{tikzcd}$$

Observe that for a relation (d,e) on a

(d,e) is included in p

$\equiv$ representation of relations and equivalences, set theory

$$\begin{array}{lll}
 & & d \,;\, p \;=\; e \,;\, p \\
(*) & \equiv & \quad \text{definition } \bigwedge(p \lrcorner p) \text{ below} \\
 & & (d,e) \text{ in } \bigwedge(p \lrcorner p).
\end{array}$$

And, for relations on a,

$$\begin{array}{lll}
 & & (d,e) \text{ is included in } (f,g) \\
 & \equiv & \quad \text{representation of relations, set theory} \\
 & & d \;=\; x \,;\, f \;\wedge\; e \;=\; x \,;\, g \quad \text{for some } x\text{: } \mathrm{src}(d,e) \to \mathrm{src}(f,g) \\
(*) & \equiv & \quad \text{definition } \bigwedge(p \lrcorner p) \text{ below} \\
 & & x\text{: } (d,e) \to (f,g) \text{ in } \bigwedge(p \lrcorner p) \quad \text{for some } x.
\end{array}$$

Before defining the induced relation, let us first define category $\bigwedge(p \lrcorner p)$ used above in steps $(*)$. This category is designed in such a way that the steps become valid; so it is built upon $\mathcal{C}$ as follows. An object in $\bigwedge(p \lrcorner p)$ is: a pair (f,g) of morphisms in $\mathcal{C}$ satisfying $f \,;\, p = g \,;\, p$ (it follows that f,g is a parallel pair with a as common target). A morphism from (d,e) to (f,g) in $\bigwedge(p \lrcorner p)$ is: a morphism x in $\mathcal{C}$ satisfying $d = x \,;\, f$ and $e = x \,;\, g$.

Now, the relation on a **induced** by p is: a greatest, proper, relation on a included in p; greatest meaning including each relation that is included in p. This is a lesser known concept in daily set theory, since in set theory a relation is rarely represented as a pair (f,g) of functions, and moreover the relation induced by p represents the very same relation as p. In $\mathcal{S}et$ the relation induced by p is $(\mathit{exl}, \mathit{exr})$ with common source $\{(x,y) |\; p(x) = p(y)\}$.

Alternatively, the relation induced by p can be expressed by finality as follows. Let (f,g) be the relation induced by p, and let (d,e) be an arbitrary relation including p. Then the finality statement

$$x\text{: } (d,e) \to (f,g) \text{ in } \bigwedge(p \lrcorner p) \qquad \equiv \qquad x = \text{some expression not involving } x$$

is readily established.

$$\begin{array}{ll}
 & x\text{: } (d,e) \to (f,g) \text{ in } \bigwedge(p \lrcorner p) \\
\equiv & \quad \text{definition } \bigwedge(p \lrcorner p) \\
 & d = x \,;\, f \quad \wedge \quad e = x \,;\, g \\
\equiv & \quad \text{definition } (f,g)\text{, observation above} \\
 & d = x \,;\, \mathit{exl} \quad \wedge \quad e = x \,;\, \mathit{exr} \\
\equiv & \quad \text{(cartesian) product} \\
 & x = d \vartriangle e.
\end{array}$$

Thus (f,g) is final in $\bigwedge(p \lrcorner p)$.

Abstracting from $\mathcal{S}et$ and the application here, the final object in $\bigwedge(p \lrcorner p)$ is called the kernel pair for p.

21 Laws for kernel pairs. Let $\mathcal{C}$ be arbitrary, the default category, and let p be a morphism. A **kernel pair** of p is: a final object in $\bigwedge(p \lrcorner p)$. Let (f,g) be a kernel pair of p, supposing one exists. This time we use the notation $(d,e)/_p(f,g)$ or simply $(d,e)/(f,g)$ instead of $[\![d,e \to f,g]\!]_{\bigwedge(p \lrcorner p)}$.

$$d \mathbin{;} p = e \mathbin{;} p \quad \Rightarrow \quad (d,e)/(f,g)\colon\ \operatorname{src} d \to \operatorname{src} f \ = \ \operatorname{src} e \to \operatorname{src} g \qquad /\text{-TYPE}$$

Then the laws for (f,g) and $/$ work out as follows.

$$d = x \mathbin{;} f \ \wedge\ e = x \mathbin{;} g \quad \equiv \quad x = (d,e)/(f,g) \qquad /\text{-CHARN}$$

$$d = (d,e)/(f,g) \mathbin{;} f \ \wedge\ e = (d,e)/(f,g) \mathbin{;} g \qquad /\text{-SELF}$$

$$id = (f,g)/(f,g) \qquad /\text{-ID}$$

$$\left.\begin{array}{l} d = x \mathbin{;} f \ \wedge\ e = x \mathbin{;} g \\ d = y \mathbin{;} f \ \wedge\ e = y \mathbin{;} g \end{array}\right\} \quad \Rightarrow \quad x = y \qquad /\text{-UNIQ}$$

i.e., $$x \mathbin{;} f = y \mathbin{;} f \ \wedge\ x \mathbin{;} g = y \mathbin{;} g \quad \Rightarrow \quad x = y \qquad \text{(jointly monic)}$$

$$x \mathbin{;} (d,e)/(f,g) = (x \mathbin{;} d,\ x \mathbin{;} e)/(f,g) \qquad /\text{-FUSION}$$

$$(d,e)/(f,g) \mathbin{;} (f,g)/(h,j) = (d,e)/(h,j) \qquad /\text{-COMPOSE}$$

$$F((d,e)/(f,g)) = F(d,e)/F(f,g) \qquad /\text{-FCTR}$$

Notice that there is "pre-fusion" only. Due to the presence of so many pairs the notation is a bit cumbersome, but we refrain from simplifying it here. (We do so in paragraph 35.)

22 Application. As an example of the use of the laws we prove that the coequaliser and kernel pair form an adjunction. More precisely, let C denote a mapping that sends each parallel pair with common target a to some coequaliser of it, and similarly let K send each morphism with source a to some kernel pair of it:

$$\begin{array}{lcll} C(f,g) & = & \text{'the' coequaliser of } f,g & \text{for } (f,g) \text{ in } \bigwedge(a,a) \\ Kp & = & \text{'the' kernel pair of } p & \text{for } p \text{ in } \bigvee(a)\,. \end{array}$$

We shall extend them to functors $C\colon \bigwedge(a,a) \to \bigvee(a)$ and $K\colon \bigvee(a) \to \bigwedge(a,a)$, and then prove that they form an adjunction.

To define Cx for a morphism x in $\bigwedge(a,a)$ we make an obvious choice.

(a) $$Cx \ = \ C(d,e)\backslash C(f,g) \quad \text{for } x\colon (d,e) \to (f,g) \text{ in } \bigwedge(a,a)\,.$$

It remains to prove that C is a functor. Since in general $p\backslash q\colon p \to q$ (in the appropriate category, see $\backslash$-TYPE), it is immediate that Cx above has the right type, namely $C(d,e) \to C(f,g)$ in $\bigvee(a,a)$. The two functor axioms $C id = id$ and $C(x \mathbin{;} y) = Cx \mathbin{;} Cy$ follow immediately by $\backslash$-ID and $\backslash$-COMPOSE.

To define Ku for a morphism u in $\bigvee(a)$ we make an obvious choice too.

(b) $$Ku \ = \ Kp/Kq \quad \text{for } u\colon p \to q \text{ in } \bigvee(a)\,.$$

Thus extended, K is a functor by a similar argument as above.

To prove that C is adjoint to K we establish natural transformations $\varepsilon\colon CK \dot{\to} I$ and $\eta\colon I \dot{\to} KC$ such that $\eta K \,;\, K\varepsilon = idK$ and $C\eta \,;\, \varepsilon C = idC$. Take

$$\varepsilon q \quad = \quad CKq\backslash q \;:\; CKq \to_{\bigvee(a)} q \qquad \text{for all } q \text{ in } \bigvee(a)\,.$$

The naturality of ε is shown as follows. For arbitrary $u\colon p \to_{\bigvee(a)} q$,

$CKu \,;\, \varepsilon q$
$=$ definition C, K and ε, noting that $u\colon p \to_{\bigvee(a)} q$
$CKp\backslash CKq \,;\, CKq\backslash q$
$=$ $\backslash$-COMPOSE
$CKp\backslash q$
$=$ equation "$u\colon p \to_{\bigvee(a)} q$"
$CKp\backslash(p \,;\, u)$
$=$ $\backslash$-FUSION
$CKp\backslash p \,;\, u$
$=$ definition ε and I
$\varepsilon p \,;\, Iu$

as desired. Further we take

$$\eta(d,e) \quad = \quad (d,e)/KC(d,e) \;:\; (d,e) \to_{\bigwedge(a,a)} KC(d,e)\,.$$

We omit the proof that η is natural; this is quite similar (but not categorically dual) to the naturality of ε. Next we show that $\eta K \,;\, K\varepsilon = idK$. Let q be arbitrary, then

$(\eta K \,;\, K\varepsilon)q$
$=$ composition of natural transformations, and definitions of η, ε
$Kq/KCKq \,;\, K(CKq\backslash q)$
$=$ definition K (see (b) above with $u, p, q := q\backslash CKq, q, CKq$,
noting that $u\colon p \to q$ follows from $\backslash$-SELF)
$K(q\backslash CKq) \,;\, K(CKq\backslash q)$
$=$ functor, $\backslash$-COMPOSE
$K(q\backslash q)$
$=$ $\backslash$-ID, noting that id_q in $\bigvee(a)$ is $id_{\mathrm{tgt}q}$ in C
$K(id_q)$
$=$ functor
$idKq$.

The proof of $C\eta \,;\, \varepsilon C = idC$ is again quite similar to the above one.

2e Colimits

An initial object is a colimit of the empty diagram, and conversely, a colimit of a diagram is an initial object in the category of cocones over that diagram. Let us use the latter approach to present the algebraic properties of colimits.

23 A first description. A **diagram** in category $\mathcal{C}$ is: a directed graph D whose edges are labelled with morphisms of $\mathcal{C}$ in a way that is consistent with the typing in $\mathcal{C}$, that is, f is followed by g in D only if $f \mathbin{;} g$ makes sense. Category $\bigvee D$, built upon $\mathcal{C}$, is defined as follows. An object in $\bigvee D$, called **cocone** for D, is: an object c together with a collection γ of morphisms $\gamma_a\colon a \to c$ (one for each node a in D), satisfying for each edge $f\colon a \to b$ in diagram D:

$$\gamma_a \quad = \quad f \mathbin{;} \gamma_b \qquad \text{“commutativity of the triangle”}\,.$$

Object c is called the target of γ. Let γ and δ be cocones for D; then a morphism from γ to δ in $\bigvee D$ is: a morphism x in $\mathcal{C}$ satisfying

$$\forall(a \text{ in } D :: \quad \gamma_a \mathbin{;} x = \delta_a)\,.$$

A **colimit** for D in $\mathcal{C}$ is: an initial object in $\bigvee D$. Let γ be a colimit for D, supposing it exists. We write $\gamma\backslash_D\delta$ or simply $\gamma\backslash\delta$, instead of $(\!|\gamma \to \delta|\!)_{\bigvee D}$.

$$\forall(a \text{ in } D :: \quad \gamma_a \mathbin{;} x = \delta_a) \quad \Rightarrow \quad \gamma\backslash\delta\colon \operatorname{tgt}\gamma \to \operatorname{tgt}\delta \qquad \backslash\text{-TYPE}$$

Then the laws for γ and $\backslash$ work out as follows (a ranges over the nodes of D).

$$\begin{array}{lcll}
\forall(a :: \quad \gamma_a \mathbin{;} x = \delta_a) & \equiv & x = \gamma\backslash\delta & \backslash\text{-CHARN} \\
\forall(a :: \quad \gamma_a \mathbin{;} \gamma\backslash\delta = \delta_a) & & & \backslash\text{-SELF} \\
id = \gamma\backslash\gamma & & & \backslash\text{-ID} \\
\forall(a :: \quad \gamma_a \mathbin{;} x = \gamma_a \mathbin{;} y) & \Rightarrow & x = y \quad (\gamma \text{ jointly epic}) & \backslash\text{-UNIQ} \\
\gamma\backslash\delta \mathbin{;} x = \gamma\backslash\{a :: \quad \delta_a \mathbin{;} x\} & & & \backslash\text{-FUSION} \\
\gamma\backslash\delta \mathbin{;} \delta\backslash\varepsilon = \gamma\backslash\varepsilon & & & \backslash\text{-COMPOSE} \\
F(\gamma\backslash\delta) = F\gamma\backslash F\delta & & & \backslash\text{-FCTR}
\end{array}$$

for each D-cocone δ, ε (where, as usual, δ and $F\gamma$ are assumed to be colimits when they occur as the left argument of $\backslash$).

24 Improved description. In view of the explicit quantifications the above laws for colimits are not very suited for *algebraic calculation*, and that is what we are after. A lot of explicit quantifications are eliminated by treating a cocone as a family of functions, and defining for example $\gamma \mathbin{;} x = \delta$ to mean $\forall(a :: \quad \gamma_a \mathbin{;} x = \delta_a)$. It turns out that this can be formulated categorically by using natural transformations, which are families of morphisms indeed. Several (not all) manipulations on the subscripts can then be phrased as well-known manipulations with natural transformations as a whole. So let us redesign the definitions. (I got the suggestion from Jaap van der Woude; Mac Lane [41] and Lambek and Scott [37] and several others use the following formulation too.)

As regards the property of being a cocone we can say without loss of generality that a directed graph is a category $\mathcal{D}$: take all finite pathes of the edges as morphisms. (Conversely, each category $\mathcal{D}$ determines a graph by taking all morphisms as edges, and forgetting which morphisms are composites and which are identities.) A labelling of the edges with morphism from $\mathcal{C}$ is then a functor D: $\mathcal{D} \to \mathcal{C}$. This leads to the following definitions. A **diagram** in $\mathcal{C}$ is: a functor D: $\mathcal{D} \to \mathcal{C}$, for some category $\mathcal{D}$, called the **shape** of the diagram. Category $\bigvee D$ is built upon $\mathcal{C}$ as follows. An object in it, again called **cocone** for D, is: a natural transformation γ: $D \dot{\to} \underline{c}$ for some object c in $\mathcal{C}$ ($\underline{c}$ is the constant functor determined by c). Let γ and δ be cocones for D; then a morphism from γ to δ in $\bigvee D$ is: a morphism x in $\mathcal{C}$ satisfying $\gamma\,;\,x = \delta$ (the composition is a slight adaptation of the one in $\mathcal{C}$; see paragraph 25 below). Again, a **colimit** for D is: an initial object in $\bigvee D$.

The required "commutativity of the triangles" follows from the naturality: for each Df: $Da \to Db$ in the 'diagram' $D\mathcal{D}$ in $\mathcal{C}$

$$\begin{array}{llllll} \gamma a\,;\,\underline{c}f & = & Df\,;\,\gamma b & : & Da \to \underline{c}a & \text{that is,} \\ \gamma a & = & Df\,;\,\gamma b & : & Da \to c\,. & \end{array}$$

25 Defns for ntrfs. For natural transformations in general, hence for cocones in particular, the following definitions are standard. For γ: $D \dot{\to} \underline{c}$ and δ: $D \dot{\to} \underline{d}$:

- for each x: $c \to d$,
  $\gamma\,;\,x = \lambda(a ::\ \gamma a\,;\,x)$: $D \dot{\to} \underline{d}$ is a cocone for D again.
- for each functor F: $\mathcal{C} \to \mathcal{C}$,
  $F\gamma = \lambda(a ::\ F(\gamma a))$: $FD \dot{\to} F\underline{c}$ is a cocone for FD (note that $F\underline{c} = \underline{Fc}$).
  If in addition F preserves colimits, then $F\gamma$ is a colimit for FD if γ is so for D.
  Since by definition $(F\gamma)a = F(\gamma a)$ we omit the parentheses.
- for each functor S: $\mathcal{D} \to \mathcal{D}$,
  $\gamma S = \lambda(a ::\ \gamma(Sa))$: $DS \dot{\to} \underline{c}$ is a cocone for DS (note that $\underline{c}S = \underline{c}$).
  If S transforms the shape, γS is the transformed cocone.
  Since by definition $\gamma(Sa) = (\gamma S)a$ we omit the parentheses.

26 The laws. Let γ be a colimit for D. Then the laws for γ and $\backslash$ work out as follows.

$$\delta\colon D \dot{\to} \underline{d} \quad\Rightarrow\quad \gamma\backslash\delta\colon \operatorname{tgt}\gamma \to d \qquad \backslash\text{-TYPE}$$

and

$$\gamma\,;\,x = \delta \quad\equiv\quad x = \gamma\backslash\delta \qquad \backslash\text{-CHARN}$$

$$\gamma\,;\,\gamma\backslash\delta = \delta \qquad \backslash\text{-SELF}$$

$\gamma\backslash\gamma = id$ \-ID

$\gamma \mathbin{;} x = \gamma \mathbin{;} y \quad \Rightarrow \quad x = y \quad (\gamma \text{ jointly epic})$ \-UNIQ

$\gamma\backslash\delta \mathbin{;} x = \gamma\backslash(\delta \mathbin{;} x)$ \-FUSION

$\gamma\backslash\delta \mathbin{;} \delta\backslash\varepsilon = \gamma\backslash\varepsilon$ \-COMPOSE

$F(\gamma\backslash\delta) = F\gamma\backslash F\delta$ \-FCTR

for each D-cocone δ, ε (δ and $F\gamma$ being a colimit when occurring as the left argument of \.) Notice also that, by definition of γS and \-SELF,

$$\gamma S \mathbin{;} \gamma\backslash\delta \quad = \quad (\gamma \mathbin{;} \gamma\backslash\delta)S \quad = \quad \delta S\,.$$

If γS is a colimit, then $\gamma S\backslash\delta S$ is well-formed and the equality $\gamma\backslash\delta = \gamma S\backslash\delta S$ follows by \-CHARN from the equation.

27 Application. We present the well-known construction of an initial F-algebra. You may skip this application without loss of continuity. Our interest is *solely* in the algebraic, calculational style of various subproofs. The notion of F-algebra has been defined in paragraph 7 without any explanation. So you may postpone reading this application until you've read Chapter 3 and know what algebras are good for. The construction will require that $\mathcal{C}$ has an initial object and a colimit for each ω-chain, and that functor F preserves colimits of ω-chains; briefly: $\mathcal{C}$ is an ω-category and F is ω-cocontinuous.

Given endofunctor F we wish to construct an F-algebra, $\alpha\colon Fa \to a$ say, that is initial in $\mathcal{A}lg(F)$. Anticipating the rather easily proven fact that an initial F-algebra $\alpha\colon Fa \to a$ is an isomorphism $\alpha\colon Fa \cong a$ (see paragraph 3.31), we derive a construction of an $\alpha\colon Fa \to a$ as follows. (Read the steps and their explanation below in parallel!)

$\alpha\colon Fa \to a$

(a) $\Leftarrow$ definition isomorphism

$\alpha\colon Fa \simeq a$

(b) $\Leftarrow$ definition cocone morphism (taking $a = \text{tgt}\gamma = \text{tgt}\gamma S$)

$\alpha\colon F\gamma \simeq \gamma S$ in $\bigvee(FD) \quad \wedge \quad FD = DS$

(c) $\equiv$ $F\gamma$ is colimit for FD (taking $\alpha = F\gamma\backslash\gamma S$)

γS is colimit for $DS \quad \wedge \quad FD = DS.$

Step (a): this is motivated by the wish that α be initial in $\mathcal{A}lg(F)$, and so α will be an isomorphism; in other words, in view of the required initiality the step is *no* strengthening. *Step* (b): here we merely decide that α, a come from a (co)limit construction; this is true for many categorical constructions. So we aim at $\alpha\colon F\gamma \cong \ldots$, where γ is 'the' colimit (which we assume to exist) for a diagram D yet to be defined. Since $F\gamma$ is a FD-cocone, there has to be another FD-cocone on the dots. To keep things simple, we aim at an FD-cocone constructed from γ, say γS, where S is an endofuctor on src$D$. Since γS is evidently a DS-cocone, and must be an FD-cocone, it follows that $FD = DS$ is another

requirement.
Step (c): the hint '$F\gamma$ is colimit for FD' follows from the assumption that F preserves colimits, and the definition $\alpha = F\gamma\backslash\gamma S$ is *forced* by (the proof of) the uniqueness of initial objects. (It is indeed very easy to verify that $F\gamma\backslash\gamma S$ and $\gamma S\backslash F\gamma$ are each other's inverse.)
We shall now complete the construction in the following three parts.

1. Construction of D, S such that $FD = DS$.
2. Proof of 'γS is colimit for DS' where γ is a colimit for D.
3. Proof of 'α is initial in $\mathcal{A}lg(F)$' where $\alpha = F\gamma\backslash\gamma S$.

28 *Part 1.* (Construction of D, S such that $FD = DS$.) The requirement $FD = DS$ says that FD is a 'subdiagram' of D. This is easily achieved by making D a *chain* of iterated F applications, as follows.

Let ω be the category with objects $0, 1, 2, \ldots$ and a unique arrow from i to j (denoted $i{\le}j$) for every $i \le j$. So ω is the shape of a chain. The zero and successor functors $0, S$: $\omega \to \omega$ are defined by $0(i{\le}j) = 0{\le}0$ and $S(i{\le}j) = (i{+}1){\le}(j{+}1)$.

Let o be an initial object in $\mathcal{C}$. Define the diagram D: $\omega \to \mathcal{C}$ by $D(i{\le}j) = F^i(\!|F^{j-i}o|\!)$, where $(\!|\_|\!)$ abbreviates $(\!|o \to \_|\!)_{\mathcal{C}}$. It is quite easy to show that D is a functor, that is, $D(i{\le}j\ ;\ j{\le}k) = D(i{\le}j)\ ;\ D(j{\le}k)$. It is also immediate that $FD = DS$, since

$$FD(i{\le}j) \;=\; FF^i(\!|F^{j-i}o|\!) \;=\; F^{i+1}(\!|F^{(j+1)-(i+1)}o|\!) \;=\; D((i{+}1){\le}(j{+}1)) \;=\; DS(i{\le}j)$$

Thanks to the particular form of ω, natural transformations of the form ε: $D \dot{\to} G$ (some G) can be defined by induction, that is, by defining

$$\begin{array}{lcll} \varepsilon 0 & : & D0 \dot{\to} G0 & \text{or, equivalently} \quad \varepsilon 0\colon D0 \to G0 \\ \varepsilon S & : & DS \dot{\to} GS\,. & \end{array}$$

We shall use this form of definition in Part 2 and Part 3 below.

29 *Part 2.* (Proof of 'γS is colimit for DS' where γ is a colimit for D.) Our task is to construct for arbitrary cocone δ: $DS \dot{\to} \underline{d}$ a morphism $(\!|\gamma S \to \delta|\!)_{\mathsf{V}(DS)}$ such that

$$(\spadesuit) \qquad \gamma S\ ;\ x = \delta \quad\equiv\quad x = (\!|\gamma S \to \delta|\!)_{\mathsf{V}(DS)}\,.$$

Our guess is that $\gamma\backslash\varepsilon$ may be chosen for $(\!|\gamma S \to \delta|\!)_{\mathsf{V}(DS)}$ for some suitably chosen ε: $D \dot{\to} \underline{d}$ that depends on δ. This guess is sufficient to start the proof of ($\spadesuit$); we shall derive a definition of ε (more specifically, for $\varepsilon 0$ and εS) along the way.

$$\begin{array}{cl} & x = \gamma\backslash\varepsilon \\ \equiv & \quad \backslash\text{-CHARN} \end{array}$$

$\gamma \mathbin{;} x = \varepsilon$

$\equiv$ observation at the end of Part 1

$(\gamma \mathbin{;} x)0 = \varepsilon 0 \;\;\wedge\;\; (\gamma \mathbin{;} x)S = \varepsilon S$

$\equiv$ 'standard definition' for natural transformations (see paragraph 25)

$\gamma 0 \mathbin{;} x = \varepsilon 0 \;\;\wedge\;\; \gamma S \mathbin{;} x = \varepsilon S$

$\equiv$ { aiming at the left hand side of (♠) }
define $\varepsilon S = \delta$ (noting that $\delta\colon DS \dashrightarrow \underline{d} = DS \dashrightarrow \underline{d}S$)

$\gamma 0 \mathbin{;} x = \varepsilon 0 \;\;\wedge\;\; \gamma S \mathbin{;} x = \delta$

(∗) $\equiv$ define $\varepsilon 0$ below such that $\gamma S \mathbin{;} x = \delta \;\Rightarrow\; \gamma 0 \mathbin{;} x = \varepsilon 0$ for all x

$\gamma S \mathbin{;} x = \delta$.

In order to define $\varepsilon 0$ satisfying the requirement derived at step (∗), we calculate

$\gamma 0 \mathbin{;} x$

$=$ { anticipating next steps, introduce an identity }

$\gamma 0 \mathbin{;} \underline{c}(0 \leq 1) \mathbin{;} x$

$=$ naturality γ ("commutativity of the triangle")

$D(0 \leq 1) \mathbin{;} \gamma 1 \mathbin{;} x$

$=$ using $\gamma S \mathbin{;} x = \delta$

$D(0 \leq 1) \mathbin{;} \delta 0$

so that we can fulfill the requirement $\gamma 0 \mathbin{;} x = \varepsilon 0$ by defining $\varepsilon 0 = D(0 \leq 1) \mathbin{;} \delta 0$.

30 *Part 3.* (Proof of 'α is initial in $\mathcal{A}lg(F)$' where $\alpha = F\gamma \backslash \gamma S$.) Put $a = \mathrm{tgt}\alpha = \mathrm{tgt}\gamma$ (as we did in the main steps (a), (b), (c) at the start). Let $\varphi\colon Fb \to b$ be arbitrary. We have to construct a morphism $(\!|\alpha \to \varphi|\!)_F\colon a \to b$ in $\mathcal{C}$ such that

(♣) $\quad F\gamma \backslash \gamma S \mathbin{;} x = Fx \mathbin{;} \varphi \quad\equiv\quad x = (\!|\alpha \to \varphi|\!)_F$.

Our guess is that the required morphism $(\!|\alpha \to \varphi|\!)_F$ can be writt en as $\gamma \backslash \delta$ for some suitably chosen D-cocone δ. This guess is sufficient to start the proof of (♣), deriving a definition for δ (more specifically, for $\delta 0$ and δS) along the way.

$F\gamma \backslash \gamma S \mathbin{;} x = Fx \mathbin{;} \varphi$

$\equiv$ $\backslash$-FUSION

$F\gamma \backslash (\gamma S \mathbin{;} x) = Fx \mathbin{;} \varphi$

$\equiv$ $\backslash$-CHARN$[\gamma, \delta, x := F\gamma,\ \gamma S \mathbin{;} x,\ Fx \mathbin{;} \varphi]$

$F\gamma \mathbin{;} Fx \mathbin{;} \varphi = \gamma S \mathbin{;} x$

$\equiv$ lhs: functor, rhs: 'standard definition' for ntrf (see paragraph 25)

$F(\gamma \mathbin{;} x) \mathbin{;} \varphi = (\gamma \mathbin{;} x)S$

(∗) $\equiv$ explained and proved below (defining δ)

$$\begin{array}{ll} & \gamma \mathbin{;} x = \delta \\ \equiv & \qquad \backslash\text{-CHARN} \\ & x = \gamma\backslash\delta. \end{array}$$

Arriving at the line above $(*)$ I see no way to make progress except to work bottom-up from the last line. Having the lines above and below $(*)$ available, we define δSn in terms of δn by

$$\delta S \quad = \quad F\delta \mathbin{;} \varphi,$$

a definition that is also suggested by type considerations alone. Now part $\Leftarrow$ of equivalence $(*)$ is immediate:

$$\begin{array}{ll} & F(\gamma \mathbin{;} x) \mathbin{;} \varphi = (\gamma \mathbin{;} x)S \\ \Leftarrow & \qquad \text{definition } \delta S\text{: } F\delta \mathbin{;} \varphi = \delta S \\ & \gamma \mathbin{;} x = \delta. \end{array}$$

For part $\Rightarrow$ of equivalence $(*)$ we argue as follows, assuming the line above $(*)$ as a premise, and defining $\delta 0$ along the way.

$$\begin{array}{ll} & \gamma \mathbin{;} x = \delta \\ \equiv & \qquad \text{induction principle} \\ & (\gamma \mathbin{;} x)0 = \delta 0 \quad \wedge \quad \forall(n :: \ \ (\gamma \mathbin{;} x)n = \delta n \ \ \Rightarrow \ \ (\gamma \mathbin{;} x)Sn = \delta Sn) \\ \equiv & \qquad \text{proved below: the ‘induction base’ in (i), and the ‘induction step’ in (ii)} \\ & \textsf{true}. \end{array}$$

For (i), the induction base, we calculate

$$\begin{array}{ll} & \gamma 0 \mathbin{;} x \\ = & \qquad \text{CHARN, using } \gamma 0\text{: } \mathit{0} \to c \\ & (\!|a|\!)_C \mathbin{;} x \\ = & \qquad \text{FUSION, using } x\text{: } a \to b \\ & (\!|b|\!)_C \\ = & \qquad \textbf{define } \delta 0 = (\!|b|\!)_C \\ & \textsf{true}. \end{array}$$

And for (ii), the induction step, we calculate for arbitrary n, using the induction hypothesis $(\gamma \mathbin{;} x)n = \delta n$,

$$\begin{array}{ll} & (\gamma \mathbin{;} x)Sn \\ = & \qquad \text{line above } (*) \\ & (F(\gamma \mathbin{;} x) \mathbin{;} \varphi)n \\ = & \qquad \text{hypothesis } (\gamma \mathbin{;} x)n = \delta n \end{array}$$

$$\begin{array}{ll} & (F\delta \mathbin{;} \varphi)n \\ = & \quad \text{definition } \delta S \\ & (\delta S)n \end{array}$$

as desired. This completes the entire construction and proof.

2f Induced congruence categorically

This section may be skipped without loss of continuity; the remainder of the text is independent of the notions and theorems presented here. We include it mainly to illustrate once more an algebraic calculational approach to category theory, in particular in a case where pushouts are involved. I wouldn't dare to claim that the approach presented here is the best one when dealing with pushouts. I consider it rather a case study. Although you should be able to follow the calculations step by step, you will probably not understand what is going on if you are not familiar with the notions of pushout and colimit.

We start with a categorical description of two different notions of induced congruence, then we introduce a notation that facilitates an algebraic calculation with pushouts, and finally we give a construction of one of the induced congruences and its correctness proof. The notions and notations of the preceding sections are used throughout.

31 Induced congruence categorically. Let functor F, F-algebra $\varphi\colon Fa \to a$, and object (f,g) in $\bigwedge(a,a)$ be given, and fixed throughout the following. Recall from Section 2d the notion of equivalence.

Aiming at a formulation in $\mathcal{A}lg(F)$ the following definition suggests itself. An alg-**congruence** for φ is: an F-homomorphism from φ to another F-algebra, that is, an object in $\bigvee_F(\varphi)$ (where $\bigvee_F$ abbreviates $\bigvee_{\mathcal{A}lg(F)}$). The alg-congruence for φ **induced** by (f,g) is: an initial object in $\bigvee_F(\varphi) \cap \bigvee(f\|g)$. (The intersection makes sense since both categories are subcategories of another one, namely $\bigvee(a)$.) Notice the close analogy with the equivalence on a induced by (f,g) (the coequaliser), which is an initial object in $\bigvee(a) \cap \bigvee(f\|g)$. The analogy may be exploited in generalising a construction of coequaliser to a construction of the induced alg-congruence. This has been done by Lehmann [39].

However, the underlying category $\mathcal{C}$, and not $\mathcal{A}lg(F)$, is the universe of discourse. The morphisms of $\mathcal{C}$ are —for us— all the algorithms that exist, and only some of these are in $\mathcal{A}lg(F)$ too. So, here is my self-made definition directly in terms of $\mathcal{C}$. A base-**congruence** for φ is: an object in $\mathcal{C}ongr(\varphi)$. Category $\mathcal{C}ongr(\varphi)$ is the full subcategory of $\bigvee(a)$ containing those equivalences p on a that satisfy

32 $$x \mathbin{;} Fp = y \mathbin{;} Fp \quad \Rightarrow \quad x \mathbin{;} \varphi \mathbin{;} p = y \mathbin{;} \varphi \mathbin{;} p$$

for all x, y. That is, 'componentwise' equivalent arguments x, y are mapped by φ to equivalent results. The base-congruence for φ **induced** by (f,g) is: an initial object in $\mathcal{C}ongr(\varphi) \cap \bigvee(f\|g)$. For later use we rephrase this as follows. Morphism p is the base-congruence for φ induced by (f,g) iff it is in $\mathcal{C}ongr(\varphi) \cap \bigvee(f\|g)$, and for each q in that category there exists a morphism, which we shall denote $p\backslash q$, such that

33 $x\colon p \to_{\bigvee(a)} q$ (meaning $p \mathbin{;} x = q$) $\equiv$ $x = p\backslash q$ congruence-CHARN

Here is a relationship between the two notions of congruences. As regards clause (ii) of the lemma, notice that in $\mathcal{S}et$ an u satisfying $p \mathbin{;} u \mathbin{;} p = id$ exists for every p (by the axiom of choice: define $u(y)$ to be some x for which $p(x) = y$ if such an x exists, and arbitrary otherwise.)

34 Lemma.
(a) *p is alg-congruence for φ $\Rightarrow$ p is base-congruence for φ.*
(b) *the converse of (i) is true if there exists an u for which $p \mathbin{;} u \mathbin{;} p = id$.*

Proof. (a) Let x, y be arbitrary. Then

$x \mathbin{;} \varphi \mathbin{;} p = y \mathbin{;} \varphi \mathbin{;} p$
$\equiv$ p is a homomorphism from φ, say $p\colon \varphi \to_F \psi$
$x \mathbin{;} Fp \mathbin{;} \psi = y \mathbin{;} Fp \mathbin{;} \psi$
$\Leftarrow$ Leibniz
$x \mathbin{;} Fp = y \mathbin{;} Fp$

as desired.
(b) Let u be such that $p \mathbin{;} u \mathbin{;} p = id$. We show that p is a homomorphism from φ to another algebra ψ that is yet to be constructed.

$p\colon \varphi \to_F \psi$
$\equiv$ definition $\to_F$
$\varphi \mathbin{;} p = Fp \mathbin{;} \psi$
$\equiv$ aiming at the hint of the next step, **define** $\psi = \chi \mathbin{;} \varphi \mathbin{;} p$
$\varphi \mathbin{;} p = Fp \mathbin{;} \chi \mathbin{;} \varphi \mathbin{;} p$
$\Leftarrow$ p is base-congruence for φ (taking $x, y := id, (Fp \mathbin{;} \chi)$ in formula 32)
$id \mathbin{;} Fp = Fp \mathbin{;} \chi \mathbin{;} Fp$
$\equiv$ **define** $\chi = Fu$, functor, property u
true.

When the u is a post-inverse, the premise that p is a base-congruence for φ is not needed, since with $\psi := Fu \mathbin{;} \varphi \mathbin{;} p$ the second step of the above calculation already reduces to true. (It is needed that $\operatorname{src} p = \operatorname{tgt} \varphi$.)

When the u is a pre-inverse, the target algebra of the homomorphism p is independent of the choice for a pre-inverse of p: if both $u \mathbin{;} p = id = v \mathbin{;} p$, then, by formula 32, $Fu \mathbin{;} \varphi \mathbin{;} p = \psi = Fv \mathbin{;} \varphi \mathbin{;} p$. □

So, in $\mathcal{S}et$ the notions of alg- and base-congruence coincide, and in arbitrary categories an initial base-congruence p has also the initiality property with respect to the alg-congruences, though p itself is not necessarily an alg-congruence. Thus it is to be expected that a categorical construction of the initial base-congruence requires stronger conditions of the underlying category and F than the construction of Lehmann [39]. I have not been able to check this in detail. Lehmann's construction does require that functor F preserves epis, and that free F-algebras exist. (In the notation of Chapter 3 the free F-algebra is something like $\mu(\underline{a}+F)$.) Our construction assumes that $\mathcal{C}$ has arbitrary finite coequalisers, pushouts and kernel pairs, and that F and the kernel pair functor K of paragraph 22 are ω-cocontinuous.

* * *

Henceforth we say just *congruence* rather than *base-congruence*. Before we can present a construction of the induced congruence, we introduce some more notation and formalise categorically the union of equivalences.

35 More notation. In order to compactify the formulas considerably, we introduce the following abbreviations for parallel pairs (relations, hence the letter ρ). For $\rho = (f, g)$, $\sigma = (h, j)$, and single morphisms x, y we define

$$\begin{array}{lcl} x \,;;\, \rho & = & (x \,;\, f,\ x \,;\, g) \\ \rho \,;;\, y & = & (f \,;\, y,\ g \,;\, y) \\ \rho \,;;\, \sigma & = & (f \,;\, h,\ g \,;\, j) \\ \rho\ \underline{\text{equal}} & \equiv & f = g\,. \end{array}$$

We give $;$ priority over $;;$, so that $;;$ binds even weaker than $;$ and $(x \,;\, x') \,;;\, \rho \,;;\, \sigma \,;;\, (y \,;\, y')$ can be written without parentheses, thus $x \,;\, x' \,;;\, \rho \,;;\, \sigma \,;;\, y \,;\, y'$. It is quite important to be aware that the source category of K is $\bigvee(a)$ and not $\mathcal{C}$. For suppose that p is an object in $\bigvee(a)$ and x, y are morphisms in $\bigvee(a)$ so that all three are morphisms in $\mathcal{C}$. Then, of course, $K(x \,;\, y) = Kx \,;\, Ky$, but $K(p \,;\, x)$ is not equal to $Kp \,;\, Kx$ since morphism Kx in $\bigwedge(a, a)$ is a single morphism in $\mathcal{C}$ and object Kp in $\bigwedge(a, a)$ is a parallel pair in $\mathcal{C}$. Even with the $;$ replaced by $;;$ a composition of Kp with Kx (in either order) doesn't make sense in general.

With this notation the definition of congruence admits an alternative formulation. For a parallel pair $\rho = (f, g)$,

$$\begin{array}{lcl} p \textit{ includes } (f, g) & \equiv & \rho \,;;\, p\ \underline{\text{equal}} \\ p \text{ is congruence for } \varphi & \equiv & KFp \,;;\, \varphi \,;\, p\ \ \underline{\text{equal}}\,. \end{array} \qquad \mathbf{36}$$

The former claim is obvious. For the latter we argue

$$\begin{array}{ll} & p \text{ is congruence for } \varphi \\ \equiv & \quad \text{original definition 32} \\ & \forall \rho :: \ \ \rho \,;;\, Fp\ \underline{\text{equal}} \ \Rightarrow\ \rho \,;;\, \varphi \,;\, p\ \underline{\text{equal}} \\ \equiv & \quad \text{definition } K \text{ (paragraph 21 and 22), and } \bigwedge(\text{-},\text{-}) \end{array}$$

$\forall\rho :: \ (\exists x :: \ \rho = x \mathbin{;;} KFp) \ \Rightarrow \ \rho \mathbin{;;} \varphi \mathbin{;} p \ \underline{\text{equal}}$

$\equiv$ proposition logic

$\forall\rho :: \ \forall x :: \ x \mathbin{;;} KFp \mathbin{;;} \varphi \mathbin{;} p \ \underline{\text{equal}}$

$\equiv$ proposition logic, Leibniz (for $\Leftarrow$) and instantiation $[x := id]$ (for $\Rightarrow$)

$KFp \mathbin{;;} \varphi \mathbin{;} p \ \underline{\text{equal}}$.

37 Uniting equivalences — pushout. Recall the concepts 'proper' and 'equivalence' discussed in Section 2d: a function p on set a is (or represents) an equivalence relation on a, namely the one containing all $(x,y) \in a \times a$ for which $p(x) = p(y)$. An equivalence p on a is called proper if function p is surjective. We shall now give a categorical description of the (proper) union of two proper equivalences; this turns out to be a pushout construct. So, let $\mathcal{C}$ be $\mathcal{S}et$, and let a be a set and p, q be proper equivalences on a, fixed for the following discussion. Here is the typing of p and q, and the variables used in the sequel.

$$\begin{tikzcd} a \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, "r"] & {} \\ {} & {} \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, "r"] & {} \arrow[d, "r''"] \\ {} \arrow[r, "r'"'] & {} \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, "r"] & {} \arrow[d, "r''"] \arrow[ddr, "s''"] & \\ {} \arrow[r, "r'"'] \arrow[drr, "s'"'] & {} \arrow[dr] & \\ & & {} \end{tikzcd}$$

Each pair (r', r'') with $p \mathbin{;} r' = q \mathbin{;} r''$ determines an equivalence r on a that includes both p and q, namely

$$r \ = \ p \mathbin{;} r' \ = \ q \mathbin{;} r''.$$

(Indeed, the r so defined has source a, and if two elements of a have an equal image under p, or q, then they have an equal image under r as well.) Conversely, if an equivalence r on a includes both p and q, then $p \mathbin{;} r' = r = q \mathbin{;} r''$ for a pair (r', r'') uniquely determined by r. (Indeed, let u satisfy $u \mathbin{;} p = id$ (it exists since p is surjective), then we can construct an expression for r' as follows.

$p \mathbin{;} r' = r$

$\equiv$ $p = p \mathbin{;} u \mathbin{;} p$ and r includes p, hence $r = p \mathbin{;} u \mathbin{;} r$

$p \mathbin{;} r' = p \mathbin{;} u \mathbin{;} r$

$\equiv$ surjectivity of p

$r' = u \mathbin{;} r$.

Similarly for r''.) So, using the definition of $\bigvee(p \ulcorner q)$ in paragraph 7,

r is an equivalence on a including both p and q

$\equiv$

$$p ; r' = r = q ; r'' \text{ for some –uniquely determined– object } (r', r'') \text{ in } \bigvee(p \ulcorner q).$$

Now, for objects (r', r'') and (s', s'') in $\bigvee(p \ulcorner q)$,

$$
\begin{array}{ll}
& \text{“equivalence” } (r', r'') \text{ is included in “equivalence” } (s', s'') \\
\equiv & \quad \text{above representation of equivalences} \\
& \text{equivalence } p ; r' \ (= q ; r'') \text{ is included in equivalence } p ; s' \ (= q ; s'') \\
\equiv & \quad \text{representation equivalences, set theory} \\
& p ; r' ; x = p ; s' \ \wedge \ q ; r'' ; x = q ; s'' \qquad \text{for some } x \\
\equiv & \quad \text{properness (surjectivity) of } p \text{ and } q \\
& r' ; x = s' \ \wedge \ r'' ; x = s'' \qquad \text{for some } x \\
\equiv & \quad \text{definition } \bigvee(p \ulcorner q) \\
& x \colon (r', r'') \to (s', s'') \text{ in } \bigvee(p \ulcorner q) \qquad \text{for some } x.
\end{array}
$$

Finally, $p \sqcup q$ is: the least, proper, equivalence that includes both p and q; least meaning being included in each equivalence that also includes both p and q. An explicit expression in for $p \sqcup q$ is readily constructed. Define normal equivalence relations $P, Q, R \subseteq a \times a$ by

$$
\begin{array}{lcl}
P & = & \{(x, y) | \ p(x) = p(y)\} \\
Q & = & \{(x, y) | \ q(x) = q(y)\} \\
R & = & \bigcup n :: \ (P \cup Q)^n .
\end{array}
$$

Then

$$
\begin{array}{lcl}
p \sqcup q & : & a \to a/R \\
p \sqcup q & = & x \mapsto \text{ the } R\text{-equivalence class of } x .
\end{array}
$$

Alternatively, $p \sqcup q$ can be expressed by initiality as follows. Let $p \sqcup q$ be represented by (r', r'') in $\bigvee(p \ulcorner p)$, and take u such that $u ; p \sqcup q = id$, which is possible since $p \sqcup q$ is surjective. Let equivalence s, including both p and q and represented by (s', s''), be arbitrary. Then the initiality statement

$$x \colon (r', r'') \to (s', s'') \text{ in } \bigvee(p \ulcorner q) \qquad \equiv \qquad x = \text{some expr not involving } x$$

is established as follows.

$$
\begin{array}{ll}
& x \colon (r', r'') \to (s', s'') \text{ in } \bigvee(p \ulcorner q) \\
\equiv & \quad \text{definition } \bigvee(p \ulcorner q) \\
& r' ; x = s' \ \wedge \ r'' ; x = s'' \\
\equiv & \quad \text{properness (surjectivity) } p \text{ and } q \\
& p ; r' ; x = p ; s' \ \wedge \ q ; r'' ; x = q ; s'' \\
\equiv & \quad \text{representation } p \sqcup q = r \text{ by } (r', r''), \text{ and } s \text{ by } (s', s'') \\
& p \sqcup q ; x = s \ \wedge \ p \sqcup q ; x = s \\
\equiv & \quad \text{proposition logic}
\end{array}
$$

$$\begin{array}{lll}
 & p \sqcup q \,;\, x = s & \\
(*) & \equiv & \text{below: } s = p \sqcup q \,;\, u \,;\, s \\
 & p \sqcup q \,;\, x = p \sqcup q \,;\, u \,;\, s & \\
 & \equiv & \text{surjectivity } p \sqcup q \\
 & x = u \,;\, s. &
\end{array}$$

For step $(*)$ we argue by extensionality. For each $x\colon 1 \to a$ (an element in a considered as a nullary function)

$$\begin{array}{ll}
 & x \,;\, s = x \,;\, p \sqcup q \,;\, u \,;\, s \\
\Leftarrow & \quad s \text{ includes both } p \text{ and } q \\
 & x \vartriangle (x \,;\, p \sqcup q \,;\, u) \;\in\; (P \cup Q)^n \quad \text{for some } n \\
\equiv & \quad \text{above observation } p \sqcup q \approx \bigcup n :: \; (P \cup Q)^n \\
 & x \,;\, p \sqcup q = x \,;\, p \sqcup q \,;\, u \,;\, p \sqcup q \\
\equiv & \quad \text{property } u \,;\, p \sqcup q = id \\
 & \textbf{true}.
\end{array}$$

So, indeed, $p \sqcup q$ is initial in $\bigvee(p \ulcorner q)$.

Abstracting from *Set* and the application here, an initial object in $\bigvee(p \ulcorner q)$ is called a pushout of p and q.

38 Pushout. Let $\mathcal{C}$ be arbitrary, the default category. Let p and q be morphisms with common source. The **pushout** of p and q is: an initial object of $\bigvee(p \ulcorner q)$. Inspired by the discussion above we use the notation

$$(p \sqcup\!| \; q,\; p \;|\!\sqcup q) \quad = \quad \text{the initial object in } \bigvee(p \ulcorner q)$$

and put

$$p \sqcup q \quad = \quad p \,;\, \sqcup\!|\, q \quad = \quad q \,;\, p \;|\!\sqcup q\,,$$

thus avoiding duplication of p and q in the composites. (For those who know pushouts, $p \sqcup\!|\; q$ is the pushout of q *along* p, and, in the conventional diagrammatic representation of the pushout square, $p \sqcup\!|\; q$ is parallel to q as suggested by the symbol $\sqcup\!|$. Similarly for $p \;|\!\sqcup q$, and $p \sqcup q$ denotes the diagonal.)

$$\begin{tikzcd}
a \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, "p \sqcup q" description] & {} \arrow[d, "p \;|\!\sqcup q"] \\
{} \arrow[r, "p \sqcup\!| \; q"'] & {}
\end{tikzcd}$$

We write

$$(\!| r', r'' |\!)_{p,q} \quad \text{for} \quad (\!| p \sqcup\!\mid q,\; p \mid\!\sqcup q \;\rightharpoonup\; r', r'' |\!)_{\bigvee(p\ulcorner q)}\,.$$

Then the well-formedness law reads

$$p \mathbin{;} r' = q \mathbin{;} r'' \quad\Rightarrow\quad (\!| r', r'' |\!)_{p,q} \;:\; \operatorname{tgt}(p \sqcup q) \;\to\; \operatorname{tgt} r' (=\; \operatorname{tgt} r'') \qquad \text{pushout-TYPE}$$

and the characterisation for pushouts works out as follows.

$$p \sqcup\!\mid q \mathbin{;} x = r' \;\wedge\; p \mid\!\sqcup q \mathbin{;} x = r'' \quad\equiv\quad x = (\!| r', r'' |\!)_{p,q} \qquad \text{pushout-CHARN}$$

for each (r', r'') in $\bigvee(p\ulcorner q)$.

As an illustration of some of the notation, here is a well-known fact that we'll use later.

39 Fact. *$p \mid\!\sqcup q$ is epic whenever f is epic.*

Proof. Writing ρ for (x, y) we argue

$$\begin{array}{ll}
 & \rho\ \underline{\text{equal}} \\
\equiv & \qquad \text{each colimit, hence pushout, is jointly epic} \\
 & p \mid\!\sqcup q \mathbin{;;} \rho\ \underline{\text{equal}}\ \text{ and }\ p \sqcup\!\mid q \mathbin{;;} \rho\ \underline{\text{equal}} \\
\equiv & \qquad \text{premise: } p \text{ epic} \\
 & p \mid\!\sqcup q \mathbin{;;} \rho\ \underline{\text{equal}}\ \text{ and }\ p \mathbin{;} p \sqcup\!\mid q \mathbin{;;} \rho\ \underline{\text{equal}} \\
\equiv & \qquad \text{remember } p \mathbin{;} p \sqcup\!\mid q = q \mathbin{;} p \mid\!\sqcup q \\
 & p \mid\!\sqcup q \mathbin{;;} \rho\ \underline{\text{equal}}\ \text{ and }\ q \mathbin{;} p \mid\!\sqcup q \mathbin{;;} \rho\ \underline{\text{equal}} \\
\equiv & \qquad \text{Leibniz, proposition logic} \\
 & p \mid\!\sqcup q \mathbin{;;} \rho\ \underline{\text{equal}}.
\end{array}$$

□

40 Global constants. Category ω with endofunctor S has been defined in paragraph 28. Let the default category $\mathcal{C}$ be an ω-cocomplete category that has all finite coequalisers, kernel pairs, and pushouts, and for which the kernel pair functor K is ω-cocontinuous. Let F be an ω-cocontinuous endofunctor on $\mathcal{C}$, φ: $Fa \to a$ be an algebra, and ρ a parallel pair with target a. These entities, as well as D, γ, and p defined below, are fixed throughout the sequel.

41 The construction. Define an ω-chain D in $\bigvee(a)$ as follows. First we define the objects Dn in $\bigvee(a)$. (Interpreted in $\mathcal{S}et$ the objects Dn form an ascending chain of proper equivalences, each DSn being the union of proper equivalence Dn with the equivalence induced by $KFDn \mathbin{;;} \varphi$, so as to become more like a congruence, see equation 36.)

$$D0 \quad = \quad C\rho \qquad\qquad \text{an object in } \bigvee(a)$$

$$DSn \quad = \quad Dn \sqcup C(KFDn \mathbin{;;} \varphi) \quad \text{an object in } \bigvee(a)$$

for all n. The wish that D is a functor of type $\omega \to \bigvee(a)$ almost forces the definition of the morphisms $D(m{\le}n)$. We shall nowhere use these clauses explicitly.

$$\begin{array}{lcllr} D(n{\le}n) & = & id & : & Dn \to Dn \text{ in } \bigvee(a) \\ D(m{\le}Sn) & = & D(m{\le}n) \mathbin{;} Dn \sqcup\!| \; C(KFDn \mathbin{;;} \varphi) & : & Dm \to DSn \text{ in } \bigvee(a) \end{array}$$

for all $m \le n$. It is routine to verify, by induction on n, that D satisfies the typing as indicated, and hence D: $\omega \to \bigvee(a)$ indeed.

Define γ to be the colimit for D in $\bigvee(a)$, and define p to be its target:

$$\gamma\text{: } D \dot\to \underline{p} \quad \text{is colimit for } D, \text{ in } \bigvee(a).$$

This definition of γ and p presupposes that $\bigvee(a)$ is ω-cocomplete, which in turn follows from ω-cocompleteness of $\mathcal{C}$; see Mac Lane [41, exercise 1 on page 108]. (Interpreted in $\mathcal{S}et$ equivalence p is defined to be the union of all the equivalences Dn.) By the naturality of γ it follows that

$$\gamma n\text{: } Dn \to p \text{ in } \bigvee(a)$$

that is,

42 $\qquad Dn \mathbin{;} \gamma n = p$

for all objects n in ω.

43 Theorem (Correctness) *The p so defined is the congruence for φ induced by ρ. Moreover, p is epic in $\mathcal{C}$.*

We shall prove the theorem in the three lemmas 44, 46, 47 that follow:

44: Morphism p is epic in $\mathcal{C}$.
46: Morphism p is a congruence for φ including ρ.
47: Let q be a congruence for φ including ρ.
Then there exists a morphism $p\backslash q$ satisfying congruence-CHARN 33.

The hint 'coequaliser' means that '$\sigma \mathbin{;;} C\sigma$ $\underline{\text{equal}}$' holds for each σ.

44 Lemma. *Morphism p is epic in $\mathcal{C}$.*

Proof.

$x = y$

$\equiv$ each colimit is jointly epic;
For all n:

$\gamma n \mathbin{;} x = \gamma n \mathbin{;} y$

$\equiv$ each Dn is epic (shown below in Lemma 45)

$Dn \mathbin{;} \gamma n \mathbin{;} x = Dn \mathbin{;} \gamma n \mathbin{;} y$

$\equiv$ observe γ: $D \dot\to \underline{p}$, hence $Dn \mathbin{;} \gamma n = p$

$p \mathbin{;} x = p \mathbin{;} y$.

□

45 Lemma. *Each Dn is epic.*

Proof. By induction on n. The Basis is immediate since $D0 = C\rho$ and each coequaliser is epic. For the Step we argue

DSn epic
$\equiv$ definition DSn and $\sqcup$
$Dn \;;\; Dn \sqcup\!| \; C(KFDn \;;;\; \varphi)$ epic
$\Leftarrow$ composition of epis is epic
Dn and $Dn \sqcup\!| \; C(KFDn \;;;\; \varphi)$ both epic
$\Leftarrow$ pushout of epi is epic: Fact 39
Dn and $C(KFDn \;;;\; \varphi)$ both epic
$\equiv$ induction hypothesis, each coequaliser is epic
true.

□

46 Lemma. *Morphism p is a congruence for φ including ρ.*

Proof. Morphism p includes ρ since

$\rho \;;;\; p$ $\underline{\text{equal}}$
$\equiv$ observed in 42: $D0 \;;\; \gamma 0 = p$
$\rho \;;;\; D0 \;;\; \gamma 0$ $\underline{\text{equal}}$
$\Leftarrow$ Leibniz
$\rho \;;;\; D0$ $\underline{\text{equal}}$
$\equiv$ definition $D0 = C\rho$, coequaliser
true.

Morphism p is a congruence for φ since

$KFp \;;;\; \varphi \;;\; p$ $\underline{\text{equal}}$
$\Leftarrow$ F and K are ω-cocontinuous so preserve colimits,
hence $KF\gamma$ is a colimit and, hence, jointly epic;
For all n:
$KF\gamma n \;;;\; KFp \;;;\; \varphi \;;\; p$ $\underline{\text{equal}}$
$\equiv$ observed in 42: $p = DSn \;;\; \gamma Sn$; similarly,
$KF\gamma$: $KFD \dot{\to} KF\underline{p}$ in $\bigwedge(Fa, Fa)$, so $KF\gamma n \;;;\; KFp = KFDn$
$KFDn \;;;\; \varphi \;;\; DSn \;;\; \gamma Sn$ $\underline{\text{equal}}$
$\equiv$ definition $DSn = Dn \sqcup C(KFDn \;;;\; \varphi)$, definition $\sqcup$

$KFDn \mathbin{;;} \varphi \mathbin{;} C(KFDn \mathbin{;;} \varphi) \mathbin{;} Dn \mathbin{\iota\sqcup} C(KFDn \mathbin{;;} \varphi) \mathbin{;} \gamma Sn$ $\underline{\text{equal}}$

$\Leftarrow$ Leibniz

$KFDn \mathbin{;;} \varphi \mathbin{;} C(KFDn \mathbin{;;} \varphi)$ $\underline{\text{equal}}$

$\equiv$ coequaliser

true.

□

47 Lemma. *Let* q *be a congruence for* φ *including* ρ. *Then there exists a morphism* $p\backslash q$ *satisfying* congruence-CHARN 33.

Proof. Throughout the proof the notation $\rightarrow_a$ abbreviates $\rightarrow_{\bigvee(a)}$.
We guess that the desired $p\backslash q$ has the form $\gamma\backslash\delta$ for some cocone $\delta \colon D \dashrightarrow \underline{q}$. (This is a very weak guess since many categorical constructions have this form.) (Both $p\backslash q$ and $\gamma\backslash\delta$ are morphisms in $\bigvee(a)$ and hence in $\mathcal{C}$ as well.) The existence of a $\delta \colon D \dashrightarrow \underline{q}$ for each q is sufficient to establish congruence-CHARN.

$x \colon p \rightarrow_a q$

$\equiv$ definition $\rightarrow_a$

$p \mathbin{;} x = q$

$\equiv$ observed in 42: $p = Dn \mathbin{;} \gamma n$, similarly $q = Dn \mathbin{;} \delta n$

$Dn \mathbin{;} \gamma n \mathbin{;} x = Dn \mathbin{;} \delta n$ for all n

$\equiv$ Lemma 45: each Dn is epic

$\gamma n \mathbin{;} x = \delta n$ for all n

$\equiv$ colimit-CHARN

$x = \gamma\backslash\delta.$

It remains to construct some cocone $\delta \colon D \dashrightarrow \underline{q}$ in $\bigvee(a)$ for arbitrary q as in the statement of the lemma. We shall derive $\delta n \colon Dn \rightarrow_a q$ by induction on n, and show the naturality afterwards.
For the Basis we argue

$x \colon D0 \rightarrow_a q$

$(*)$ $\equiv$ definition $D0 = C\rho$; coequaliser-CHARN (Section 2d)

$x = D0\backslash q$

where the use of coequaliser-CHARN in step $(*)$ requires as well-formedness condition that q includes ρ; this is given by the premise. So we define $\delta 0 = D0\backslash q$, and then have

(a) $x \colon D0 \rightarrow_a q \quad\equiv\quad x = \delta 0\,.$

For the induction Step the induction hypothesis says that $\delta n \colon Dn \rightarrow_a q$ exists. Aiming at a definition for $\delta Sn \colon DSn \rightarrow_a q$ we argue

$x \colon DSn \rightarrow_a q$

$\equiv$ definition $\rightarrow_a$ and $DSn = Dn \sqcup C(KFDn \mathbin{;;} \varphi)$

$Dn \sqcup C(KFDn \mathbin{;;} \varphi) \mathbin{;} x = q$

$\equiv$ proposition logic

$Dn \sqcup C(KFDn \mathbin{;;} \varphi) \mathbin{;} x = q \quad \wedge \quad Dn \sqcup C(KFDn \mathbin{;;} \varphi) \mathbin{;} x = q$

$\equiv$ for the left conjunct:
property $f \sqcup g = f \mathbin{;} f \sqcup\!| \; g$ and
induction hypothesis $q = Dn \mathbin{;} \delta n$;
for the right conjunct:
property $f \sqcup g = g \mathbin{;} f \; |\!\sqcup g$ and

(*) explained below $q = C(KFDn \mathbin{;;} \varphi) \mathbin{;} C(KFDn \mathbin{;;} \varphi)\backslash q$

$Dn \mathbin{;} Dn \sqcup\!| \; C(KFDn \mathbin{;;} \varphi) \mathbin{;} x = Dn \mathbin{;} \delta n \quad \wedge$

$C(KFDn \mathbin{;;} \varphi) \mathbin{;} Dn \; |\!\sqcup C(KFDn \mathbin{;;} \varphi) \mathbin{;} x = C(KFDn \mathbin{;;} \varphi) \mathbin{;} C(KFDn \mathbin{;;} \varphi)\backslash q$

$\equiv$ each coequaliser is epic, and so is Dn (Lemma 45)

$Dn \sqcup\!| \; C(KFDn \mathbin{;;} \varphi) \mathbin{;} x = \delta n \quad \wedge$

$Dn \; |\!\sqcup C(KFDn \mathbin{;;} \varphi) \mathbin{;} x = C(KFDn \mathbin{;;} \varphi)\backslash q$

$\equiv$ pushout-CHARN

$x = (\!|\delta n, \; C(KFDn \mathbin{;;} \varphi)\backslash q|\!)_{Dn, C(KFDn;;\varphi)}$

In hint (*) it is assumed that $C(KFDn \mathbin{;;} \varphi)\backslash q$ is well-formed (exists). The condition for this is that q includes $KFDn \mathbin{;;} \varphi$, which is shown as follows.

$KFDn \mathbin{;;} \varphi \mathbin{;} q \; \underline{\text{equal}}$

$\equiv$ induction hypothesis $\delta n \colon Dn \to_a q$,
so $KF\delta n \colon KFDn \to_{\bigwedge(Fa,Fa)} KFq$, that is, $KFDn = KF\delta n \mathbin{;;} KFq$

$KF\delta n \mathbin{;;} KFq \mathbin{;;} \varphi \mathbin{;} q \; \underline{\text{equal}}$

$\Leftarrow$ Leibniz, premise: q is congruence for φ

true.

So we define $\delta Sn = (\!|\delta n, \; C(KFDn \mathbin{;;} \varphi)\backslash q|\!)_{Dn, C(KFDn;;\varphi)}$, and then have

(b) $\quad x \colon DSn \to_a q \quad \equiv \quad x = \delta Sn$.

Finally, to show naturality (commutativity of all triangles) we argue

$\delta \colon D \dashrightarrow \underline{q}$

$\equiv$ definition $\dashrightarrow$, and $\underline{q}(n{\le}Sn) = id_q = id$
For all n:

$D(n{\le}Sn) \mathbin{;} \delta Sn = \delta n$

$\equiv$ above (a), (b): $x \colon Dn \to_a q \quad \equiv \quad x = \delta n$

$D(n{\le}Sn) \mathbin{;} \delta Sn \colon Dn \to_a q$

$\Leftarrow$ above (b): $\delta Sn \colon DSn \to_a q$; composition

$D(n{\le}Sn)\colon\ Dn \to_a DSn$

$\equiv$ functor; definition category ω

true.

□

2g Conclusion

This chapter contains nontrivial examples of algebraic calculation in the framework of category theory. The calculations are quite smooth; there were few occasions where we had to interrupt the calculation, for establishing an auxiliary result or for introducing a new (name for a) morphism. Thanks to the systematisation of the notation and laws for the unique arrows brought forward by initiality, there is less or no need to draw or remember commutative diagrams for the inspiration or verification of a step in a calculation. Each step is easily verified, and there is ample opportunity for machine assistance in this respect. More importantly, the *construction* of required morphisms from others is performed as a calculation as well. There are several places where a morphism is constructed by beginning to prove the required property while, along the way, determining more and more of (an expression for) the morphism. Thus proof and construction go hand-in-hand, in an algebraic style.

There is one purpose for which pictures are certainly helpful: namely to present the typing of various morphisms, in particular to see what morphisms have a common source or common target. For example, in the course of constructing the proof in the last section (and correcting failing attempts) I have used a picture of the pushout of Dn and $C(KFDn \,;\, \varphi)$ several times in order to convince myself that the formulas I wrote down made sense — which was not always the case.

All calculations can be interpreted in $\mathcal{S}et$ so that, actually, we have quite involved calculations with algorithms (functions). Calculations with algorithms working on more usual datatypes will be explored further in the next chapter.

Chapter 3

Algebras categorically

Roughly speaking, an algebra is a collection of operations, and a homomorphism between two algebras is a function that commutes with the operations. Homomorphisms are computationally relevant and calculationally attractive; they occur frequently in transformational programming. Algebras are also used to define the notion of datatypes.

The language of category theory provides for a simple and elegant formalisation and investigation of homomorphisms and algebras; it also suggests a dualisation and several generalisations.

3a Algebra, homomorphism

1 Distributivity. In transformational programming, distributivity and commutativity properties of functions play an important rôle. We say that f *distributes over* binary operation $\oplus$ if

$$f(x \oplus y) \quad = \quad fx \oplus fy$$

for all x, y. Expressed at the function level this reads:

$$\oplus \mathbin{;} f \quad = \quad I\!I f \mathbin{;} \oplus ,$$

and this is a slight generalisation of the property that f *commutes with* $\oplus$. A further generalisation of the equation reads:

$$\oplus \mathbin{;} f \quad = \quad I\!I f \mathbin{;} \otimes .$$

The equation asserts the semantic equality of two different ways of computing the same value. In case the equation holds, the efficiency of a program may be improved by replacing a part $\oplus \mathbin{;} f$ in a program by $I\!I f \mathbin{;} \otimes$ (or just the other way around, that depends on the the operations and function at hand). Thus function f is "promoted" (in the sense of Bird's [8] 'Promotion and Accumulation strategies', and Darlington's [15] 'filter promotion') from being a post-process of $\oplus$ into being a pre-process for $\otimes$. (In view of this some authors say "f is $\oplus \to \otimes$ promotable".) Notice also that such a program transformation need not be done with an immediate efficiency improvement in mind, but may be done to enable future transformations that do improve the efficiency in the end. Therefore such generalised distributivity properties are relevant for transformational programming.

2 Generalisation. The typing of the above operations and function is

$$\oplus\colon I\!I a \to a \qquad f\colon a \to b \qquad \otimes\colon I\!I b \to b$$

for some a, b. For a useful formal treatment we generalise the source structure of the operations from $I\!I$ to an arbitrary functor F. So, writing φ, ψ for $\oplus, \otimes$, the typing reads

$$\varphi\colon Fa \to a \qquad f\colon a \to b \qquad \psi\colon Fb \to b$$

for some a, b, and the more generalised distributivity property reads

$$\varphi \mathbin{;} f \quad = \quad Ff \mathbin{;} \psi .$$

This generalisation also captures the distribution over several operations simultaneously, as shown by the following calculation.

$$\begin{array}{ll}
 & \varphi_0 \mathbin{;} f \;=\; F_0 f \mathbin{;} \psi_0 \;\wedge\; \varphi_1 \mathbin{;} f = F_1 f \mathbin{;} \psi_1 \\
\equiv & \quad \text{sum} \\
 & (\varphi_0 \mathbin{;} f) \triangledown (\varphi_1 \mathbin{;} f) \;=\; (F_0 f \mathbin{;} \psi_0) \triangledown (F_1 f \mathbin{;} \psi_1) \\
\equiv & \quad \text{sum} \\
 & \varphi_0 \triangledown \varphi_1 \mathbin{;} f \;=\; F_0 f + F_1 f \mathbin{;} \psi_0 \triangledown \psi_1 \\
\equiv & \quad \text{functor} \\
 & \varphi_0 \triangledown \varphi_1 \mathbin{;} f \;=\; (F_0 + F_1) f \mathbin{;} \psi_0 \triangledown \psi_1 .
\end{array}$$

Notice also that the composite $\varphi_0 \triangledown \varphi_1$ has a type of the form $Fa \to a$ for some F:

$$\varphi_0\colon F_0 a \to a \;\wedge\; \varphi_1\colon F_1 a \to a \qquad \Rightarrow \qquad \varphi_0 \triangledown \varphi_1\colon (F_0 + F_1)a \to a ,$$

so that $\varphi_0 \triangledown \varphi_1\colon Fa \to a$ by taking $F = F_0 + F_1$. Similarly, $\psi_0 \triangledown \psi_1\colon Fb \to b$ if each ψ_i has type $F_i b \to b$.

In the computing science literature a collection

$$\langle a;\ \varphi_0\colon F_0 a \to a,\ \varphi_1\colon F_1 a \to a,\ \ldots\rangle$$

is called an *algebra*. Combining the individual operations φ_i into a single operation $\varphi = \varphi_0 \triangledown \varphi_1 \triangledown \ldots\colon (F_0 + F_1 + \cdots)a \to a$, the collection is fully determined by φ alone:

$$a \;=\; \text{tgt}\,\varphi \qquad \text{and} \qquad \varphi_i \;=\; in_i \mathbin{;} \varphi \qquad \text{for all } i .$$

Conversely, each $\varphi\colon Fa \to a$ determines such a collection:

$$\langle \mathrm{tgt}\,\varphi;\ \ \varphi\colon Fa \to a\rangle\,.$$

Thus each $\varphi\colon Fa \to a$ is called an algebra. Accordingly, a function f satisfying $\varphi \mathbin{;} f = Ff \mathbin{;} \psi$ is called an *F-homomorphism* from φ to ψ; we write $f\colon \varphi \to_F \psi$. (Thus promotability of f is nothing but the property that f is a homomorphism.) More precise definitions are given in the sequel.

The generalisation from $I\!I$ to an arbitrary functor F is not yet the full story. Consider an operation like

$$\mathrm{div} \vartriangle \mathrm{mod} \quad : \quad I\!I nat \to I\!I nat\,,$$

accepting and producing pairs of values. Abstracting from the particulars, this is an operation $\varphi\colon I\!I a \to I\!I a$. Let $\psi\colon I\!I b \to I\!I b$ be another binary operation that yields binary results, and let $f\colon a \to b$ be a function. Then a generalised distributivity property for these operations reads

$$\varphi \mathbin{;} I\!I f \quad = \quad I\!I f \mathbin{;} \psi\,.$$

Such a property is again quite relevant for transformational programming. The two occurrences of $I\!I$ generalise to two functors F, G so that

$$\varphi\colon Fa \to Ga \qquad f\colon a \to b \qquad \psi\colon Fb \to Gb\,,$$

and the distributivity then reads

$$\varphi \mathbin{;} Gf \quad = \quad Ff \mathbin{;} \psi\,.$$

Such φ and ψ are called *F, G-dialgebras* (pronounced di-algebras), and such an f is a homomorphism for F, G-dialgebras. Taking $G = I$ we get the case of F-algebras as a particular instance. Taking $F = I$ gives the same result as what is got by dualising the notion of algebra, hence known as *co-algebra*. (We keep saying 'homomorphism' in all these cases, rather than 'di-homomorphism' etc.) We shall see in Section 3d that a collection of algebras and co-algebras together is a single dialgebra, and that the notion of dialgebra also covers many-sortedness.

3 Datatypes. A further motivation to study (di)algebras is their use in formalising the notion of datatype. Briefly, a datatype is a collection of operations some of which are "constructors": each element of the datatype can be constructed by the constructors in a finite way, and via these constructors functions on the datatype may be defined. So, part of a datatype is a particular algebra; the distinguishing property is categorically known as initiality of the algebra. Dualisation leads to the notion of final co-algebra; less known, but quite useful as we shall see. There are reasonable conditions on F in order that an initial F-algebra, or final F-co-algebra, exists. (I do not know of similar conditions for dialgebras in general. Moreover, I do not know of 'normal' datatypes that can only be modeled by initial or final nontrivial dialgebras. Hagino [29] shows that function spaces, *exponentials* in category speak, are dialgebras.)

The following definition captures the preceding observations. Anticipating laws homo-ID and homo-COMPOSE in paragraph 13, we also define the category of (di-,co-)algebras (but see the remarks that follow the definition). We postpone the discussion and formalisation of laws (conditional equations) satisfied by operations (algebras) to Chapter 5.

4 Definition. Let $\mathcal{A}, \mathcal{C}$ be categories, $\mathcal{C}$ the default one, and $F, G\colon \mathcal{A} \to \mathcal{C}$ be functors. An F, G**-dialgebra** is: a morphism φ typed

$$\varphi\colon Fa \to Ga \qquad \text{DIALGEBRA}$$

for some a called the **carrier**. Let φ, ψ be F, G-dialgebras. An F, G**-homomorphism** from φ to ψ is: a morphism f for which

$$\varphi \,;\, Gf = Ff \,;\, \psi, \qquad \text{denoted} \quad f\colon \varphi \to_{F,G} \psi \qquad \text{HOMO}$$

It follows that $f\colon \text{carrier}\,\varphi \to \text{carrier}\,\psi$. We say just 'homomorphism' when F and G are clear from the context.

Category $\mathcal{D}iAlg(F,G)$ is: the category built upon $\mathcal{C}$ that has the F,G-dialgebras as objects, and the F,G-homomorphisms as morphisms in such a way that $f\colon \varphi \to_{F,G} \psi$ abbreviates $f\colon \varphi \to_{\mathcal{D}iAlg(F,G)} \psi$. Functor $U\colon \mathcal{D}iAlg(F,G) \to \mathcal{C}$ is defined by

$$\begin{array}{lcll} U\varphi & = & \text{the carrier of } \varphi \ \ (\text{an object in } \mathcal{C}), & \text{for dialgebra } \varphi \\ Uf & = & f \qquad\qquad : \ U\varphi \to U\psi, & \text{for } f\colon \varphi \to_{F,G} \psi\,. \end{array}$$

Notice that U depends on F, G; a more precise notation would be $U_{F,G}$.

An F**-algebra** is: an F, I-dialgebra ($\varphi\colon Fa \to a$), and $\mathcal{A}lg(F) = \mathcal{D}iAlg(F, I)$,
an F**-co-algebra** is: an I, F-dialgebra ($\varphi\colon a \to Fa$), and $\mathcal{C}o\mathcal{A}lg(F) = \mathcal{D}iAlg(I, F)$;
here it follows that $\mathcal{A} = \mathcal{C}$ and F is an endofunctor.
Finally, $\to_F$, $\succ\!\!-_F$ abbreviate $\to_{F,I}$, $\to_{I,F}$ respectively.

5 Remarks on the definition. The two formulas for HOMO are easy to remember, in spite of the swap of F, G when comparing the two formulas. The order of F, G in the notation $f\colon \varphi \to_{F,G} \psi$ is the same as the order of F, G in the typing of the dialgebras $\varphi\colon Fa \to Ga$ and $\psi\colon Fb \to Gb$. As regards the equation, since F describes the source structure of the algebras, morphism Ff can only sensibly occur at the source side of an dialgebra; similarly, Gf can only sensibly occur at an target side. Moreover, since f is *from* φ *to* ψ, the occurrences of f are at the target side of φ and at the source side of ψ. The equations denoted by $\to_F$ and $\succ\!\!-_F$ differ only in the place of F; the position is indicated by the position of > on the "arrow" symbol.

Strictly speaking the definition of the categories is wrong in the sense that the morphisms in $\mathcal{D}iAlg(F,G)$ —as defined above— do not have a unique source and target. It may happen that both the equation denoted by $f\colon \varphi \to_{F,G} \psi$ and the equation denoted by $f\colon \chi \to_{F,G} \omega$ are valid, while (φ, ψ) differs from (χ, ω). To repair this defect, the morphisms in $\mathcal{D}iAlg(F,G)$ should be triples (φ, f, ψ). Since category $\mathcal{C}$ is intended as the universe of discourse, the *equation* $f\colon\ \varphi \rightarrow_{F,G} \psi$ is often the statement of interest, and **not** the statement that (φ, f, ψ) is a morphism in $\mathcal{D}iAlg(F,G)$.

Functor U is usually called an Underlying or forgetful functor. Underlying, because its target is the underlying category; forgetful, because it maps special morphisms of $\mathcal{C}$, namely F,G-homomorphisms, into the collection of all morphisms of $\mathcal{C}$, thus "forgetting" the homomorphism property. We shall use U mainly as an abbreviation for 'the carrier of'.

It may happen that $\varphi\colon\ Fa \rightarrow Ga$ as well as $\varphi\colon\ Fb \rightarrow Gb$ for $a \neq b$, and in that case $U\varphi$ is not well defined: its result should be both a and b. To repair this defect too, the objects of $\mathcal{D}iAlg(F,G)$ must more precisely be considered to be pairs (φ, a) for which $\varphi\colon\ Fa \rightarrow Ga$ in $\mathcal{C}$. Thus, whenever we introduce an 'F,G-dialgebra φ' and then use $U\varphi$ to denote its carrier, we should more precisely have introduced 'F,G-dialgebra φ with carrier a' so that $U\varphi$ is uniquely defined to be a. (For algebras and co-algebras there is nothing the matter since functor I is injective.)

Whenever the *equation* denoted by $f\colon\ \varphi \rightarrow_{F,G} \psi$ holds, it follows that φ and ψ are F,G-dialgebras. Indeed,

$$
\begin{array}{ll}
 & \varphi \mathbin{;} Gf = Ff \mathbin{;} \psi \\
\Rightarrow & \quad \text{Leibniz} \\
 & \mathrm{src}(\varphi \mathbin{;} Gf) = \mathrm{src}(Ff \mathbin{;} \psi) \ \wedge\ \mathrm{tgt}(\varphi \mathbin{;} Gf) = \mathrm{tgt}(Ff \mathbin{;} \psi) \\
\equiv & \quad \text{assumption that the two compositions are well-formed} \\
 & \mathrm{src}\,\varphi = \mathrm{src}\,Ff \ \wedge\ \mathrm{tgt}\,Gf = \mathrm{tgt}\,\psi \\
 & \mathrm{tgt}\,\varphi = \mathrm{src}\,Gf \ \wedge\ \mathrm{tgt}\,Ff = \mathrm{src}\,\psi \\
\equiv & \quad \text{functor, source-target notation} \\
 & \varphi\colon\ F\mathrm{src}f \rightarrow G\mathrm{src}f \ \wedge\ \psi\colon\ F\mathrm{tgt}f \rightarrow G\mathrm{tgt}f.
\end{array}
$$

Examples (dialgebras)

6 Naturals. Recall the datatype of naturals as explained in paragraph 1.12. The single operation *zero* is a $\underline{1}$-algebra with carrier *nat*. Indeed,

$$zero\colon\ 1 \rightarrow nat \quad = \quad \underline{1}(nat) \rightarrow nat\,.$$

The single operation *succ* is an I-algebra with carrier *nat*. Indeed,

$$succ\colon\ nat \rightarrow nat \quad = \quad I nat \rightarrow nat\,.$$

The combined operation $zero \triangledown succ$ is an $\underline{1} + I$-algebra with carrier *nat*. Indeed,

$$zero \triangledown succ\colon\ 1 + nat \rightarrow nat \quad = \quad (\underline{1} + I)nat \rightarrow nat\,.$$

The operation $zero \triangledown one \triangledown succ \triangledown add \triangledown mult$ is an $\underline{1} + \underline{1} + I + I\!I + I\!I$-algebra. Indeed,

$$
\begin{array}{lcl}
zero \triangledown one \triangledown succ \triangledown add \triangledown mult & : & 1 + 1 + nat + I\!I nat + I\!I nat \rightarrow nat \\
 & = & (\underline{1} + \underline{1} + I + I\!I + I\!I)nat \rightarrow nat\,.
\end{array}
$$

7 Cons lists. Recall the datatype of cons lists over a as explained in paragraph 1.12. The single operation *nil* is an $\underline{1}$-algebra with carrier La. Indeed,

$$nil\colon\ 1 \to La \;=\; \underline{1}(La) \to La\,.$$

The single operation *cons* is an $\underline{a} \times I$-algebra with carrier La. Indeed,

$$cons\colon\ a \times La \to La \;=\; (\underline{a} \times I)La \to La\,.$$

The combined operation $nil \triangledown cons$ is a $\underline{1} + \underline{a} \times I$-algebra with carrier La. Indeed,

$$nil \triangledown cons\colon\ 1 + a \times La \to La \;=\; (\underline{1} + \underline{a} \times I)La \to La\,.$$

The single operation *size* is a $\underline{nat}$-co-algebra with carrier La, as well as a $\underline{La}$-algebra with carrier nat. Indeed,

$$size\colon\ La \to nat \;=\; La \to \underline{nat}(La) \;=\; \underline{La}(nat) \to nat\,.$$

Combined operation $nil \triangledown cons$ is a bijection between the sets $1 + a \times La$ and La, and has therefore an inverse $(nil \triangledown cons)^{\cup}$. Operation $(nil \triangledown cons)^{\cup}$ has type $La \to 1 + a \times La$ and is a $\underline{1} + \underline{a} \times I$-co-algebra; it decomposes a cons list into its constituents, the constituent of the empty list being the sole member of 1.

8 Streams. Similarly as above, various combinations of *hd*, *tl*, and *from* form F-co-algebras or F-algebras, for suitably chosen functors F. Here is just one example. The combined operation $hd \vartriangle tl$ is a $\underline{a} \times I$-co-algebra. Indeed,

$$hd \vartriangle tl\colon\ Sa \to a \times Sa \;=\; Sa \to (\underline{a} \times I)Sa\,.$$

9 Rose trees. A rose tree over a is a multi-forking tree with labels at the tips. Meertens [46] discusses these in detail. Let Ra be the set of rose trees over a. The constructors are $tip\colon a \to Ra$ and $fork\colon LRa \to Ra$, so that *fork* builds one rose tree from a list of rose trees. Then $tip \triangledown fork$ is an $\underline{a} + L$-algebra. Indeed,

$$tip \triangledown fork\colon\ a + LRa \to Ra \;=\; (\underline{a} + L)Ra \to Ra\,.$$

We shall later see that L (La denotes the set of lists over a) can be extended to a functor, so that $\underline{a} + L$ is a functor indeed. □

Examples (homomorphisms)

10 Taking $F, G, \varphi, \psi = I\!I, I, \oplus, \otimes$ the statement $f\colon \varphi \to_{F,G} \psi$ specialises to the equation $\oplus \,;\, f = I\!I f \,;\, \otimes$, which was discussed in paragraph 1.

11 The function $f\colon nat \to nat$ mapping n to 2^n is an $\underline{1} + I$-homomorphism from $zero \triangledown succ$ to $one \triangledown double$ since

$$zero \triangledown succ \,;\, f$$
$$=$$

$$\begin{aligned}
& (zero \mathbin{;} f) \triangledown (succ \mathbin{;} f) \\
= \; & \\
& (id_1 \mathbin{;} one) \triangledown (f \mathbin{;} double) \\
= \; & \\
& id_1 + f \mathbin{;} one \triangledown double \\
= \; & \\
& (\underline{1} + I)f \mathbin{;} one \triangledown double.
\end{aligned}$$

Actually, both $f\colon\ zero \rightarrow_{\underline{1}} one$ and $f\colon\ succ \rightarrow_I double$ are valid, and therefore also the claim above; see law homo-SUM 18.

12 Function *size* is a homomorphism. Specifically, the defining equations of *size* in paragraph 1.12 actually say:

$$\begin{aligned}
& size\colon\ nil \rightarrow_{\underline{1}} zero \\
& size\colon\ cons \rightarrow_{\underline{a}\times I} add
\end{aligned}$$

hence

$$size\colon\ nil \triangledown cons \rightarrow_{\underline{1}+\underline{a}\times I} zero \triangledown add\,.$$

The last line is immediate by writing out the equations in detail, as we did above for f, or by applying homo-SUM 18. □

13 Laws for homomorphisms. We have already argued in paragraph 1 and 2 that homomorphisms are computationally relevant. They are also calculationally attractive since they satisfy a lot of algebraic properties. The first two are very important and frequently used. Each of the laws is (an abstraction and generalisation of) a pattern of reasoning that occurs somewhere in this text.

14	$id\colon\ \varphi \rightarrow_{F,G} \varphi$			homo-ID
15	$f\colon\ \varphi \rightarrow_{F,G} \psi \;\wedge\; g\colon\ \psi \rightarrow_{F,G} \chi$	$\Rightarrow$	$f \mathbin{;} g\colon\ \varphi \rightarrow_{F,G} \chi$	homo-COMPOSE
16	$f\colon\ \varphi \rightarrow_{FH,GH} \psi$	$\equiv$	$Hf\colon\ \varphi \rightarrow_{F,G} \psi$	homo-FTR1
17	$\left.\begin{array}{l} HFf = FHf \;\wedge\; HGf = GHf \\ f\colon\ \varphi \rightarrow_{F,G} \psi \end{array}\right\}$	$\Rightarrow$	$Hf\colon\ H\varphi \rightarrow_{F,G} H\psi$	homo-FTR2
18	$f\colon\ \varphi_i \rightarrow_{F_i,G} \psi_i \;\; (i = 0,1)$	$\equiv$	$f\colon\ \varphi_0 \triangledown \varphi_1 \rightarrow_{F_0+F_1,\, G} \psi_0 \triangledown \psi_1$	h-SUM
19	$f\colon\ \varphi_i \rightarrow_{F,G_i} \psi_i \;\; (i = 0,1)$	$\equiv$	$f\colon\ \varphi_0 \vartriangle \varphi_1 \rightarrow_{F,\, G_0\times G_1} \psi_0 \vartriangle \psi_1$	h-PROD
20	$\left.\begin{array}{l} \varepsilon\colon\ H \stackrel{.}{\rightarrow} F \;\wedge\; \eta\colon\ G \stackrel{.}{\rightarrow} J \\ f\colon\ \varphi \rightarrow_{F,G} \psi \end{array}\right\}$	$\Rightarrow$	$f\colon\ \varepsilon \mathbin{;} \varphi \mathbin{;} \eta \rightarrow_{H,J} \varepsilon \mathbin{;} \psi \mathbin{;} \eta$	homo-NTRF
21	$f\colon\ H\varphi \rightarrow_{F,G} J\varphi$	$\equiv$	$\varphi\colon\ Ff \rightarrow_{H,J} Gf$	homo-SWAP
22	If $F = I$ or $G = I$, then:			
	φ is an F,G-dialgebra	$\equiv$	$\varphi\colon\ F\varphi \rightarrow_{F,G} G\varphi$	homo-TRIV

The proofs are all rather simple; in most cases it suffices to unfold the arrow notation into the equation and use functor properties. Law homo-SUM is proved in paragraph 2 for the special case that $G = I$, and in paragraph 11 for a specific example. In the proof of homo-NTRF the naturality of ε and η is used, of course.

Law homo-COMPOSE states that homomorphisms compose nicely; together with homo-ID it asserts that F, G-dialgebras form a category; the category is called $\mathcal{D}iAlg(F,G)$ and defined in paragraph 4.

Law homo-FTR2 states that functors $H\colon \mathcal{C} \to \mathcal{C}$ that commute with both F and G can be considered (or are) also functors typed $\mathcal{D}iAlg(F,G) \to \mathcal{D}iAlg(F,G)$. In particular, $F\colon \mathcal{A}lg(F) \to \mathcal{A}lg(F)$.

The condition in homo-NTRF is stronger than necessary; it is sufficient if $Hx \mathbin{;} \varepsilon = \varepsilon \mathbin{;} Fx$ for $x = f$ only, and similarly for η. Actually, this law states that the mapping $\chi \mapsto \varepsilon \mathbin{;} \chi \mathbin{;} \eta$ is a transformer, a notion that plays a major rôle in Chapter 5.

Law homo-SWAP is less general than it seems upon first sight: in order that one side is well defined, the functors cannot be completely unrelated to each other.

All of the laws specialise to algebras and co-algebras, of course, by taking $F = I$ or $G = I$.

23 On the arrow notation. The notation $f\colon \varphi \to_{F,G} \psi$ as an abbreviation for the equation $\varphi \mathbin{;} Gf = Ff \mathbin{;} \psi$ works pretty well: it avoids the duplication of f and it makes the *pattern* of the equation into a single symbol. However, sometimes the unabbreviated formula may be much clearer than that with the arrow notation. As an example, the following law becomes almost trivial by just unfolding the arrow.

24 $$f\colon \varphi \to_{F,G} \psi \ \wedge\ f \text{ is an } I\text{-algebra} \ \Rightarrow$$
$$f : \ Ff \mathbin{;} \varphi \mathbin{;} Gf \ \to_{F,G} \ Ff \mathbin{;} \psi \mathbin{;} Gf \qquad \text{homo-ADHOC}$$

(Using the arrow notation only, homo-NTRF with the weakened premise may be used as the main step in the proof.)

25 Example. (Use of the laws) Suppose that *inits*, *tails*: $Lf \to_I LLf$ for all f, and also *flatten*: $LLf \to_I Lf$. Define $segs = inits \mathbin{;} Ltails \mathbin{;} flatten$. Then

$$Lf \mathbin{;} segs \quad = \quad segs \mathbin{;} LLf$$

for all f. The proof is simple, thanks to the notation and laws for homomorphisms.

$segs\colon Lf \to_I LLf$

$\Leftarrow$ unfold *segs*, law homo-COMPOSE 15

$inits\colon Lf \to_I LLf,\ \ Ltails\colon LLf \to_I LLLf,\ \ flatten\colon LLLf \to_I LLf$

$\Leftarrow$ for the middle conjunct: homo-FTR2 17;
given equations (taking $f := Lf$ for the right conjunct)

true.

Actually, the proof can be simplified further by noting that *inits*, *tails*, and *flatten* are natural transformations, and so is *segs*. See paragraph A.13. □

3b Initiality and catamorphisms

We explain here informally what initiality in $\mathcal{A}lg(F)$ means, and also finality in $\mathcal{C}o\mathcal{A}lg(F)$. Initiality or finality in $\mathcal{D}i\mathcal{A}lg(F,G)$ in general has, as far as I know, no immediate practical relevance; moreover, I know of no simple conditions on F, G that ensure that an initial or final object in $\mathcal{D}i\mathcal{A}lg(F,G)$ exists.

26 Initiality: catamorphisms. Suppose that $\mathcal{A}lg(F)$ has an initial object, α say. Fix this α throughout what follows, and write $(\!|\varphi|\!)_F$ or just $(\!|\varphi|\!)$ for $(\!|\alpha - \varphi|\!)_{\mathcal{A}lg(F)}$, the unique F-homomorphism from α to φ. This notation supposes that φ is an F-algebra:

$$\varphi \text{ is an } F\text{-algebra} \quad \Rightarrow \quad (\!|\varphi|\!): U\alpha \to U\varphi . \qquad \text{cata-TYPE}$$

Each morphism that can be written as $(\!|\varphi|\!)$ is called a **catamorphism**. Prefix *cata* is explained below in paragraph 30. The laws for α and $(\!|\ |\!)$ as explained in Chapter 2 work out as follows.

$$\begin{array}{llll}
\alpha ; x = Fx ; \varphi & \equiv & x = (\!|\varphi|\!) & \text{cata-CHARN} \\
\alpha ; (\!|\varphi|\!) = F(\!|\varphi|\!) ; \varphi & & & \text{cata-SELF} \\
id = (\!|\alpha|\!) & & & \text{cata-ID} \\
\alpha ; x = Fx ; \varphi \ \wedge \ \alpha ; y = Fy ; \varphi & \Rightarrow & x = y & \text{cata-UNIQ} \\
\varphi ; x = Fx ; \psi & \Rightarrow & (\!|\varphi|\!) ; x = (\!|\psi|\!) & \text{cata-FUSION}
\end{array}$$

Most equations merely express that x is a homomorphism of a certain type. The premise of cata-FUSION for instance can be formulated as $x\colon \varphi \to_F \psi$. The arrow notation makes it easier to apply the homo-Laws discussed in paragraph 13. Using the arrow notation the laws read as follows.

$$\begin{array}{llll}
x\colon \alpha \to_F \varphi & \equiv & x = (\!|\varphi|\!) & \text{cata-CHARN} \\
(\!|\varphi|\!)\colon \alpha \to_F \varphi & & & \text{cata-SELF} \\
x, y\colon \alpha \to_F \varphi & \Rightarrow & x = y & \text{cata-UNIQ} \\
x\colon \varphi \to_F \psi & \Rightarrow & (\!|\varphi|\!) ; x = (\!|\psi|\!) & \text{cata-FUSION}
\end{array}$$

Here is yet another law that is specific for algebras (and cannot be formulated for initiality in general). For arbitrary G-algebra φ and initial G-algebra β,

$$\varepsilon\colon F \dot{\to} G \quad \Rightarrow \quad (\!|\varepsilon ; \beta|\!)_F ; (\!|\varphi|\!)_G \ = \ (\!|\varepsilon ; \varphi|\!)_F \qquad \text{cata-COMPOSE}$$

Another reading of the law is this: for $\varepsilon\colon F \dot{\to} G$ the composite $(\!|\varepsilon ; \beta|\!)_F ; f$ is a catamorphism whenever f is a G-catamorphism. The proof of cata-COMPOSE is simple:

$$\begin{array}{ll}
& (\!|\varepsilon ; \beta|\!)_F ; (\!|\varphi|\!)_G \ = \ (\!|\varepsilon ; \varphi|\!)_F \\
\Leftarrow & \quad \text{cata-FUSION}
\end{array}$$

$([\varphi])_G\colon \varepsilon \mathbin{;} \beta \rightarrow_F \varepsilon \mathbin{;} \varphi$

$\Leftarrow$ homo-NTRF 20, $\varepsilon\colon F \dot{\rightarrow} G$

$([\varphi])_G\colon \beta \rightarrow_G \varphi$

$\equiv$ cata-SELF

true.

27 Interpretation. Let a be the carrier of α, $a = U\alpha$. In paragraph 31 we shall show that α is an isomorphism $\alpha\colon Fa \cong a$, the inverse of which we denote $\alpha^{\cup}$. We might call α a "constructor", since in $\mathcal{S}et$ operation α is a bijection and each element of a can be obtained as the outcome of α for precisely one argument, called the "constituents" of the element. The inverse $\alpha^{\cup}$ is then a "destructor"; it maps each element of a into its constituents in Fa. Now look at the left hand side of cata-CHARN:

28 $\alpha \mathbin{;} x \;=\; Fx \mathbin{;} \varphi$ or, equivalently,
29 $x \;=\; \alpha^{\cup} \mathbin{;} Fx \mathbin{;} \varphi.$

Thus cata-CHARN says that this "inductive" equation has a unique solution for x. If α were not initial, an equation like 28 might have no or several solutions: in $\mathcal{S}et$, the equation

$$\alpha' \mathbin{;} x \;=\; Fx \mathbin{;} \varphi$$

has at least one solution for x only if α' is injective ("there is no confusion" and α' has a post-inverse), and the equation has at most one solution for x only if α' is surjective ("there is no junk" and α' has a pre-inverse).

Notice that equation 29 uses explicitly the destructor $\alpha^{\cup}$ to decompose the argument into its constituents, whereas the equivalent equation 28 uses 'pattern matching' (the α) as in functional languages.

The other laws have a similar informal interpretation. Law cata-UNIQ says that if two morphisms x and y both satisfy the same "inductive pattern", namely $x = \alpha^{\cup} \mathbin{;} Fx \mathbin{;} \varphi$ and $y = \alpha^{\cup} \mathbin{;} Fy \mathbin{;} \varphi$, then they are the same. Thus cata-UNIQ captures, in a sense, induction. Law cata-FUSION may also be read as giving a sufficient condition on x and φ in order that the composite $([\varphi]) \mathbin{;} x$ is a catamorphism.

30 'cata'. Meertens [46] has coined the name F**-catamorphism** for $([\varphi])_F$ ($\kappa\alpha\tau\alpha$ meaning 'downwards') since, interpreted as a computing agent, $([\varphi])$ descends along the structure of the argument (systematically replacing each α by φ, see example 39 below). So a catamorphism is nothing but a homomorphism on an initial algebra. It is useful to have a separate name, since in contrast to homomorphisms they are not closed under composition but do satisfy the laws listed above. In the literature on functional programming catamorphisms on cons lists are called *fold* or *iterate*.

31 Existence of $\alpha^{\cup}$. An initial F-algebra α is —up to isomorphism— a fixed point of F, that is, $F\alpha \cong \alpha$ in $\mathcal{A}lg(F)$. To prove this, we have to establish a pair x,y of morphisms in $\mathcal{A}lg(F)$ (F-algebra homomorphisms in $\mathcal{C}$),

$$\begin{array}{lll} x & : & F\alpha \rightarrow_F \alpha \\ y & : & \alpha \rightarrow_F F\alpha, \end{array}$$

that are each other's inverse. Law homo-TRIV 22 immediately implies that x is α. Law cata-CHARN implies that y is $(\!|F\alpha|\!)$. (The existence of a candidate for y is problematic for dialgebras in general.) It remains to show that these choices are each others inverse indeed. For this we argue:

$$\begin{array}{ll} & (\!|F\alpha|\!) \mathbin{;} \alpha = id \\ \equiv & \qquad \text{cata-ID} \\ & (\!|F\alpha|\!) \mathbin{;} \alpha = (\!|\alpha|\!) \\ \Leftarrow & \qquad \text{cata-FUSION} \\ & F\alpha \mathbin{;} \alpha = F\alpha \mathbin{;} \alpha \\ \equiv & \qquad \text{equality} \\ & \textsf{true}. \end{array}$$

So $(\!|F\alpha|\!)$ is a pre-inverse of α. It is a post-inverse too:

$$\begin{array}{ll} & \alpha \mathbin{;} (\!|F\alpha|\!) \\ = & \qquad \text{cata-SELF} \\ & F(\!|F\alpha|\!) \mathbin{;} F\alpha \\ = & \qquad \text{functor, above: } (\!|F\alpha|\!) \text{ is pre-inverse of } \alpha \\ & F\,id \\ = & \qquad \text{functor} \\ & id. \end{array}$$

As a corollary it follows that α: $Fa \cong a$ in $\mathcal{C}$, where $a = U\alpha$:

$$\begin{array}{lll} & & \alpha\colon FU\alpha \cong U\alpha \text{ in } \mathcal{C} \\ (*) & \equiv & \qquad U \text{ is the identity on morphisms, explained below: } UF = FU \\ & & U\alpha\colon UF\alpha \cong U\alpha \text{ in } \mathcal{C} \\ & \Leftarrow & \qquad \text{functor} \\ & & \alpha\colon F\alpha \cong \alpha \text{ in } \mathcal{A}lg(F) \\ & \equiv & \qquad \text{just shown} \\ & & \textsf{true}. \end{array}$$

For step $(*)$ recall from paragraph 13 that F: $\mathcal{C} \rightarrow \mathcal{C}$ implies F: $\mathcal{A}lg(F) \rightarrow \mathcal{A}lg(F)$ as well, so that both UF and FU make sense. The equality $UFf = FUf$ is now immediate since U is the identity on morphisms, and the equality $FU\varphi = UF\varphi$ is one of the functor axioms for F.

32 Existence of initial algebra μF. Recall that in each category all initial objects are isomorphic to each other, even with precisely one isomorphism between each pair. So we shall sometimes say 'the' initial F-algebra rather than 'an', assuming the choice to be arbitrary but fixed. We let μF denote that initial F-algebra, if one exists; in paragraph 61 we define the notation *sumtype*( ) as an extension of $\mu(\,)$. Variables α, β range over initial algebras (and also over final co-algebras).

There exist categories and endofunctors F for which there is no initial F-algebra. Yet, for *Set* and various order-enriched categories such as *CPO* the class of functors for which an initial algebra exists is quite large. The key to this result has been shown in paragraph 2.27: in each ω-category there exists an initial algebra for each ω-cocontinuous functor. All functors generated by the grammar

$$F ::= I \mid \underline{a} \mid F + F \mid F \times F \mid \text{type functor induced by } F_a$$

are ω-cocontinuous. (The induced type functor is defined in paragraph 54. Malcolm [42] has proved the result especially with regards to the last clause. We shall return to this in paragraph 6.5.)

33 Establishing initiality. In order to prove that an F-algebra α' is initial in $\mathcal{A}lg(F)$, it is required to define a function $(\!|\alpha' \rightarrow \_|\!)_F$ and to establish the validity of law cata-CHARN. There are several instances of such proofs in the sequel.

Sometimes another F-algebra α is known to be initial. (As remarked above, for specific categories and specific functors a construction of an initial algebra α is known.) In that case it suffices to prove that α and α' are isomorphic in $\mathcal{A}lg(F)$. Since cata-CHARN implies that $(\!|\alpha \rightarrow \alpha'|\!)_F$ is the the unique homomorphism from α to α', it suffices to establish a homomorphism $f\colon \alpha' \to \alpha$ that is both a pre- and a post-inverse of $(\!|\alpha \rightarrow \alpha'|\!)_F$. In our experience this method is less elegant than directly establishing cata-CHARN, not using α at all.

Examples (initial algebras and catamorphisms)

34 Naturals. Let $F = \underline{1} + I$. Recall the F-algebra $\alpha = zero \triangledown succ$ as explained in paragraph 1.12. It is initial in $\mathcal{A}lg(F)$. To prove this, we provide a definition for $(\!|zero \triangledown succ \rightarrow \_|\!)_F$, or briefly $(\!|\_|\!)$, and show the validity of cata-CHARN. So, let $\varphi = e \triangledown f\colon 1 + a \to a$ be arbitrary. We argue

$$\begin{array}{ll}
 & \alpha \,;\, x = Fx \,;\, \varphi \\
\equiv & \quad \text{definition } \alpha, \varphi, F \\
 & zero \triangledown succ \,;\, x = (\underline{1} + I)x \,;\, e \triangledown f \\
\equiv & \quad \text{functor, sum} \\
 & (zero \,;\, x) \triangledown (succ \,;\, x) = (id_1 \,;\, x) \triangledown (x \,;\, f) \\
\equiv & \quad \text{sum} \\
 & zero \,;\, x = e \ \wedge\ succ \,;\, x = x \,;\, f \\
\equiv & \quad \text{knowledge about the well-known set } nat \text{ and functions } zero, succ
\end{array}$$

$$x = \text{the function } n \mapsto f^n(e()) \qquad \text{where () denotes the sole member of } \imath.$$

Hence, defining $(\!|e \triangledown f|\!)$ as the right hand side of the last line, the above calculation establishes law cata-CHARN.

So, up to isomorphism $zero \triangledown succ$ is the same as μF, and if you did not know the operations $zero \triangledown succ$ and the set nat beforehand, we could define them now (or rather an isomorphic collection) as μF and $U\mu F$. In the sequel we assume that $zero \triangledown succ = \mu F$.

The inverse of α is $(\!|F\alpha|\!)$, see paragraph 31. So, the inverse of $zero \triangledown succ$ is $pred = (\!|F(zero \triangledown succ)|\!)_F$: $nat \to \imath + nat$. Working out this definition, we find:

$$\begin{array}{ll}
 & pred = (\!|F(zero \triangledown succ)|\!)_F \\
\equiv & \quad \text{cata-SELF} \\
 & zero \triangledown succ \,;\, pred = Fpred \,;\, F(zero \triangledown succ) \\
\equiv & \quad \text{functor, sum, definition } F \\
 & (zero \,;\, pred) \triangledown (succ \,;\, pred) = id + (pred \,;\, zero \triangledown succ) \\
\equiv & \quad \text{sum} \\
 & (zero \,;\, pred) \triangledown (succ \,;\, pred) = inl \triangledown (pred \,;\, zero \triangledown succ \,;\, inr) \\
\equiv & \quad \text{sum, } pred \text{ is inverse of } zero \triangledown succ \\
 & zero \,;\, pred = inl \ \wedge\ succ \,;\, pred = inr.
\end{array}$$

The latter equations clearly express that $pred$ is the inverse of $zero \triangledown succ$. Writing (n) for $zero \,;\, succ^n$, we have $(n{+}1) \,;\, pred = (n) \,;\, inr$.

35 Cons lists. Let $F = \underline{\imath} + \underline{a} \times I$. Recall the F-algebra $\alpha = nil \triangledown cons$ as explained in paragraph 1.12. It is initial in $\mathcal{A}lg(F)$. To prove this, we provide a definition for $(\!|nil \triangledown cons \to \_|\!)_F$ or briefly $(\!|\_|\!)$, and show the validity of law cata-CHARN. So, let $\varphi = e \triangledown \psi$: $Fb \to b$ be arbitrary. We argue

$$\begin{array}{ll}
 & \alpha \,;\, x = Fx \,;\, \varphi \\
\equiv & \quad \text{definition } \alpha, \varphi, F \\
 & nil \triangledown cons \,;\, x = (\underline{\imath} + \underline{a} \times I)x \,;\, e \triangledown \psi \\
\equiv & \quad \text{functor, sum} \\
 & (nil \,;\, x) \triangledown (cons \,;\, x) = (id_\imath \,;\, e) \triangledown (id_a \times x \,;\, \psi) \\
\equiv & \quad \text{sum} \\
 & nil \,;\, x = e \ \wedge\ cons \,;\, x = id \times x \,;\, \psi \\
\equiv & \quad \text{knowledge about the well-known set } La \text{ and functions } nil, cons \\
 & x = (\text{the function defined by } nil \,;\, x = e \ \wedge\ cons \,;\, x = id \times x \,;\, \psi).
\end{array}$$

Specifically, the function mentioned in the last line is

$$cons(a_0, \dots cons(a_{n-1}, nil())) \quad \mapsto \quad \psi(a_0, \dots \psi(a_{n-1}, e())),$$

for each $a_0, \dots, a_{n-1} \in a$. Hence, defining $(\!|e \triangledown \psi|\!)$ as that function, the above calculation establishes law cata-CHARN.

So, $nil \triangledown cons$ is up to isomorphism the same as μF, and if you did not know the operations $nil \triangledown cons$ and the set La beforehand, we could define them now (or rather an isomorphic collection) as μF and $U\mu F$. In the sequel we assume that $nil \triangledown cons = \mu F$.

36 Similarly the finite binary trees form an initial algebra, and so do the finite rose trees, and so on.

37 Enumerated types. Modern programming languages allow, amongst others, to define a new type by enumerating the elements of the type. As an example we show how to define a type *color* with three elements *red, white, blue*. To this end take $F = \underline{1} + \underline{1} + \underline{1}$, and $\alpha = U\mu F$, and define

$$\begin{array}{lcll} color & = & U\alpha & \\ red \triangledown white \triangledown blue & = & \alpha & : \ 1+1+1 \to color\,. \end{array}$$

Let $\varphi = f \triangledown g \triangledown h\colon Fa \to a$ be arbitrary, and consider the following equation for unknown $x\colon color \to a$.

$$\begin{array}{ll} & \alpha \,;\, x = Fx \,;\, \varphi \\ \equiv & \quad \text{definition } \alpha, \varphi, F \\ & red \triangledown white \triangledown blue \,;\, x = id + id + id \,;\, f \triangledown g \triangledown h \\ \equiv & \quad \text{sum} \\ & red \,;\, x = f \ \wedge \ white \,;\, x = g \ \wedge \ blue \,;\, x = h. \end{array}$$

Initiality of α says that these equations define function x uniquely. Hence, *color* is a set consisting of just threee elements, called *red*, *white*, and *blue*.

As a particularly important application one may define

$$\begin{array}{lcl} true \triangledown false & = & \mu(\underline{1} + \underline{1}) \\ bool & = & U\mu(\underline{1} + \underline{1}) \end{array}$$

so that

$$true, false \quad : \quad 1 \to bool\,.$$

38 Sum. Generalising the previous example, for each a, b the sum of a and b is an initial $\underline{a} + \underline{b}$-algebra. To see this, just observe that

$$inl_{a,b} \triangledown inr_{a,b} \,;\, x = (\underline{a} + \underline{b})x \,;\, f \triangledown g \quad \equiv \quad x = f \triangledown g\,.$$

So, cata-CHARN is valid with $F, \alpha, (\!|\alpha \to f \triangledown g|\!)_F \ := \ \underline{a}+\underline{b},\ inl_{a,b} \triangledown inr_{a,b},\ f \triangledown g$. As a corollary it follows that $(\!|f + g|\!)_{\underline{a}+\underline{b}} = f + g$, since by definition $f + g = (f \,;\, inl_{a,b}) \triangledown (g \,;\, inr_{a,b})$. (To be continued in paragraph 64. See also Section 5f.)

39 Repeated application of law cata-SELF gives for any φ of the right type

$$F^n\alpha \,;\, \dots \,;\, FF\alpha \,;\, F\alpha \,;\, \alpha \,;\, (\!|\varphi|\!) \quad = \quad FF^n(\!|\varphi|\!) \,;\, F^n\varphi \,;\, \dots \,;\, FF\varphi \,;\, F\varphi \,;\, \varphi\,.$$

This shows that "catamorphism $(\!|\varphi|\!)$ systematically replaces the constructor α by operation φ." Actually, both sides may be viewed as linearised notations of tree structured expressions (the tree-structure being determined by F). The left hand side has the components of α at the nodes (and $(\!|\varphi|\!)$ is applied to the entire expression), the right hand side has the components of φ at the nodes (and $(\!|\varphi|\!)$ has sunk to depth $n+1$). For intuitive understanding take, for example, $F = I\!I$ or better $F = \imath + I\!I$.

40 Take $F = \underline{1} + I$ and $zero \triangledown succ = \mu F$. For this choice the equation of the previous example is quite complicated to write down in a readable way as one equation. But it is easy to derive a similar equation. First notice that cata-SELF equivales

$$\begin{array}{lcl} zero \,;\, (\!|e \triangledown f|\!)_F & = & e \\ succ \,;\, (\!|e \triangledown f|\!)_F & = & (\!|e \triangledown f|\!)_F \,;\, f\,. \end{array}$$

Repeated application of the latter equation, and once using the former, gives

$$zero \,;\, succ \,;\, succ \,;\, \ldots \,;\, succ \,;\, (\!|e \triangledown f|\!)_F \;=\; e \,;\, f \,;\, f \,;\, \ldots \,;\, f$$

that is,

$$zero \,;\, succ^n \,;\, (\!|e \triangledown f|\!)_F \;=\; e \,;\, f^n\,.$$

This is just one of the 'paths' present in the equation of the previous example.

41 In $\mathcal{S}et$ the initial I-algebra has the empty set $\emptyset$ as carrier; the algebra itself is the identity $id_\emptyset$: $\emptyset \to \emptyset$. Indeed, for any φ: $a \to a$ law cata-CHARN holds true:

$$id_\emptyset \,;\, x = x \,;\, \varphi \quad\equiv\quad x = \text{(the unique morphism of type } \emptyset \to a)\,.$$

More generally, in each category the morphism id_0: $0 \to 0$ is the initial I-algebra, where 0 is the initial object. □

42 Finality and anamorphisms. By definition final co-algebras and anamorphisms are the dual notions of initial algebras and catamorphisms. The definitions and laws are obtained by the mechanical process of dualising. So we can be brief here. The notation νF denotes an arbitrary, but fixed, final F-co-algebra, assuming one exists.

Let F be an endofuctor. An F-co-algebra α is final in $CoAlg(F)$ iff law ana-CHARN holds, and therefore also the derived laws listed below. We write $[\![\varphi]\!]$ for $[\![\varphi \to \alpha]\!]_F$. This notation supposes that φ is an F-co-algebra:

$$\varphi \text{ is an } F\text{-co-algebra} \;\Rightarrow\; [\![\varphi]\!]\colon U\varphi \to U\alpha\,. \qquad \text{ana-TYPE}$$

The laws work out as follows.

$$\begin{array}{llr}
\varphi \,;\, Fx = x \,;\, \alpha \quad\equiv\quad x = [\![\varphi]\!] & & \text{ana-CHARN} \\
\varphi \,;\, F\,[\![\varphi]\!] = [\![\varphi]\!] \,;\, \alpha & & \text{ana-SELF} \\
id = [\![\alpha]\!] & & \text{ana-ID} \\
\varphi \,;\, Fx = x \,;\, \alpha \;\wedge\; \varphi \,;\, Fy = y \,;\, \alpha \;\Rightarrow\; x = y & & \text{ana-UNIQ} \\
\varphi \,;\, Fx = x \,;\, \psi \quad\Rightarrow\quad [\![\varphi]\!] = x \,;\, [\![\psi]\!] & & \text{ana-FUSION} \\
\varepsilon\colon F \dot{\to} G \quad\Rightarrow\quad [\![\varphi]\!]_F \,;\, [\![\nu F \,;\, \varepsilon]\!]_G = [\![\varphi \,;\, \varepsilon]\!]_G & & \text{ana-COMPOSE}
\end{array}$$

Notice that most equations merely express that x is a homomorphism of a certain type. The premise of ana-FUSION, for instance, can be written as $x\colon \varphi \succ\!\!-_F \psi$. As for initial algebras, a final co-algebra α is an isomorphism: $\alpha\colon a \cong Fa$ where $a = U\alpha$, the inverse of which we denote $\alpha^\cup$. We might call α a "destructor", since in $\mathcal{S}et$ operation α is a bijection and it maps each element of a one-one to a result, called the "constituents" of the element. The inverse $\alpha^\cup$ is then a "constructor"; it maps each collection of constituents in Fa onto the assembled element in a. Now look at the left hand side of ana-CHARN:

43 $\varphi \mathbin{;} Fx \;=\; x \mathbin{;} \alpha$ or, equivalently,
44 $x \;=\; \varphi \mathbin{;} Fx \mathbin{;} \alpha^\cup$.

Thus ana-CHARN says that this equation has a unique solution for x. Equation 43 tells that the *destruction* of the result of x (the right hand side) can be computed as given in the left hand side.

Equation 44 uses explicitly the constructor $\alpha^\cup$ to compose the intermediate results (constituents) into the result, whereas the equivalent equation 43 uses 'pattern matching' (the α) *on the result* in the right hand side. This type of definition, and algebra, is far less known than that for initial algebras.

The other laws have a similar interpretation. Morphism $[\!(\varphi)\!]_F$ is called **anamorphism** ($\alpha\nu\alpha$ meaning 'upwards'); the name is due to Erik Meijer.

Examples (final co-algebras and anamorphisms)

45 In $\mathcal{S}et$ 'the' final I-co-algebra has 'the' one-point set 1 as carrier; the algebra itself is the identity $id_1\colon 1 \to 1$. Indeed, for each $\varphi\colon a \to a$ law ana-CHARN holds true:

$$\varphi \mathbin{;} x = x \mathbin{;} id_1 \quad\equiv\quad x = (\text{the unique morphism of type } a \to 1)\,.$$

More generally, in each category morphism $id_1\colon 1 \to 1$ is the final I-co-algebra, where 1 is the initial object. Actually, this is nothing but the dual of the observation in paragraph 41.

46 Streams. Let $F = \underline{a} \times I$. Recall the F-co-algebra $\alpha = hd \vartriangle tl\colon Sa \to FSa$ as explained in paragraph 1.12. It is final in $CoAlg(F)$. To prove this, we provide a definition for $[\!(\_ \to hd \vartriangle tl)\!]_F$ or briefly $[\!(\_)\!]$, and show the validity of law ana-CHARN. So, let $\varphi = e \vartriangle f\colon b \to Fb$ be arbitrary. We argue

$$\begin{array}{lll}
 & & \varphi \mathbin{;} Fx = x \mathbin{;} \alpha \\
 & \equiv & \quad \text{definition } \alpha, \varphi, F \\
 & & e \vartriangle f \mathbin{;} id_a \times x = x \mathbin{;} hd \vartriangle tl \\
 & \equiv & \quad \text{product, functor} \\
 & & e = x \mathbin{;} hd \;\wedge\; f \mathbin{;} x = x \mathbin{;} tl \\
(*) & \equiv & \quad \text{knowledge of set } Sa \text{ and functions } hd, tl \\
 & & x = (\text{the function defined by } e = x \mathbin{;} hd \;\wedge\; f \mathbin{;} x = x \mathbin{;} tl).
\end{array}$$

Step $(*)$ is justified since one can prove by induction on n that if x satisfies the two equations, then

$$x \mathbin{;} tl^n \mathbin{;} hd = f^n \mathbin{;} e\,.$$

So the outcome of x is expressed in a way not involving x, and therefore function x itself is well defined. Specifically, for each $b_0 \in b$

$$x(b_0) \quad = \quad [e(b_0), e(f(b_0)), \ldots, e(f^n(b_0)), \ldots]\,.$$

Hence, defining $[\![e \vartriangle f]\!]$ as that function x, the above calculation establishes law ana-CHARN.

So, $hd \vartriangle tl$ is up to isomorphism the same as νF, and if you did not know the operations $hd \vartriangle tl$ and the set Sa beforehand, we could define them now (or rather an isomorphic collection) as νF and $U\nu F$. In the sequel we assume that $hd \vartriangle tl = \nu F$.

47 Cons′ lists. Take $F = \underline{1} + \underline{a} \times I$. This is the functor for cons lists over a, that is, the initial F-algebra is $nil \triangledown cons$, see paragraph 35. Recall the F-co-algebra $destruct'$: $L'a \to FL'a$ as explained in paragraph 1.12. It is final in $CoAlg(F)$. To prove this, we provide a definition for $[\![destruct' \to \_]\!]_F$ or briefly $[\![\_]\!]$, and show the validity of law ana-CHARN. So, let φ: $b \to Fb$ be arbitrary. We shall argue informally that the equation

$$\varphi \mathbin{;} Fx \quad = \quad x \mathbin{;} destruct'$$

fully determines the outcome of x, and therefore has one solution for x, thus establishing law ana-CHARN (even though we are not very specific about $[\![\_]\!]$). The only informality in our argument is the claim that the infinite sequence

$$\begin{array}{lcl}
x \mathbin{;} ! & : & b \to 1 \\
x \mathbin{;} destruct' \mathbin{;} F! & : & b \to 1 + a \times 1 \\
x \mathbin{;} destruct' \mathbin{;} F destruct' \mathbin{;} F^2! & : & b \to 1 + a \times (1 + a \times 1) \\
\vdots & & \\
x \mathbin{;} F^0 destruct' \mathbin{;} \ldots \mathbin{;} F^n destruct' \mathbin{;} FF^n! & : & b \to F\,F^n\,1 \\
\vdots & &
\end{array}$$

determines x completely. Accepting the claim, the reasoning is straightforward. By induction on n it is easily shown that, for all n,

$$x \mathbin{;} F^0 destruct' \mathbin{;} \ldots \mathbin{;} F^n destruct' \mathbin{;} FF^n! \quad = \quad F^0\varphi \mathbin{;} \ldots \mathbin{;} F^n\varphi \mathbin{;} FF^n!\,.$$

So each of the functions in the list can be written as an expression in terms of known functions, not involving x.

Since $destruct'$ is final, it has an inverse $destruct'^{\cup}$, and so $nil' \triangledown cons' = destruct'^{\cup}$ is an F-algebra. (This equation complies with the explanation in paragraph 1.12.) It is not an initial one: for example, the equations

$$nil' \mathbin{;} x \quad = \quad nil$$

$$cons' \mathbin{;} x \quad = \quad id \times x \mathbin{;} cons$$

that is,

$$x \quad = \quad destruct' \mathbin{;} id + id \times x \mathbin{;} nil \triangledown cons$$

have no solution x: $L'a \to La$. Indeed, the typing implies that x maps an *infinite* list onto a *finite* list, but the equations imply that each result has the *same* length as its argument. Also, for finite sets a containing at least two elements the cardinality of La is countably infinite, whereas that of $L'a$ is uncountable; hence the carriers are not isomorphic, implying that the algebras are not isomorphic in $\mathcal{A}lg(F)$.

48 Iterate for streams. For arbitrary f: $a \to a$ function f-iterate, denoted f^ω, is a function that yields a stream of all iterated applications of f to the argument:

$$f^\omega x \quad = \quad [x, fx, f^2x, f^3x, \ldots].$$

A definition as an anamorphism is derived as follows. Put $F = \underline{a} \times I$, the functor for streams over a. Then

$$\begin{array}{ll}
& f^\omega \mathbin{;} hd = id_a \;\wedge\; f^\omega \mathbin{;} tl = f \mathbin{;} f^\omega \\
\equiv & \quad \text{product} \\
& (f^\omega \mathbin{;} hd) \vartriangle (f^\omega \mathbin{;} tl) = id_a \vartriangle (f \mathbin{;} f^\omega) \\
\equiv & \quad \text{product} \\
& f^\omega \mathbin{;} hd \vartriangle tl = id_a \vartriangle f \mathbin{;} id_a \times f^\omega \\
\equiv & \quad \text{definition } F\text{, ana-}\textsc{Charn}\text{, interchanging left and right hand side} \\
& f^\omega = [\!(id \vartriangle f)\!]_F.
\end{array}$$

As an example, the streams *nats*, *ones*, and *nils* are now readily defined.

$$\begin{array}{lclcl}
nats & = & zero \mathbin{;} succ^\omega & : & 1 \to Snat \\
ones & = & one \mathbin{;} id^\omega & : & 1 \to Snat \\
nils & = & nil \mathbin{;} id^\omega & : & 1 \to SLa.
\end{array}$$

49 Generalised iterate. Continuing the previous example, the generalisation of $\times$ to an arbitrary bifunctor $\dagger$, and of $\mathcal{S}et$ to an arbitrary category, suggests itself when $id \vartriangle f$ is rewritten as follows.

$$id \vartriangle f \quad = \quad split \mathbin{;} id \times f$$

where

$$split \quad = \quad id \vartriangle id \quad = \quad split_\times \quad : \quad I \dot{\to} I \times I.$$

Indeed, let the default category and $\dagger$ be arbitrary, and suppose that there exists a natural transformation

$$split_\dagger \quad : \quad I \dot{\to} I \dagger I.$$

For each a and f: $a \to a$ morphism f**-iterate**, denoted f^ω, is defined as follows.

$$F \quad = \quad \underline{a} \dagger I$$

$$f^\omega \quad = \quad [\![split_\dagger \,;\, Ff]\!]_F \quad : \quad a \to U\nu F\,.$$

Notice that $split_\dagger$ is not necessarily unique, and hence the f^ω so defined depends on the choice for $split_\dagger$. (Moreover, the entire construction is natural in I-algebra f. This is formally shown in paragraph 66.)

50 Iteration for cons′ lists. Let us specialise the general iteration construct to cons′ lists. The functor for cons′ lists over a is

$$\begin{array}{lcl} F & = & \underline{1} + \underline{a} \times I \\ & = & \underline{a} \dagger I \end{array}$$

where

$$x \dagger y \quad = \quad id_1 + x \times y \qquad \text{for morphisms } x, y\,.$$

Take

$$split_\dagger \quad = \quad split_\times \,;\, inr \quad : \quad I \dot\to \underline{1} + I \times I \quad = \quad I \dot\to I \dagger I\,.$$

and let a and f: $a \to a$ be arbitrary. Then f-iterate $f^{\omega'}$: $a \to L'a$ specialises as follow.

$$\begin{array}{ll}
 & f^{\omega'} \\
= & \quad \text{definition iterate, } split_\dagger \text{ and } \dagger \\
 & [\![split_\times \,;\, inr \,;\, id_1 + id_a \times f]\!] \\
= & \quad \text{sum} \\
 & [\![split_\times \,;\, id \times f \,;\, inr]\!] \\
= & \quad \text{definition } split_\times = id \vartriangle id \text{, product} \\
 & [\![id_a \vartriangle f \,;\, inr]\!] \\
= & \quad \text{ana-S{\scriptsize ELF}, invertibility } destruct' = \nu F \\
 & id \vartriangle f \,;\, inr \,;\, Ff^{\omega'} \,;\, destruct'^{\cup} \\
= & \quad \text{definition } F\text{, } nil' \triangledown cons' = destruct'^{\cup} \\
 & id \vartriangle f \,;\, inr \,;\, id + id \times f^{\omega'} \,;\, nil' \triangledown cons' \\
= & \quad \text{sum} \\
 & id \vartriangle f \,;\, id \times f^{\omega'} \,;\, inr \,;\, nil' \triangledown cons' \\
= & \quad \text{product, sum} \\
 & id \vartriangle (f \,;\, f^{\omega'}) \,;\, cons'.
\end{array}$$

Now the cons′ list $nats'$ of natural numbers is defined

$$nats' \quad = \quad zero \,;\, succ^{\omega'} \quad : \quad nat \to L'nat\,.$$

51 List of predecessors. Take $F = \underline{1} + I$, being the functor for the naturals, and $G = \underline{1} + \underline{nat} \times I$ the functor for cons and cons′ lists over nat. We wish to express the cons and cons′ list of all predecessors of a nat argument as a cata- and anamorphism, respectively.

For *preds*: $nat \rightarrow Lnat$ we argue as follows. We start with equations that express the desired outcome of *preds* with induction on the $zero \triangledown cons$ structure of the argument. Hence in $\mathcal{S}et$ they *define* *preds* uniquely.

$$\begin{array}{ll}
 & zero \,;\, preds = nil \\
 & succ \,;\, preds = id \vartriangle preds \,;\, cons \\
\equiv & \qquad \text{sum} \\
 & (zero \,;\, preds) \triangledown (succ \,;\, preds) = nil \triangledown (id \vartriangle preds \,;\, cons) \\
\equiv & \qquad \text{sum, definition } F \\
 & zero \triangledown succ \,;\, preds = F(id \vartriangle preds) \,;\, nil \triangledown cons \\
\equiv & \qquad \text{putting } \alpha = zero \triangledown succ \text{ and } \varphi = nil \triangledown cons \\
 & \alpha \,;\, preds = F(id \vartriangle preds) \,;\, \varphi.
\end{array}$$

This equation has almost the form of the equations for F-catamorphisms. The only difference is the appearance of $id \vartriangle preds$ in the right hand side instead of just *preds*. So it is not obvious that *preds* is a catamorphism. This type of equations has been studied by Meertens [49] and the morphisms so defined are called *paramorphisms*; specifically, *preds* is an F-paramorphism from $zero \triangledown succ$ to $nil \triangledown cons$. (Actually, not every paramorphism is a catamorphism, but this one is.) We will discuss paramorphisms briefly in Section 4b.

For *preds'*: $nat \rightarrow L'nat$ we argue as follow. Again the top line of the following calculation is taken for granted, and at least in $\mathcal{S}et$ it *defines* *preds'* as a total function. Notice the correspondence with the equations and calculation for *preds*.

$$\begin{array}{ll}
 & zero \,;\, preds' = nil' \\
 & succ \,;\, preds' = id \vartriangle preds' \,;\, cons' \\
\equiv & \qquad \text{as above: sum, definition } F \\
 & zero \triangledown succ \,;\, preds' = F(id \vartriangle preds') \,;\, nil' \triangledown cons' \\
\equiv & \qquad \text{invertibility } pred = (zero \triangledown succ)^{\cup} \text{ and } nil' \triangledown cons' = destruct'^{\cup} \\
 & preds' \,;\, destruct' = pred \,;\, F(id \vartriangle preds') \\
\equiv & \qquad \text{aiming at ana-}\textsc{Charn}\text{ interchange lhs and rhs;} \\
 & \qquad \text{for all } f\text{: } F(id \vartriangle f) = F split \,;\, Gf \\
 & pred \,;\, F split \,;\, G preds' = preds' \,;\, destruct' \\
\equiv & \qquad \text{ana-}\textsc{Charn}\text{, with } F, \alpha, \varphi := G,\ destruct',\ pred \,;\, F split \\
 & preds' = [\![pred \,;\, F split]\!]_G.
\end{array}$$

So *pred'* is an anamorphism indeed.

Moreover, the above reasoning is readily generalised to arbitrary categories and functor F, provided that μF and so on exist. Specifically, define for arbitrary F and $\alpha = \mu F$

$$G \quad = \quad F \circ \underline{U\alpha} \times I$$

and

$$\begin{array}{lcll} preds & = & \text{the } F\text{-paramorphism from } \alpha \text{ to } \mu G & : U\alpha \to U\mu G \\ preds' & = & [\![\alpha^{\cup} \mathbin{;} F\,split]\!]_G & : U\alpha \to U\nu G\,. \end{array}$$

Then, for *preds* by its very definition and for *preds'* by the same calculation as above, it follows that

$$\begin{array}{lcl} \alpha \mathbin{;} pred & = & F(id \vartriangle preds) \mathbin{;} \mu G \\ \alpha \mathbin{;} preds' & = & F(id \vartriangle preds') \mathbin{;} (\nu G)^{\cup}. \end{array}$$

52 Until. Functions $f^{\omega'}$ and *nats'* produce infinite cons′ lists, whereas *preds'* produces finite cons′ lists. A possibly finite, possibly infinite cons′ list is produced by an until construct: f until p is a cons′ list containing all repeated applications of f until predicate p holds.

Define in $\mathcal{S}et$ for predicate p on a the function $p?$: $a \to a + a$ by

$$\begin{array}{lcll} p?(x) & = & inl(x) & \text{if } p(x) \text{ holds} \\ & = & inr(x) & \text{if } p(x) \text{ doesn't hold}\,. \end{array}$$

A construction of $p?$: $a \to a + a$ from p: $a \to \imath + \imath$ in arbitrary categories is not possible in general. Now, for arbitrary f: $a \to a$ the until-construct is defined as follows.

$$\begin{array}{lcl} f \text{ until } p & : & a \to L'a \\ f \text{ until } p & = & p? \mathbin{;} (!_a \mathbin{;} nil') \triangledown (id \vartriangle (f \mathbin{;} f \text{ until } p) \mathbin{;} cons')\,. \end{array}$$

In spite of the recursion this is a proper definition since it is equivalent to defining f until p as an anamorphism (like in the previous example, $G = \underline{\imath} + \underline{a} \times I$):

$$\begin{array}{ll} & f \text{ until } p \;=\; p? \mathbin{;} (!_a \mathbin{;} nil') \;\triangledown\; (id \vartriangle (f \mathbin{;} f \text{ until } p) \mathbin{;} cons') \\ \equiv & \qquad \text{product and sum} \\ & f \text{ until } p \;=\; p? \mathbin{;} !_a + (id \vartriangle f) \mathbin{;} id + id \times (f \text{ until } p) \mathbin{;} nil' \triangledown cons' \\ \equiv & \qquad \text{definition } G\text{, invertibility } nil' \triangledown cons' = destruct'^{\cup} \\ & f \text{ until } p \mathbin{;} destruct' \;=\; p? \mathbin{;} !_a + (id \vartriangle f) \mathbin{;} G(f \text{ until } p) \\ \equiv & \qquad \text{ana-}\textsc{Charn}\text{, interchanging lhs and rhs} \\ & f \text{ until } p \;=\; [\![p? \mathbin{;} !_a + (id \vartriangle f)]\!]_G. \end{array}$$

Apart from the construction of $p?$ out of p, the reasoning is completely general and applicable in an arbitrary category, provided of course that $\nu G = L'a$ and so on exist. □

3c Type functors (formerly map functors)

53 Map for lists. For the datatype of lists the so-called *map* is well known and frequently used; see for example Bird [9]. Recall the algebra of cons lists over a where, now, a is considered to be a parameter rather than a fixed set, and actually *nil*, *cons* are functions of a:

$$nil_a \triangledown cons_a \quad = \quad \mu F_a \quad : \quad F_a La \to La\,.$$

Writing Lf for the well-known function *map f* it follows that

$$Lf \quad : \quad La \to Lb \quad \text{for } f\colon a \to b$$

and

$$\begin{aligned} L\mathit{id}_a &= \mathit{id}_{La} \\ L(f \mathbin{;} g) &= Lf \mathbin{;} Lg\,. \end{aligned}$$

These three statements together precisely assert that L is a functor. The omission of the subscripts to *nil* and *cons* is now *formally* justified if $\mathit{nil} \triangledown \mathit{cons}$ is a natural transformation typed

$$\mathit{nil} \triangledown \mathit{cons} \quad : \quad \underline{1} + I \times L \dot{\to} L$$

that is,

$$\begin{aligned} \mathit{nil} &: \quad \underline{1} \dot{\to} L \\ \mathit{cons} &: \quad I \times L \dot{\to} L\,. \end{aligned}$$

This is true indeed, as observed in paragraph 1.12.

We shall show that these observations hold not only for the particular functor F_a for cons lists, but also for each *parameterised* functor F_a that depends *functorially* on a: it induces a functor M such that

$$\alpha_a \quad = \quad \mu F_a \qquad : \quad F_a Ma \to Ma$$

and, in fact,

$$\alpha \quad : \quad G \dot{\to} M \quad \text{where } Ga = F_a Ma\,.$$

Functor M is often called the *map* functor, extending the terminology for lists. We prefer the name *sumtype* functor or briefly *type functor*, since the word *map* is already in use for various meanings, and *sumtype* is quite well chosen as explained in paragraph 60. Malcolm [42] has already formulated and proved all the laws. My contribution is merely some extra subscripts at various places, some slight generalisation, and some more examples.

We discuss initial algebras in detail, and then dualise the results to final co-algebras, giving *prodtype functors*. Both sumtype and prodtype functors are called just *type functors*. The generalisation to arbitrary dialgebras is sketched in paragraph 55.

54 Defining the type functors. Take a category $\mathcal{C}$ as the default category. Let F_a be an endofunctor that depends functorially on a, that is, F_a can be written $\underline{a} \dagger I$ for some bifunctor $\dagger$. The most general typing is $\dagger\colon \mathcal{A} \times \mathcal{C} \to \mathcal{C}$ for some category $\mathcal{A}$. (In most of the examples $\mathcal{A} = \mathcal{C} \times \cdots \times \mathcal{C}$ with zero, one or more factors). Using the so-called *section notation* of functional languages we write $a\dagger$ for the endofunctor $\underline{a} \dagger I$,

$$\begin{aligned} (a\dagger)b &= a \dagger b \\ (a\dagger)f &= \mathit{id}_a \dagger f\,. \end{aligned}$$

(For cons lists over a we have $F_a = a\dagger$ where $x \dagger y = \mathit{id}_1 + x \times y$ for morphisms x, y.) Suppose that an initial $a\dagger$-algebra α_a exists for each a. Define a mapping M on objects by

$$Ma \quad = \quad U\alpha_a$$

so that

$$\alpha_a \quad : \quad a \dagger Ma \to Ma \quad = \quad F_a Ma \to Ma\,.$$

We wish to define M on morphisms as well, in such a way that M becomes a functor; apparently of type $\mathcal{A} \to \mathcal{C}$. The requirement '$Mf$: $Ma \to Mb$ whenever f: $a \to_{\mathcal{A}} b$' together with the wish 'Mf is a catamorphism' does not leave much room for the definition of Mf (at each step "there is only one thing you can do"):

Mf: $Ma \to Mb$

$\equiv$ guess and **define** (what else?) $Mf = (\!|\varphi|\!)_{a\dagger}$

$(\!|\varphi|\!)_{a\dagger}$: $Ma \to Mb$

$\Leftarrow$ cata-TYPE

φ: $a \dagger Mb \to Mb$

$\equiv$ since α_b: $b \dagger Mb \to Mb$, guess and **define** $\varphi = \psi \,;\, \alpha_b$

$\psi \,;\, \alpha_b$: $a \dagger Mb \to Mb$

$\Leftarrow$ typing of composition, type of α_b

ψ: $a \dagger Mb \to b \dagger Mb$

$\equiv$ since f: $a \to_{\mathcal{A}} b$, guess and **define** $\psi = f \dagger id_{Mb}$

$f \dagger id_{Mb}$: $a \dagger Mb \to b \dagger Mb$

$\equiv$ typing axiom for functors, type of f

true.

Thus we have derived a candidate definition for M: $\mathcal{A} \to \mathcal{C}$:

$$Ma \quad = \quad U\alpha_a \qquad\qquad \text{SUMTYPE}$$
$$Mf \quad = \quad (\!|f \dagger id_{Mb} \,;\, \alpha_b|\!)_{a\dagger} \;:\; Ma \to Mb, \qquad \text{for } f\text{: } a \to_{\mathcal{A}} b \qquad \text{SUMTYPE}$$

That the M so defined is a functor indeed is asserted in paragraph 57 below. It is called the **sumtype functor** induced by $\dagger$ (actually, M also depends on the particular α_a for each a). Notice that in $\mathcal{S}et$ function $f \dagger id_{Mb}$ subjects the a-constituents of its argument to f, and leaves its argument unaffected otherwise. In the term $f \dagger id$ the "functoriality" of $F_a = a\dagger$ in argument a is exploited by writing f instead of a as the left operand of $\dagger$.

The definition of the **prodtype functor** induced by $\dagger$ is dual to the sumtype functor. Suppose that $\alpha_a = \nu(a\dagger)$ exists for each a. Then

$$Ma \quad = \quad U\alpha_a \qquad\qquad \text{PRODTYPE}$$
$$Mf \quad = \quad [\!(\alpha_a \,;\, f \dagger id_{Ma})\!]_{b\dagger} \;:\; Ma \to Mb, \qquad \text{for } f\text{: } a \to_{\mathcal{A}} b \qquad \text{PRODTYPE}$$

In the former terminology M is the *map* for co-algebras induced by $\dagger$.

55 Type functors for dialgebras. The generalisation to arbitrary dialgebras is a bit tricky, and we shall nowhere use it. Consider bifunctors † and ‡ for which an initial $a\dagger, a\ddagger$-dialgebra α_a exists for all a. Then a tentative definition for the sumtype functor might read

$$\begin{array}{lcll} Ma & = & U\alpha_a & \\ Mf & = & (\!| f \dagger id_{Mb} \,;\, \alpha_b \,;\, f \ddagger id_{Mb} |\!)_{a\dagger, a\ddagger} \; : \; Ma \to Mb, & \text{for } f\text{: } a \to_{\mathcal{A}} b. \end{array}$$

But the expression enclosed by the brackets '(|' and '|)' is only well defined if $f \ddagger id_{Mb}$ has type $b \ddagger Mb \to a \ddagger Mb$ rather than $a \ddagger Mb \to b \ddagger Mb$, categorically: ‡ is contravariant in its left argument. The most general typing, then, is

$$\begin{array}{lcll} \dagger & : & \mathcal{A} \times \mathcal{B} \to \mathcal{C} & \\ \ddagger & : & \mathcal{A}^{op} \times \mathcal{B} \to \mathcal{C} & \text{(contravariance is indicated by } ^{op}\text{)} \\ U & : & \mathcal{D}iAlg(a\dagger, a\ddagger) \to \mathcal{B}, & \end{array}$$

so that

$$M \; : \; \mathcal{A} \to \mathcal{B}.$$

Dually, a prodtype functor for dialgebras exists only if † instead of ‡ is contravariant in its left argument.

56 Example. Cons lists. Take $a\dagger = \underline{1} + \underline{a} \times I$, the functor for cons lists, so that

$$nil_a \triangledown cons_a \;=\; \alpha_a \;=\; \mu(a\dagger) \;:\; a \dagger La \to La.$$

We shall prove that the sumtype functor and the well-known 'map' function are the same, when applied to an arbitrary $f\colon a \to b$. In the following calculation the top line gives the definition of L as the sumtype functor, the bottom line as the 'map' function.

$$\begin{array}{ll} & Lf = (\!| f \dagger id_{Lb} \,;\, \alpha_b |\!)_{a\dagger} \\ \equiv & \quad \text{cata-CHARN} \\ & \alpha_a \,;\, Lf \;=\; a \dagger Lf \,;\, f \dagger id_{Lb} \,;\, \alpha_b \\ \equiv & \quad \text{functor} \\ & \alpha_a \,;\, Lf \;=\; f \dagger Lf \,;\, \alpha_b \\ \equiv & \quad \text{definition of } \dagger\text{, and } nil\text{, } cons \\ & nil_a \triangledown cons_a \,;\, Lf \;=\; id_1 + f \times Lf \;\;;\;\; nil_b \triangledown cons_b \\ \equiv & \quad \text{sum, functor} \\ & nil_a \,;\, Lf \;=\; nil_b \\ & cons_a \,;\, Lf \;=\; f \times Lf \,;\, cons_b \;. \end{array}$$

Notice also that the two bottom lines assert

$$\begin{array}{lcl} nil & : & \underline{1} \dot{\to} L \\ cons & : & I \times L \dot{\to} L \end{array}$$

hence

$$nil \triangledown cons \;:\; \underline{1} + I \times L \dot{\to} L \;=\; I \dagger L \dot{\to} L.$$

This is also expressed in the third line of the calculation.

57 Laws for sumtype. Here are some useful laws. Let $\dagger\colon \mathcal{A} \times \mathcal{C} \to \mathcal{C}$ be a functor, and take $\mathcal{C}$ as the default category. In each law, functor $M\colon \mathcal{A} \to \mathcal{C}$ is the sumtype functor induced by $\dagger$, and $\alpha_a = \mu(a\dagger)$. Each $(\!|\_|\!)$ is short for $(\!|\alpha_a \dashv \_|\!)_{a\dagger}$ for some a, and f, g are morphisms in $\mathcal{A}$. Some explanation and explicit typing follows the enumeration.

$$\begin{array}{lcll}
M\,id_a & = & id_{Ma} & \text{sumtype-ID} \\
M(f \mathbin{;} g) & = & Mf \mathbin{;} Mg & \text{sumtype-DISTR} \\
Mf \mathbin{;} (\!|\varphi|\!) & = & (\!|f \dagger id \mathbin{;} \varphi|\!) & \text{type-cata-FUSION} \\
f\colon \varphi \to_{I\dagger I} \psi & \Rightarrow & f\colon (\!|\varphi|\!) \to_M (\!|\psi|\!) & \text{cata-TRANSFORMER} \\
\alpha & : & I \dagger M \dot\to M & \text{initial-NTRF} \\
\varphi\colon F \dagger G \dot\to G & \Rightarrow & (\!|\varphi|\!)\colon MF \dot\to G & \text{cata-NTRF}
\end{array}$$

Examples 62–67 illustrate some laws. Law sumtype-DISTR is well-known under the name map distribution; we shall invoke it by the more general phrase '(sumtype is a) functor'. Law type-cata-FUSION is Verwer's [72] factorisation theorem; this law asserts pre-fusion in contrast with cata-FUSION. Fully typed the law reads:

$$Mf \mathbin{;} (\!|\varphi|\!)_{b\dagger} \quad = \quad (\!|f \dagger id_c \mathbin{;} \varphi|\!)_{a\dagger} \ : \quad Ma \to c \quad \text{for } \varphi\colon b \dagger c \to c, \quad f\colon a \to_{\mathcal{A}} b.$$

Law cata-TRANSFORMER asserts that the mapping $\varphi \mapsto (\!|\varphi|\!)$ is a transformer; the notion of transformer is explained in Chapter 5. The ingredients of law cata-TRANSFORMER are typed as follows: $\mathcal{A} = \mathcal{C}$ and

$$\varphi\colon a \dagger a \to a, \qquad f\colon a \to_{\mathcal{A}} b, \qquad \psi\colon b \dagger b \to b$$

hence

$$(\!|\varphi|\!)_{a\dagger}\colon Ma \to a, \qquad (\!|\psi|\!)_{b\dagger}\colon Mb \to b.$$

58 Laws for prodtype. Dually, let M be the prodtype functor induced by $\dagger$, and let $\alpha_a = \nu(a\dagger)$. Then the preceding laws dualise to these.

$$\begin{array}{lcll}
M\,id_a & = & id_{Ma} & \text{prodtype-ID} \\
M(f \mathbin{;} g) & = & Mf \mathbin{;} Mg & \text{prodtype-DISTR} \\
[\!(\varphi)\!] \mathbin{;} Mf & = & [\!(\varphi \mathbin{;} f \dagger id)\!] & \text{ana-type-FUSION} \\
f\colon \varphi \rightarrowtail_{I\dagger I} \psi & \Rightarrow & f\colon [\!(\varphi)\!] \rightarrowtail_M [\!(\psi)\!] & \text{ana-TRANSFORMER} \\
\alpha & : & M \dot\to I \dagger M & \text{final-NTRF} \\
\varphi\colon F \dot\to G \dagger F & \Rightarrow & [\!(\varphi)\!]\colon F \dot\to MG & \text{ana-NTRF}
\end{array}$$

59 Proofs for the laws. For the —simple— proofs of the sumtype and prodtype laws we refer to Malcolm [42]. Law cata-COMPOSE 26 can be exploited for sumtype-DISTR, since $f \dagger id$, considered as the mapping $c \mapsto f \dagger id_c$, is a natural transformation of type $\underline{a} \dagger I \dot\to \underline{b} \dagger I$ whenever $f\colon a \to b$.

60 Syntactic sugar. Inspired by Hagino [29] and justified by initial-NTRF and cata-NTRF, Wraith [77] suggests the following concrete syntax for the simultaneous definition of the initial algebra (or its 'components'), the sumtype functor, and a name for the catamorphism brackets. For example, cons lists can be defined by the declaration

$$\begin{array}{l} \textbf{sumtype } Lx \textbf{ with } \mathit{rightreduce} \textbf{ is} \\ \quad \begin{array}{lcl} \mathit{nil} & : & 1 \to Lx \\ \mathit{cons} & : & x \times Lx \to Lx \,. \end{array} \end{array}$$

Apart from defining L and $nil, cons$ in the obvious way, the declaration also defines

$$\mathit{rightreduce}_a\ (e, f) \quad = \quad (\!| \mathit{nil}_a \triangledown \mathit{cons}_a \text{-} e \triangledown f |\!) \ : \ La \to b\,,$$

for all a, b and e: $1 \to b$ and f: $a \times b \to b$. More abstractly, the declaration

$$\textbf{sumtype } Mx \textbf{ with } \mathit{cata} \textbf{ is } \mathit{alpha}\colon\ x \dagger Mx \to Mx$$

or, without bound variables,

$$\textbf{sumtype } M \textbf{ with } \mathit{cata} \textbf{ is } \mathit{alpha}\colon\ I \dagger M \dot{\to} M$$

stands for the three definitions

$$\begin{array}{lcll} M & = & \text{sumtype functor induced by } \dagger & \\ \mathit{alpha}_a & = & \mu(a\dagger) \ : \ a \dagger Ma \to a & \text{for all } a \\ \mathit{cata}_a(\varphi) & = & (\!| \alpha_a \text{-} \varphi |\!)_{a\dagger} \ : \ Ma \to U\varphi & \text{for all } a\,. \end{array}$$

The name 'sumtype' is well-chosen since a sumtype generalises the categorical sum:

$$\begin{array}{l} \textbf{sumtype } \mathit{Sum}\ (x, y) \textbf{ with } \mathit{junc} \textbf{ is} \\ \quad \begin{array}{lcl} \mathit{inleft} & : & x \to \mathit{Sum}\ (x, y) \\ \mathit{inright} & : & y \to \mathit{Sum}\ (x, y) \end{array} \end{array}$$

defines the categorical sum:

$$\begin{array}{lcl} \mathit{Sum}\ (x, y) & = & x + y \\ \mathit{inleft},\ \mathit{inright} & = & \mathit{inl},\ \mathit{inr} \\ \mathit{junc}\ (f, g) & = & f \triangledown g\,. \end{array}$$

(This has been explained in detail in Examples 38 and 64.)

Dually, given a bifunctor $\dagger$, the prodtype functor M induced by $\dagger$, the polymorphic final co-algebra $alpha$ and anamorphism ana are defined by the single declaration

$$\textbf{prodtype } Mx \textbf{ with } \mathit{ana} \textbf{ is } \mathit{alpha}\colon\ Mx \to x \dagger Mx\,.$$

Using again an enumeration of the components, the declaration

$$\begin{array}{l} \textbf{prodtype } Sx \textbf{ with } \mathit{generate} \textbf{ is} \\ \quad \begin{array}{lcl} \mathit{hd} & : & Sx \to x \\ \mathit{tl} & : & Sx \to Sx \end{array} \end{array}$$

defines the final co-algebra of streams together with its prodtype functor.

Needless to say that a declaration itself does not guarantee the existence of the declared entities. There are, however, simple sufficient conditions on $\dagger$, see paragraph 32.

61 Sumtype, Prodtype. Let $\dagger$ be a bifunctor for which $\mu(a\dagger)$ exists for each a. Then *sumtype*$(\dagger)$ denotes the pair α, M where $\alpha\colon I \dagger M \dot{\to} M$ is the family $\mu(a\dagger)$, and M is the sumtype functor induced by $\dagger$. This is a convenient shorthand for the concrete syntax proposed above, since we use $(\!|\ |\!)$ as a generic name for the catamorphisms (writing $(\!|\alpha \to |\!)_F$ to avoid ambiguity). We say '$-, M = \mathit{sumtype}(\dagger)$' when it is only M that matters.

Dually, *prodtype*$(\dagger)$ denotes the pair α, M where $\alpha\colon M \dot{\to} I \dagger M$ is the family $\alpha_a = \nu(a\dagger)$ and M is the prodtype functor induced by $\dagger$.

Examples

62 (To illustrate sumtype-DISTR.) For arbitrary $-, M = \mathit{sumtype}(\dagger)$, define the "shape" of elements from Ma to be elements of $M1$:

$$\mathit{shape}_a \quad = \quad M\,!_a \quad : \quad Ma \to M1\,.$$

Then "each type functor preserves the shape", since for any $f\colon a \to b$

$$Mf \,;\, \mathit{shape}_b \quad = \quad Mf \,;\, M\,!_b \quad = \quad M(f \,;\, !_b) \quad = \quad M\,!_a \quad = \quad \mathit{shape}_a\,.$$

63 (To illustrate type-cata-FUSION 57.) In the context of cons lists and naturals, define

$$\mathit{sumsquares} \quad = \quad L\,sq \,;\, (\!|\mathit{zero} \triangledown \mathit{add}|\!) \quad : \mathit{Lnat} \to \mathit{nat}\,.$$

Then the composite *sumsquares* is a single catamorphism; usually this is proved by the Fold-Unfold technique. By type-cata-FUSION 57 in step $(*)$ we have

$$\begin{array}{ll}
 & \mathit{sumsquares} \\
= & \\
 & L\,sq \,;\, (\!|\mathit{zero} \triangledown \mathit{add}|\!) \\
(*) \quad = & \\
 & (\!|sq \dagger id \,;\, \mathit{zero} \triangledown \mathit{add}|\!) \\
= & \\
 & (\!|id + sq \times id \,;\, \mathit{zero} \triangledown \mathit{add}|\!) \\
= & \\
 & (\!|\mathit{zero} \triangledown (sq \times id \,;\, \mathit{add})|\!).
\end{array}$$

64 (Bifunctor $+$ is a sumtype functor.) Let $\dagger$ be defined by $(x,y) \dagger z = x + y$. In Example 38 we've shown that the initial $\underline{a} + \underline{b}$-algebra is $\mathit{inl}_{a,b} \triangledown \mathit{inr}_{a,b}$, with $f \triangledown g$ being the catamorphism for $\underline{a} + \underline{b}$-algebra $f \triangledown g\colon a + b \to c$. Now, the sum functor $+$ is just the sumtype functor induced by $\dagger$. To see this, let $-, M = \mathit{sumtype}(\dagger)$, and consider arbitrary objects a, b. Then

$$\begin{array}{ll}
 & M(a,b) \\
= & \qquad \text{SUMTYPE} \\
 & \text{carrier of the initial } \underline{a} + \underline{b}\text{-algebra } \mathit{inl}_{a,b} \triangledown \mathit{inr}_{a,b} \\
= & \qquad \text{observed in paragraph 38} \\
 & a + b,
\end{array}$$

and for arbitrary (f,g): $(a,b) \to_{C\times C} (c,d)$

$$\begin{array}{ll} & M(f,g) \\ = & \quad \text{SUMTYPE} \\ & (\!|(f,g) \dagger id \,;\, inl \triangledown inr|\!)_{(a,b)\dagger} \\ = & \quad \text{definition } \dagger\text{, sum} \\ & (\!|f+g|\!)_{(a,b)\dagger} \\ = & \quad \text{observed in paragraph 38} \\ & f+g. \end{array}$$

So, indeed, $M = +$.

65 (Zip, to illustrate ana-type-FUSION.) Consider once more the datatype of streams: $\alpha, M = prodtype(\times)$ and $hd \vartriangle tl = \alpha$. The well-know *zip* maps a pair of streams onto a stream of pairs, like a zipper, and '*zipwith-f*' applies in addition function f to each pair in the result stream of *zip*.

$$[a,\ldots],\ [b,\ldots] \ \overset{zip}{\mapsto}\ [(a,b),\ \ldots] \ \overset{Sf}{\mapsto}\ [f(a,b),\ \ldots].$$

The following calculations show that *zip* and *zipwith-f* are anamorphisms. As a preparation define

$$abide_F \quad = \quad F\,exl \vartriangle F\,exr$$

so that

$$abide_{I\!I} \quad = \quad I\!I\,exl \vartriangle I\!I\,exr\,.$$

The next line formalises the informal specification of *zip* above.

$$\begin{array}{ll} & zip \ = \ I\!I hd \vartriangle I\!I tl \,;\, id \times zip \,;\, (hd \vartriangle tl)_\cup \\ \equiv & \quad \text{definition } abide\text{, product, invertibility } \alpha = hd \vartriangle tl \\ & zip \,;\, \alpha \ = \ I\!I\alpha \,;\, abide_{I\!I} \,;\, id \times zip \\ \equiv & \quad \text{ana-CHARN exchanging the left and right hand sides} \\ & zip \ = \ [\!(I\!I\alpha \,;\, abide_{I\!I})\!]. \end{array}$$

Now, with the next line being the definition, we calculate

$$\begin{array}{ll} & zipwith\text{-}f \ = \ zip \,;\, Sf \\ \equiv & \quad \text{definiton } zip\text{, ana-type-FUSION} \\ & zipwith\text{-}f \ = \ [\!(I\!I\alpha \,;\, abide_{I\!I} \,;\, f \times id)\!]. \end{array}$$

66 (To illustrate ana-TRANSFORMER 58.) Recall the definition of f-iterate given in paragraph 49. For f: $a \to a$,

$$f^\omega \quad = \quad [\!(split_\dagger \,;\, id_a \dagger f)\!]_{a\dagger} \quad : \quad a \to Ma\,,$$

where $\alpha, M = prodtype(\dagger)$, and $split_\dagger \colon I \dashrightarrow I \dagger I$. Iteration is a transformer:

$$f\colon g \succ\!\!-_I h \quad\Rightarrow\quad f\colon g^\omega \succ\!\!-_M h^\omega \qquad \text{iterate-TRANSFORMER}$$

Here is the proof.

$$\begin{array}{ll}
 & f\colon [\![split_\dagger \mathbin{;} id \dagger g]\!] \succ\!\!-_M [\![split_\dagger \mathbin{;} id \dagger h]\!] \\
\Leftarrow & \quad \text{ana-TRANSFORMER 58} \\
 & f \;:\; split_\dagger \mathbin{;} id \dagger g \;\succ\!\!-_{I \dagger I}\; split_\dagger \mathbin{;} id \dagger h \\
\equiv & \quad \text{definition } \succ\!\!- \\
 & split_\dagger \mathbin{;} id \dagger g \mathbin{;} f \dagger f \;=\; f \mathbin{;} split_\dagger \mathbin{;} id \dagger h \\
\Leftarrow & \quad \text{naturality } split_\dagger\colon I \dashrightarrow I \dagger I\text{, Leibniz} \\
 & g \mathbin{;} f = f \mathbin{;} h.
\end{array}$$

Simple applications of iterate-TRANSFORMER are the equations

$$\begin{array}{lllll}
one \mathbin{;} id^\omega & = & id^\omega \mathbin{;} S\, one & = & ones \\
nil \mathbin{;} id^\omega & = & id^\omega \mathbin{;} S\, nil & = & nils\,.
\end{array}$$

With $\dagger = \times$, both sides denote a stream of 'ones', for arbitrary $one\colon 1 \to \mathbf{nat}$.

67 Continuing the previous example, with $\alpha, M = prodtype(\dagger)$, we derive another equation for f^ω:

$$\begin{array}{ll}
 & f^\omega \;=\; [\![split_\dagger \mathbin{;} id \dagger f]\!]_{\alpha\dagger} \\
\equiv & \quad \text{ana-CHARN} \\
 & f^\omega \;=\; split_\dagger \mathbin{;} id \dagger f \mathbin{;} id \dagger f^\omega \mathbin{;} \alpha^\cup \\
\equiv & \quad \text{functor, iterate-TRANSFORMER} \\
 & f^\omega \;=\; split_\dagger \mathbin{;} id \dagger (f^\omega \mathbin{;} Mf) \mathbin{;} \alpha^\cup.
\end{array}$$

Bird and Wadler [11, page 182–183] prove this very equation for the special case that $\dagger = \times$ (hence $\alpha = hd \vartriangle tl$) by means of the so-called *Take Lemma*:

$$\forall (n :: take\ n\ x = take\ n\ y) \quad\Rightarrow\quad x = y\,.$$

They say: "[The equation] cannot be proved by induction [on the structure of the argument] because there is no appropriate argument of *iterate* [the mapping $\_^\omega$] to do the induction over. Indeed, what we would like to do is establish the assertion by induction over the structure of the *result* of applying *iterate* [that is, the result of f^ω]." For this purpose they present and justify the Take Lemma. The proof above is just a few lines long (or 75 pages depending on what you consider to be part of the proof). □

Factorisation of the type functor

The sumtype functor M induced by $\dagger$ can be factored into a composition of two functors, namely $M = \mathit{Mu} \circ \dagger^{\mathsf{c}}$. Functor Mu is independent of $\dagger$ and occurs in other contexts as well; it is closely related to μ (mapping each functor into an initial algebra). Functor $\dagger^{\mathsf{c}}$ is the curry'd version of $\dagger$, and is therefore almost the same as $\dagger$. As a consequence, the phrase "Let M be the sumtype functor induced by $\dagger$" is in principle no longer needed: we can just write $\mathit{Mu} \circ \dagger^{\mathsf{c}}$ where M occurs. (Most of the results reported here were observed by Meertens [23].)

68 Functor *Mu*. Let $\mathcal{FC}$ denote the category of endofunctors on $\mathcal{C}$ whose morphisms are natural transformations; one might write $\mathcal{FC} = \mathcal{C} \to \mathcal{C}$. Let $\mathcal{F}'\mathcal{C}$ denote the full subcategory of $\mathcal{FC}$ whose objects are those functors F for which μF exists. Define functor Mu: $\mathcal{F}'\mathcal{C} \to \mathcal{C}$ as follows.

$$\begin{array}{lcll} \mathit{Mu}\,F & = & U\mu F, & \text{for } F \text{ in } \mathcal{F}'\mathcal{C} \\ \mathit{Mu}\,\varepsilon & = & (\!|\varepsilon \,;\, \mu G|\!)_F \;:\; \mathit{Mu}\,F \to \mathit{Mu}\,G, & \text{for } \varepsilon\colon F \dot{\to} G. \end{array}$$

Notice that $\varepsilon_{\mathit{Mu}G}$: $F\,\mathit{Mu}\,G \to G\,\mathit{Mu}\,G$ and μG: $G\,\mathit{Mu}\,G \to \mathit{Mu}\,G$, so that their composition has type $F\,\mathit{Mu}\,G \to \mathit{Mu}\,G$, hence $\mathit{Mu}\,\varepsilon$ has the type indicated. As usual we omit the subscript to ε. To prove that Mu distributes over composition, let ε: $F \dot{\to} G$ and η: $G \dot{\to} H$. Then $\varepsilon \,;\, \eta$: $F \dot{\to} H$, and

$$\begin{array}{cl} & \mathit{Mu}(\varepsilon \,;\, \eta) \\ = & \\ & (\!|\varepsilon \,;\, \eta \,;\, \mu H|\!)_F \\ = & \text{cata-COMPOSE 26, noting that } \eta \,;\, \mu H\colon G\,\mathit{Mu}\,H \to \mathit{Mu}\,H \\ & (\!|\varepsilon \,;\, \mu G|\!)_F \,;\, (\!|\eta \,;\, \mu H|\!)_G \\ = & \\ & \mathit{Mu}\,\varepsilon \,;\, \mathit{Mu}\,\eta. \end{array}$$

The equality $\mathit{Mu}\;\mathit{id} = \mathit{id}$ follows from the axiom for identity and law cata-ID. Thus Mu is a functor, Mu: $\mathcal{F}'\mathcal{C} \to \mathcal{C}$.

As a by-product it follows that the objects of the form $U\mu F$ together with the catamorphisms of the form $(\!|\varepsilon \,;\, \mu G|\!)_F$ for ε: $F \dot{\to} G$ form a subcategory $\mathcal{C}$.

Examples

69 A simple illustration for a catamorphism of the form $\mathit{Mu}\,\varepsilon$ is obtained by the choice

$$\begin{array}{lcll} F & = & \underline{1} + \underline{a} \times I & \text{hence } \mu F = \mathit{nil} \triangledown \mathit{cons} \\ G & = & \underline{1} + I & \text{hence } \mu G = \mathit{zero} \triangledown \mathit{succ} \\ \varepsilon & = & \mathit{id} + \mathit{exr} \;:\; F \dot{\to} G. & \end{array}$$

Now

$$\mathit{Mu}\,\varepsilon \;=\; (\!|\mathit{id} + \mathit{exr} \,;\, \mathit{zero} \triangledown \mathit{succ}|\!)_F \;=\; \mathit{size}\,.$$

70 Another choice is $F = G = \underline{a} + I\!I$, so that $Mu\,F = Mu\,G =$ the set of non-empty binary trees over a, and $\mu F = tip \triangledown join$. (Skip this example if you don't know the binary trees.) Take $\varepsilon = id + swap$: $F \dot{\to} G$ where $swap = swap_\times = exr \vartriangle exl$. Now

$$Mu\,\varepsilon \quad = \quad (\![id + swap \mathbin{;} tip \triangledown join]\!) \quad = \quad swap\text{-reduce} \quad = \quad reverse\,.$$

Since Mu is a functor, there is a simple proof that $reverse$ is its own inverse:

$$\begin{array}{ll} & reverse \mathbin{;} reverse \\ = & \\ & Mu\,\varepsilon \mathbin{;} Mu\,\varepsilon \\ = & \qquad \text{functor axiom} \\ & Mu(\varepsilon \mathbin{;} \varepsilon) \\ = & \qquad \text{easy: } swap \mathbin{;} swap = id\text{, hence } \varepsilon \mathbin{;} \varepsilon = id \\ & Mu\; id \\ = & \\ & id. \end{array}$$

By the remark following cata-COMPOSE 26, $reverse \mathbin{;} f$ is a catamorphism whenever f is.

71 Let $\dagger$ be a bifunctor and $\alpha, M = sumtype(\dagger)$. Take

$$\varepsilon \quad = \quad !_a \dagger id \;:\; \underline{a} \dagger I \dot{\to} \underline{1} \dagger I\,.$$

Then

$$Mu\,\varepsilon \quad = \quad (\![!_a \dagger id \mathbin{;} \alpha_1]\!)_{a\dagger} \quad = \quad shape \quad = \quad M\,!_a\,.$$

□

72 Factorisation. The latter example is the key to the factorisation of the sumtype functor. Let $\dagger$: $\mathcal{A} \times \mathcal{C} \to \mathcal{C}$ be a bifunctor for which $\alpha, M = sumtype(\dagger)$ exists. Define the curry-ed $\dagger$, denoted $\dagger^{\mathsf{C}}$: $\mathcal{A} \to \mathcal{F}'\mathcal{C}$, by

$$\begin{array}{lcll} \dagger^{\mathsf{C}} a & = & a\dagger & \\ \dagger^{\mathsf{C}} f & = & f\dagger \;:\; a\dagger \dot{\to} b\dagger & \qquad \text{where } (f\dagger)_c = f \dagger id_c = (f \dagger id)_c \end{array}$$

for any f: $a \to_{\mathcal{A}} b$. That $\dagger^{\mathsf{C}} f$ is a natural transformation is easily verified; it follows from laws ntrf-CONST, -ID, -BIFTR in paragraph A.13. Now

$$\begin{array}{lclcl} (Mu \circ \dagger^{\mathsf{C}})\,a & = & Mu(a\dagger) & = & M\,a \\ (Mu \circ \dagger^{\mathsf{C}})\,f & = & Mu(f\dagger) & = & (\![f\dagger \mathbin{;} \alpha_b]\!)_{a\dagger} \quad = \quad Mf\,. \end{array}$$

So $M = Mu \circ \dagger^{\mathsf{C}}$.

3d Many-sortedness and other variations

73 Many-sorted algebra. The notion of (di)algebra is rich enough to model many-sorted (di)algebras. As an example consider the collection

$$\langle \mathit{bool}, \mathit{nat};\ \mathit{true}, \mathit{false}, \mathit{bool\text{-}to\text{-}nat}, \mathit{zero}, \mathit{succ}, \mathit{equal} \rangle\,.$$

This collection is or suggests a two-sorted algebra, the two sorts (types) being *bool* and *nat*. In view of the typing *bool-to-nat*: $\mathit{bool} \to \mathit{nat}$ and *equal*: $I\!I\,\mathit{nat} \to \mathit{bool}$ both sorts are needed simultaneously to specify the operations.

We shall show that by instantiating the underlying category to a product category, category $\mathcal{A}lg(F)$ consists of many-sorted algebras indeed. Besides that, a single initial many-sorted algebra can be expressed as many initial single-sorted algebras. Thus the existence conditions and the construction for initial algebras over a product category are reduced to initial algebras over the component categories. These two results say that the theory for just 'normal' algebras also applies to many-sorted algebras.

We formalise only the case "many = two". It has the advantage that the formulas are simpler than in the general case, whereas all essential aspects are covered. You can easily generalise the discussion to "many = n" for arbitrary natural n.

74 Formalisation. The example above motivates the following definition. A **two-sorted †, ‡-algebra** is: a pair (φ, ψ) with

$$\varphi\colon a \dagger b \to a \qquad \text{and} \qquad \psi\colon a \ddagger b \to b$$

for some a, b called the sorts of the two-sorted algebra. Let (φ, ψ) and (χ, ω) be two-sorted †, ‡-algebras. A **two-sorted †, ‡-homomorphism** from (φ, ψ) to (χ, ω) is: a pair (f, g) with

$$\varphi \,;\, f = f \dagger g \,;\, \chi \qquad \text{and} \qquad \psi \,;\, g = f \ddagger g \,;\, \omega$$

so that

$$f\colon \mathrm{tgt}\varphi \to \mathrm{tgt}\chi \qquad \text{and} \qquad g\colon \mathrm{tgt}\psi \to \mathrm{tgt}\omega\,.$$

For this to make sense † and ‡ should be bifunctors with common source $\mathcal{A} \times \mathcal{B}$ and targets $\mathcal{A}$ and $\mathcal{B}$ respectively, for some categories $\mathcal{A}$ and $\mathcal{B}$. Clearly, the two-sorted algebras with their homomorphisms constitute a category, 2-$\mathcal{A}lg(\dagger, \ddagger)$ say.

There is, however, a simpler definition for two-sorted algebras. Recall the notion of product category: its objects and homomorphisms are pairs, and composition etc, are defined coordinatewise. Taking † and ‡ as above, the composite $\dagger \vartriangle \ddagger$ is an endofunctor on $\mathcal{A} \times \mathcal{B}$. Spelling out what it means to be an object or morphism in $\mathcal{A}lg(\dagger \vartriangle \ddagger)$ (having underlying category $\mathcal{A} \times \mathcal{B}$), we'll see that these are exactly the two-sorted †, ‡-algebras and homomorphisms defined above:

$$(\varphi, \psi) \text{ in } \mathcal{A}lg(\dagger \vartriangle \ddagger)$$
$$\equiv$$

$$\begin{array}{l} \quad (\varphi,\psi)\colon\ (\dagger \vartriangle \ddagger)(a,b) \rightarrow_{\mathcal{A}\times\mathcal{B}} (a,b) \\ \equiv \\ \quad \varphi\colon\ a \dagger b \rightarrow_{\mathcal{A}} a \quad \text{and} \quad \psi\colon\ a \ddagger b \rightarrow_{\mathcal{B}} b \end{array}$$

and further

$$\begin{array}{l} \quad (f,g)\colon\ (\varphi,\psi) \rightarrow_{\mathcal{A}lg(\dagger\vartriangle\ddagger)} (\chi,\omega) \\ \equiv \\ \quad (\varphi,\psi) ; (f,g) = (\dagger \vartriangle \ddagger)(f,g) ; (\chi,\omega) \\ \equiv \\ \quad \varphi ; f = f \dagger g ; \chi \quad \text{and} \quad \psi ; g = f \ddagger g ; \omega. \end{array}$$

Thus,

$$2\text{-}\mathcal{A}lg(\dagger,\ddagger) \quad = \quad \mathcal{A}lg(\dagger \vartriangle \ddagger),$$

and the prefix "two-sorted $\dagger,\ddagger$-" is just the same as "$\dagger \vartriangle \ddagger$-".

The case $n = 0$. The case $n = 0$ degenerates to the trivial algebra for the trivial nullary functor into the trivial category:

$$\begin{array}{lcl} \mathbb{1} & = & \text{the 0-fold product category} \\ & & \text{has } \imath \text{ as only object, and } id_\imath \text{ as only morphism} \\ \underline{\imath} & = & \text{the only 0-ary functor from } \mathbb{1} \text{ to } \mathbb{1} \\ id_\imath & : & \underline{\imath}(\imath) \rightarrow_{\mathbb{1}} \imath\,. \end{array}$$

75 Two-sortedness and initiality. The following theorem has already been observed by Wraith [77]. He gives the proof in classical categorical language and style, with many occurrences of the phrase "this gives a unique arrow." I could understand his proof only after constructing the proof below, where the uniqueness is made explicit by using formulation cata-CHARN.

76 Theorem. *Let $\dagger,\ddagger$ be bifunctors. Suppose that $\mu(\dagger b)$ and $\mu(a\ddagger)$ exist for all a,b. Then there exist endofunctors F,G such that*

$(\mu F, \mu G)$ is initial in $\mathcal{A}lg(\dagger \vartriangle \ddagger)$.

Specifically, $F = I \dagger M$ and $G = (U\mu F)\ddagger$, where $\text{-}, M = sumtype(\ddagger)$.

For this to make sense it is required that there are categories $\mathcal{A},\mathcal{B}$, and that functors $\dagger,\ddagger$ have source $\mathcal{A}\times\mathcal{B}$ and target $\mathcal{A}$ and $\mathcal{B}$ respectively, and that F,G are endofunctors on $\mathcal{A}\times\mathcal{B}$. Then it follows that $M\colon \mathcal{A} \rightarrow \mathcal{B}$.

Proof. For whatever F and G are going to be, put $\alpha,\beta = \mu F, \mu G$ and $a,b = U\alpha, U\beta$. We shall synthesise a pair F,G and an expression for $(\!|(\alpha,\beta) \rightarrow \_|\!)_{\dagger\vartriangle\ddagger}$ such that assertion

$$(x,y) \ :\ (\alpha,\beta) \ \rightarrow_{\dagger\vartriangle\ddagger} \ (\varphi,\psi) \qquad \equiv \qquad (x,y) = (\!|(\alpha,\beta) \rightarrow (\varphi,\psi)|\!)_{\dagger\vartriangle\ddagger}$$

is valid, thus establishing initiality of (α,β) in $\mathcal{A}lg(\dagger \vartriangle \ddagger)$.

Let (φ, ψ) be an arbitrary $\dagger \vartriangle \ddagger$-algebra, say

$$\begin{array}{lll} \varphi & : & c \dagger d \to c \\ \psi & : & c \ddagger d \to d \end{array}$$

for some c, d. It follows that $x\colon a \to c$ and $y\colon b \to d$ (for the x, y in the desired equivalence). Now we argue

$$\begin{array}{ll}
 & (x,y)\colon (\alpha,\beta) \to_{\dagger\vartriangle\ddagger} (\varphi,\psi) \\
\equiv & \quad \text{product category} \\
 & \alpha \mathbin{;} x = x \dagger y \mathbin{;} \varphi \ \wedge\ \beta \mathbin{;} y = x \ddagger y \mathbin{;} \psi \\
\equiv & \quad \text{bifunctor (aim: express } y \text{ as a homomorphism from } \beta) \\
 & \alpha \mathbin{;} x = x \dagger y \mathbin{;} \varphi \ \wedge\ \beta \mathbin{;} y = id_a \ddagger y \mathbin{;} x \ddagger id_d \mathbin{;} \psi \\
\equiv & \quad \textbf{define } G := a\ddagger = (U\mu F)\ddagger\text{, cata-}\textsc{Charn} \\
 & \alpha \mathbin{;} x = x \dagger y \mathbin{;} \varphi \ \wedge\ y = (\!|x \ddagger id_d \mathbin{;} \psi|\!)_G \\
\equiv & \quad \text{type-cata-}\textsc{Fusion}, \\
 & \quad \textbf{abbreviate } f = (\!|\psi|\!)_{c\ddagger} \text{ and } \textbf{define } \text{-}, M = \mathit{sumtype}(\ddagger) \\
 & \alpha \mathbin{;} x = x \dagger y \mathbin{;} \varphi \ \wedge\ y = Mx \mathbin{;} f \\
\equiv & \quad \text{substitute } y = Mx \mathbin{;} f \text{ in left conjunct, functor} \\
 & \alpha \mathbin{;} x = (I \dagger M)x \mathbin{;} id_c \dagger f \mathbin{;} \varphi \ \wedge\ y = Mx \mathbin{;} f \\
\equiv & \quad \textbf{define } F = I \dagger M\text{, cata-}\textsc{Charn} \\
 & x = (\!|id_c \dagger f \mathbin{;} \varphi|\!)_F \ \wedge\ y = Mx \mathbin{;} f \\
\equiv & \quad \text{product category} \\
 & (x,y) = ((\!|id_c \dagger f \mathbin{;} \varphi|\!)_F,\ \ Mx \mathbin{;} f).
\end{array}$$

Thus we have found the required definitions of F, G and $(\!|\_|\!)_{\dagger\vartriangle\ddagger}$. □

Bialgebras

77 Bialgebras. A datatype like *stack* with operations *empty*, *push*, *isempty*, *top* and *pop* has not the form of an algebra $\varphi\colon Fa \to a$, but is rather a pair (φ, ψ) with $\varphi\colon Fa \to a$ and $\psi\colon a \to Ga$, for some endofunctors F, G. For *stack* we have

$$\begin{array}{llllllll} \varphi & = & \mathit{empty} \triangledown \mathit{push} & : & 1 + b \times a \to a & = & Fa \to a \\ \psi & = & \mathit{isempty} \vartriangle \mathit{top} \vartriangle \mathit{pop} & : & a \to \mathit{bool} \times b \times a & = & a \to Ga\,, \end{array}$$

where b is the type of the stacked values (and a is the type of the stacks themselves). We call such a pair (φ, ψ) a (single-sorted) F, G**-bialgebra**. For *stack* there are some laws that relate the operations to each other; this aspect, not relevant here, is discussed in Chapter 5. Also, bialgebra *stack* is special among the bialgebras of the same type in that it is initial in some sense. Also this aspect is irrelevant for the formalisation of bialgebra proper. An F,G-**bialgebra homomorphism** from (φ,ψ) to (χ,ω) is a morphism f satisfying

$$\varphi \mathbin{;} f = Ff \mathbin{;} \chi \qquad \text{and} \qquad \psi \mathbin{;} Gf = f \mathbin{;} \omega\,.$$

Clearly, the F,G-bialgebras and homomorphisms form a category, called $\mathcal{B}iAlg(F,G)$, that is built upon the default category.

78 Bialgebras are dialgebras. Similarly to many-sorted algebras, a bialgebra is a particular dialgebra. Let F,G be endofunctors (on the default category), and consider $\mathcal{D}iAlg(F \vartriangle I, I \vartriangle G)$. Then

$$\begin{array}{ll} & (\varphi,\psi) \text{ in } \mathcal{D}iAlg(F \vartriangle I, I \vartriangle G) \\ \equiv & \\ & (\varphi,\psi)\colon (F \vartriangle I)a \to_{\mathcal{C}\times\mathcal{C}} (I \vartriangle G)a \quad \text{for some } a \\ \equiv & \\ & \varphi\colon Fa \to a \quad \text{and} \quad \psi\colon a \to Ga \quad \text{for some } a \end{array}$$

and moreover

$$\begin{array}{ll} & f\colon (\varphi,\psi) \to_{\mathcal{D}iAlg(F\vartriangle I, I\vartriangle G)} (\chi,\omega) \\ \equiv & \\ & (\varphi,\psi) \mathbin{;} (I \vartriangle G)f = (F \vartriangle I)f \mathbin{;} (\chi,\omega) \\ \equiv & \\ & \varphi \mathbin{;} f = Ff \mathbin{;} \chi \quad \text{and} \quad \psi \mathbin{;} Gf = f \mathbin{;} \omega. \end{array}$$

Hence

$$\mathcal{B}iAlg(F,G) \quad = \quad \mathcal{D}iAlg(F \vartriangle I, I \vartriangle G)\,.$$

(Notice, by the way, that f above is a morphism in the source of $F \vartriangle I$ and $I \vartriangle G$, which is the default category. So in contrast to the case for two-sorted algebras, f is not a pair.)

79 Two-sorted bialgebras. The generalisation to two-sorted bialgebras is straightforward. We won't use these algebras anywhere in the sequel, and give the discussion only to show that the notion of dialgebra is quite general indeed. Let us say that a two-sorted bialgebra is a tuple $(\varphi,\psi,\varphi',\psi')$ with typing

$$\begin{array}{ll} \varphi\colon a \dagger b \to a & \psi\colon a \to a \ddagger b \\ \varphi'\colon a \dagger' b \to b & \psi'\colon b \to a \ddagger' b\,. \end{array}$$

These two-sorted bialgebras form the objects of category $\mathcal{D}iAlg((\dagger \vartriangle \dagger') \vartriangle I,\ I \vartriangle (\ddagger \triangledown \ddagger'))$, as is easily verified by just unfolding the definitions. A morphism in this category is probably just what you might wish as a morphism for two-sorted bialgebra. The most general typing of the bifunctors reads

$$\begin{array}{l} \dagger, \dagger'\colon \mathcal{A} \times \mathcal{B} \to \mathcal{A} \\ \ddagger, \ddagger'\colon \mathcal{A} \times \mathcal{B} \to \mathcal{B}\,. \end{array}$$

The morphisms of the category are just what you might have expected.

3e Conclusion

Distributivity properties play an important rôle in transformational programming. The categorical formalisation of distributivity leads to the notion of homomorphism, with a collection of operations being a dialgebra. There are a lot of laws for dialgebras, and these facilitate to calculate with algorithms in an algebraic way (in the sense of high school algebra). In order to make clear the pattern or structure of an algorithm, and thus to discover the homomorphisms involved, it is very helpful if algorithms are expressed as compositions of functions, rather than as cascaded applications of functions to arguments. A possible drawback is the presence of a lot of combinators that are algorithmically not interesting, and whose only purpose is to get the arguments in the right place. As soon as one deals with actual algorithms, rather than with general schemes with a few φ's and ψ's, the amount of argument shuffling combinators may become irritatingly large; an example of this occurs in the next chapter: transpose in Section 4c.

Initial algebras and final co-algebras turn out to be a formalisation of the intuitive notion of datatype. The initiality (or finality) of the (co)algebras gives further laws that facilitate to calculate with functions defined with induction on the structure of the source algebra (or target co-algebra). The FUSION laws are quite important for efficiency improvement since they exploit the distributivity property of one of the functions involved. (Interestingly, in a more general context and without efficiency considerations in mind, the FUSION law has turned out to be an important law for calculation with functions, in Chapter 2.)

The categorical technique of dualisation is not just a formal game, but gives results that are relevant for practice, as demonstrated by the many examples that we have given.

Though most of the theorems of this chapter may be known, or even well-known, it is certainly not the case that the algebraic style of calculating with algorithms is common coin.

Chapter 4

Unique Fixed Points

Initiality of an algebra asserts that a certain type of equation has precisely one solution, which may be called the morphism 'defined by' the equation. There are more types of equations that have precisely one solution and can therefore be characterised by laws like CHARN. In some cases such laws have consequences similar to the FUSION law that we know for cata- and anamorphisms (and that is so useful for program transformation). One type of equation gives an alternative view on the datatype; another type of equation gives mutumorphisms (mutually recursive definitions); a third type has solutions that we call prepromorphisms and dually postpromorphisms.

As an aside, we derive sufficient conditions for the equality of an cata- and anamorphism, and illustrate these by expressing a transpose function both as a cata- and as an anamorphism.

1 Introduction. Let F be an endofunctor and α be an initial F-algebra. Initiality of α means that the equation $\alpha \,;\, x = Fx \,;\, \varphi$, or equivalently $x = \alpha^{\cup} \,;\, Fx \,;\, \varphi$, has precisely one solution, for arbitrary F-algebra φ. The solution is denoted $(\!|\alpha \rightarrow \varphi|\!)_F$ or briefly $(\!|\varphi|\!)$. The laws cata-CHARN,... , cata-FUSION facilitate reasoning about the function thus defined. In practice one encounters equations that do not fit the pattern above, yet have precisely one solution. The uniqueness means that a characterisation like CHARN is possible, hence also laws like SELF and UNIQ, and maybe also ID and FUSION. These laws are useful for program transformations.

For example, Meertens [49] discusses equations of the form

$$\alpha \,;\, f \quad = \quad F(id \vartriangle f) \,;\, \varphi .$$

For arbitrary φ of the right type this equation has a unique solution f, called the F-**paramorphism** for φ. Interpreted in *Set*, operation φ gets not only the results of the recursive (inductive) invocations of f, but also the arguments that were passed to those recursive invocations. As Meertens proves, and we will do so in Section 4b, paramorphisms satisfy properties similar to those of catamorphisms; in particular a FUSION law. Moreover, Meertens proves that each morphism with the carrier of α as source is a paramorphism.

2 Overview. We investigate three different kinds of equations. First, in Section 4a we consider the definition of morphisms on $U\alpha$ via an inductive pattern that does not conform to α and F. Then in Section 4b we consider mutually recursive definitions, or rather simultaneous inductive equations. The morphisms so defined generalise Meertens' paramorphisms. Third, in Section 4d we consider schemes that differ from the catamorphism equation in that the recursive invocations are preceded by some 'preprocessing'. In the dual case the recursive invocations in the equation for anamorphisms are succeeded by some postprocessing. As an aside, we give in Section 4c two conditions under which a catamorphism equals an anamorphism. The law is illustrated by the equivalence proof for a catamorphism and an anamorphism expression for a kind of array transpose.

4a Views on datatypes

In practice one often views cons lists as snoc or join lists, and vice versa. In particular one uses both induction on the cons structure and induction on the join or snoc structure to define functions on lists. We set out to describe this phenomenon formally, for datatypes in general. For concreteness, however, we refer to lists. A snoc list is like a cons list; the difference is that the 'snoc' operation appends an element to the other side of the list, as suggested by its type: *snoc*: $Ma \times a \to Ma$. More precisely, the intended correspondence between cons and snoc lists is suggested by

$$cons(a, \ldots cons(z, nil))) \quad \approx \quad [a, \ldots, z] \quad \approx \quad snoc(snoc(lin, a) \ldots, z).$$

Here *lin*: $\underline{1} \to Ma$ denotes the empty snoc list.

3 Cons as snoc list. Let $F = \underline{1} + \underline{a} \times I$ be the usual functor for cons lists over a, and let $\alpha = nil \triangledown cons$ be the initial F-algebra μF, with carrier La. Suppose you want to define functions on cons lists by induction on the snoc pattern; think of left reduces. Then you have to define your own snoc view of cons lists, for example by defining

$$\beta' \quad = \quad nil \triangledown (swap \,;\, id \times rev \,;\, cons \,;\, rev) \quad : \quad GLa \to La\,,$$

where G is the functor for snoc lists, $G = \underline{1} + I \times \underline{a}$, and *rev*: $La \to La$ is the reverse operation. (A prime-less 'β' is defined later.) We do not elaborate *rev* here, but assume that *rev* is its own inverse; compare with *reverse* discussed in paragraph 3.70. Having done so, you will expect that for each G-algebra φ the equation

$$\beta' \,;\, f \quad = \quad Gf \,;\, \varphi$$

has precisely one solution f, so that these equations do define functions on La. The existence and uniqueness of a solution of the equation, for each φ, means that β' is an initial G-algebra. To prove the initiality of β', there are at least two ways.

The first method is to show directly that cata-CHARN holds for β', that is, there is an expression E not containing f such that the above equation equivales $f = E$. The E, then, is the catamorphism that may henceforth be written $(\!|\beta' \rightarrow \varphi|\!)_G$.

The second method is as follows. Let β be 'the' initial G-algebra μG, say $\beta = lin \triangledown snoc$ with carrier Ma. Then show that $\beta' \cong_G \beta$, that is, the unique G-homomorphism $(\!|\beta \rightarrow \beta'|\!)_G$ from β to β' has an inverse $g\colon \beta' \to_G \beta$. We shall now spend some words on both methods.

4 *The first method.* It so happens that

$$\begin{array}{lcl} \beta' & = & nil \triangledown (swap \,;\, id \times rev \,;\, cons \,;\, rev) \\ & = & id + swap \,;\, id + id \times rev \,;\, nil \triangledown cons \,;\, rev \\ & = & \varepsilon \,;\, Frev \,;\, \alpha \,;\, rev \end{array}$$

where

$$\varepsilon \quad = \quad id + swap \;:\; G \cong F \qquad \text{a natural isomorphism.}$$

To show the existence and uniqueness of f in the equation $\beta' ; f = Gf ; \varphi$ we argue

$$\begin{array}{ll} & \beta' ; f = Gf ; \varphi \\ \equiv & \quad \text{equation for } \beta' \\ & \varepsilon ; Frev ; \alpha ; rev ; f = Gf ; \varphi \\ \equiv & \quad \text{exixtence } \varepsilon^{\cup}, rev^{\cup} \; (= \varepsilon, rev) \\ & \alpha ; rev ; f = Frev^{\cup} ; \varepsilon^{\cup} ; Gf ; \varphi \\ \equiv & \quad \text{naturality } \varepsilon^{\cup}\colon F \dot{\to} G \\ & \alpha ; (rev ; f) = F(rev^{\cup} ; f) ; (\varepsilon^{\cup} ; \varphi) \\ \equiv & \quad \text{cata-CHARN for } \alpha, \text{ using that } rev = rev^{\cup} \\ & rev^{\cup} ; f = (\!|\alpha \rightarrow \varepsilon^{\cup} ; \varphi|\!)_F \\ \equiv & \quad \text{inverse } rev \\ & f = rev ; (\!|\alpha \rightarrow \varepsilon^{\cup} ; \varphi|\!)_F. \end{array}$$

Hence,

$$(\!|\beta' \rightarrow \varphi|\!)_G \quad = \quad rev ; (\!|\alpha \rightarrow \varepsilon^{\cup} ; \varphi|\!)_F \quad : \quad La \to U\varphi .$$

Notice that the carrier of the initial G-algebra β' equals that of the initial F-algebra α.

5 *The second method.* For the particular case at hand it is rather easy to construct the inverse g of $(\!|\beta \rightarrow \beta'|\!)_G$. The requirement $g\colon \beta' \to_G \beta$ determines g completely:

$$\begin{array}{ll} & \beta' ; g = Gg ; \beta \\ \equiv & \quad \text{argued above} \end{array}$$

$$g = rev \mathbin{;} (\!|\alpha \dashrightarrow \varepsilon^{\cup} \mathbin{;} \beta|\!)_F.$$

Having derived a candidate g, it remains to show both that it is a pre-inverse of $(\!|\beta \dashrightarrow \beta'|\!)_G$ and that it is a post-inverse. Thus, this method of showing the initiality of β' is more involved than the previous one.

6 Isomorphic views. Except for the specific choices the discussion in paragraph 3 is completely general. So, suppose α is an initial F-algebra, and you want to "view" α as an initial G-algebra. Then you should define a G-algebra β' in terms of α having the same carrier as α, and prove that β' is an initial G-algebra. As a result, the equations $\beta' \mathbin{;} x = Gx \mathbin{;} \varphi$, for arbitrary G-algebra φ, have a unique solution x: $U\alpha \to U\varphi$.

We can also formally define a notion of isomorphism between an F-algebra and a G-algebra, or, more generally, between dialgebras of different type. To this end define the category $\mathcal{D}i\mathcal{A}lg$, built on the default category, as follows. An object in $\mathcal{D}i\mathcal{A}lg$ is: a triple (F, φ, G) where φ is an F, G-dialgebra. Let (F, φ, G) and (H, ψ, J) be objects in $\mathcal{D}i\mathcal{A}lg$; then a morphism from (F, φ, G) to (H, ψ, J) in $\mathcal{D}i\mathcal{A}lg$ is: a triple (ε, f, η) where ε: $F \dot{\to} H$, η: $G \dot{\to} J$, and f: $U\varphi \to U\psi$ is a morphism satisfying

$$(\varphi \mathbin{;} \eta \mathbin{;} Jf \;\; =) \;\; \varphi \mathbin{;} Gf \mathbin{;} \eta \quad = \quad \varepsilon \mathbin{;} Hf \mathbin{;} \psi \;\; (= \quad Ff \mathbin{;} \varepsilon \mathbin{;} \psi).$$

So defined $\mathcal{D}i\mathcal{A}lg$ is a category indeed.

For the case considered above, α is an F, I-dialgebra, and $\beta' = \varepsilon \mathbin{;} Fh \mathbin{;} \alpha \mathbin{;} h$ is a G, I-dialgebra, where h is such that $h^{\cup}$ exists, and ε: $G \cong F$. Then α and β', or rather (F, α, I) and (G, β', I), are isomorphic in $\mathcal{D}i\mathcal{A}lg$, since

$$(\varepsilon^{\cup}, h, id) \;:\; (F, \alpha, I) \;\cong\; (G, \beta', I) \text{ in } \mathcal{D}i\mathcal{A}lg.$$

4b Mutumorphisms

7 Mutual recursion. The use of auxiliary functions is commonplace in programming. Often a function or algorithm f is easily expressed by induction on the structure of the argument, provided that some function g may be used; where g is expressed by induction too, using f in its turn. We call such functions **mutumorphisms** (*mutu* arising from mutually recursive). The discussion below formalises the folklore intuition that such mutumorphisms can be expressed in terms of a single recursive function. In addition it follows that mutumorphisms have nice calculational properties, including a FUSION law. Specific cases arise when one or both do not really depend on the other.

8 Theorem. *Let F be an endofunctor, and α be μF. Then*

$$\alpha \mathbin{;} f = F(f \vartriangle g) \mathbin{;} \varphi \;\wedge\; \alpha \mathbin{;} g = F(f \vartriangle g) \mathbin{;} \psi \quad \equiv \quad f \vartriangle g = (\!|\varphi \vartriangle \psi|\!) \qquad \text{MUTU}$$

For this to make sense it is required that φ: $F(a \times b) \to a$, ψ: $F(a \times b) \to b$, f: $U\alpha \to a$, and g: $U\alpha \to b$ for some a, b.

Notice that (φ, ψ) is a two-sorted $F \circ (\times)$-algebra, and so is (α, α). But it is type-incorrect to claim that (f, g) is an $F \circ (\times)$-homomorphism. The proof of the theorem is simple.

$$\begin{array}{ll} & \alpha \,;\, f = F(f \vartriangle g) \,;\, \varphi \quad \wedge \quad \alpha \,;\, g = F(f \vartriangle g) \,;\, \psi \\ = & \quad \text{product} \\ & (\alpha \,;\, f) \vartriangle (\alpha \,;\, g) \;=\; (F(f \vartriangle g) \,;\, \varphi) \vartriangle (F(f \vartriangle g) \,;\, \psi) \\ = & \quad \text{product} \\ & \alpha \,;\, f \vartriangle g \;=\; F(f \vartriangle g) \,;\, \varphi \vartriangle \psi \\ = & \quad \text{cata-CHARN} \\ & f \vartriangle g = (\!|\varphi \vartriangle \psi|\!). \end{array}$$

9 Mutu laws. Since the theorem asserts that the tupling of mutumorphisms is a catamorphism, there is a characterisation of mutumorphisms (the theorem!), and hence also several derived laws. A specific notation may make it more clear. For F and φ, ψ as in the theorem, the left, right and tupled **mutumorphism** is:

$$\begin{array}{lcl} [\![\varphi, \psi]\!]_0 & = & (\!|\varphi \vartriangle \psi|\!) \,;\, \mathit{exl} \\ [\![\varphi, \psi]\!]_1 & = & (\!|\varphi \vartriangle \psi|\!) \,;\, \mathit{exr} \\ [\![\varphi, \psi]\!] & = & [\![\varphi, \psi]\!]_0 \vartriangle [\![\varphi, \psi]\!]_1 \quad = \quad (\!|\varphi \vartriangle \psi|\!)\,. \end{array}$$

Then, putting $\varphi_{0,1} = \varphi_0, \varphi_1$, the following laws are immediate corollaries of the theorem and the laws for catamorphisms.

$\alpha \,;\, x_i = F(x_0 \vartriangle x_1) \,;\, \varphi_i \quad {\scriptstyle (i=0,1)}$	$\equiv$	$x_i = [\![\varphi_{0,1}]\!]_i \quad {\scriptstyle (i=0,1)}$	mutu-CHARN
$\alpha \,;\, [\![\varphi_{0,1}]\!]_i = F[\![\varphi_{0,1}]\!] \,;\, \varphi_i \quad {\scriptstyle (i=0,1)}$			mutu-SELF
$\mathit{id} = [\![F\mathit{exl} \,;\, \alpha,\; F\mathit{exr} \,;\, \alpha]\!]_i \quad {\scriptstyle (i=0,1)}$			mutu-ID
$\left.\begin{array}{l}\alpha \,;\, x_i = F(x_0 \vartriangle x_1) \,;\, \varphi_i \quad {\scriptstyle (i=0,1)} \\ \alpha \,;\, y_i = F(y_0 \vartriangle y_1) \,;\, \psi_i \quad {\scriptstyle (i=0,1)}\end{array}\right\}$	$\Rightarrow$	$x_i = y_i \quad {\scriptstyle (i=0,1)}$	mutu-UNIQ
$\varphi_0 \vartriangle \varphi_1 \,;\, f = Ff \,;\, \psi_0 \vartriangle \psi_1$	$\Rightarrow$	$[\![\varphi_{0,1}]\!] \,;\, f = [\![\psi_{0,1}]\!]$	mutu-FUSION

10 Specialisations. Substituting $\psi = F\mathit{exr} \,;\, \chi$ for some F-algebra χ in Theorem 8 gives the special case that f may still depend on g, and g does not depend on f and, in fact, g is a catamorphism itself:

$$\begin{array}{ll} & \alpha \,;\, g = F(f \vartriangle g) \,;\, F\mathit{exr} \,;\, \chi \\ \equiv & \\ & \alpha \,;\, g = Fg \,;\, \chi \\ \equiv & \\ & g = (\!|\chi|\!). \end{array}$$

Malcolm [42] investigates such f, g and calls f the F**-zygomorphism** for φ, χ. Taking in particular $\chi = \alpha$ gives $g = (\!|\alpha|\!) = \mathit{id}$, and the resulting f is called the F**-paramorphism** for φ by Meertens [49].

$[\![\varphi,\; F\mathit{exr} \,;\, \chi]\!]_0$ is Malcolm's zygomorphism $(\varphi, \chi)^\natural$

$[\![\varphi,\ F\,exr\ ;\ \alpha]\!]_0$ is Meertens' paramorphism $[\![\varphi]\!]$

(In paragraph 3.51 we have discussed the paramorphism *preds*.) In case f does not depend on g either, by substituting $\varphi := F\,exl\ ;\ \varphi$ for some F-algebra φ, the theorem asserts that the tupling of catamorphisms is a catamorphism itself:

$$(\!|\varphi|\!) \vartriangle (\!|\psi|\!) \;=\; (\!|(F\,exl\ ;\ \varphi) \vartriangle (F\,exr\ ;\ \psi)|\!) \;=\; (\!|abide_F\ ;\ \varphi \times \psi|\!) \qquad \text{BANANA SPLIT}$$

where $abide_F = F\,exl \vartriangle F\,exr$. (We call this operation $abide_F$ since for $F = I\!I$ it is the natural transformation that occurs in the equation expressing the *abide* property. Jaap van der Woude coined the name *banana* brackets for $(\!|\ |\!)$ and *split* for $\vartriangle$. Lambert Meertens put them together in juxtaposition to name this law.)

It is quite remarkable that these results are so easily proved in a categorical setting, whereas in the relational setting —where $\vartriangle$ is not the categorical product— the proofs are quite complicated, and Theorem 8 is even false; cf. Voermans and Van der Woude [73].

4c Equal cata- and anamorphisms

There are several conditions under which a catamorphism can be expressed as an anamorphism, and vice versa. We present two such conditions, and illustrate one of them by proving the equality of two ways to express a certain kind of array transpose.

11 Unique fixed points. Both an anamorphism and a catamorphism is the solution of a certain kind of fixed point equation. In order to study both kinds of equations at the same time, we abstract from the particular form, and consider morphism mappings $\mathcal{F}$ for which the equation $x = \mathcal{F}x$ has a unique solution. We use the notation $[\![\mathcal{F}]\!]$ for the unique solution of $x = \mathcal{F}x$, assuming that it exists. So $[\![\mathcal{F}]\!]$ is characterised by ufp-CHARN below, and hence satisfies the two other laws as well.

$$x = \mathcal{F}x \quad\equiv\quad x = [\![\mathcal{F}]\!] \qquad \text{ufp-CHARN}$$
$$[\![\mathcal{F}]\!] = \mathcal{F}[\![\mathcal{F}]\!] \qquad \text{ufp-SELF}$$
$$x = \mathcal{F}x \;\wedge\; y = \mathcal{F}y \;\Rightarrow\; x = y \qquad \text{ufp-UNIQ}$$

As an example, let F be an endofunctor, α be μF, and φ be an F-algebra. Take $\mathcal{F}x = \alpha^{\cup}\ ;\ Fx\ ;\ \varphi$. Then, by ufp- and cata-CHARN, $[\![\mathcal{F}]\!]$ exists and equals $(\!|\varphi|\!)_F$. Similarly, if G is an endofunctor, $\beta = \nu G$, and ψ is a G-co-algebra, then taking $\mathcal{G}x = \psi\ ;\ Gx\ ;\ \beta^{\cup}$ yields $[\![\mathcal{G}]\!] = [\!(\psi)\!]_G$.

We shall now investigate conditions for $[\![\mathcal{F}]\!] = [\![\mathcal{G}]\!]$, without prescribing the form of $\mathcal{F}$ or $\mathcal{G}$. So by suitable instantiations of $\mathcal{F}$ and $\mathcal{G}$ there result conditions not only for the equality of a cata- and an anamorphism, but also for two catamorphisms and for two anamorphisms.

12 Theorem. *Suppose* $[\![\mathcal{F}]\!]$ *and* $[\![\mathcal{G}]\!]$ *exist. Then* $\mathcal{F} = \mathcal{G} \;\Rightarrow\; [\![\mathcal{F}]\!] = [\![\mathcal{G}]\!]$.

The theorem is just a triviality. However, it has an interesting corollary:

$$\varepsilon\colon F \dot{\to} G \quad\Rightarrow\quad (\![\varepsilon \,;\, \beta^{\cup}]\!)_F = [\![\alpha^{\cup} \,;\, \varepsilon]\!]_G \qquad \text{cata-ana-EQ}$$

Taking $\mathcal{F}x = \alpha^{\cup} \,;\, Fx \,;\, \varepsilon \,;\, \beta^{\cup}$ and $\mathcal{G}x = \alpha^{\cup} \,;\, \varepsilon \,;\, Gx \,;\, \beta^{\cup}$, law cata-ana-EQ follows from the trivial theorem:

$$
\begin{array}{ll}
 & (\![\varepsilon \,;\, \beta^{\cup}]\!)_F = [\![\,[\![^{\cup}]\!] \,;\, \varepsilon]\!]_G \\
\equiv & \qquad \text{definition } \mathcal{F} \text{ and } \mathcal{G} \\
 & \llbracket \mathcal{F} \rrbracket = \llbracket \mathcal{G} \rrbracket \\
\Leftarrow & \qquad \text{theorem (!)} \\
 & \mathcal{F} = \mathcal{G} \\
\equiv & \qquad \text{extensionality} \\
 & \alpha^{\cup} \,;\, Ff \,;\, \varepsilon \,;\, \beta^{\cup} = \alpha^{\cup} \,;\, \varepsilon \,;\, Gf \,;\, \beta^{\cup} \qquad \text{for all } f \\
\Leftarrow & \qquad \text{Leibniz} \\
 & \varepsilon\colon F \dot{\to} G.
\end{array}
$$

Let us consider a less trivial theorem.

13 Theorem. *Suppose* $\llbracket \mathcal{F} \rrbracket$ *exists, and* $\mathcal{G}$ *has at most one fixed point. Then*

$$\mathcal{F}\,\mathcal{G} = \mathcal{G}\,\mathcal{F} \quad\Rightarrow\quad \llbracket \mathcal{G} \rrbracket \text{ exists, and equals } \llbracket \mathcal{F} \rrbracket .$$

Proof. We show that $\llbracket \mathcal{F} \rrbracket$ is a fixed point of $\mathcal{G}$.

$$
\begin{array}{ll}
 & \llbracket \mathcal{F} \rrbracket = \mathcal{G}\llbracket \mathcal{F} \rrbracket \\
\equiv & \qquad \text{ufp-CHARN}[\mathcal{F}, x := \mathcal{F}, \mathcal{G}\llbracket \mathcal{F} \rrbracket] \\
 & \mathcal{G}\llbracket \mathcal{F} \rrbracket = \mathcal{F}\,\mathcal{G}\llbracket \mathcal{F} \rrbracket \\
\equiv & \qquad \text{premise} \\
 & \mathcal{G}\llbracket \mathcal{F} \rrbracket = \mathcal{G}\,\mathcal{F}\llbracket \mathcal{F} \rrbracket \\
\Leftarrow & \qquad \text{Leibniz, ufp-SELF} \\
 & \textsf{true}.
\end{array}
$$

Since $\mathcal{G}$ has at most one fixed point, $\llbracket \mathcal{F} \rrbracket$ is the unique one. □

Taking $\mathcal{F}x = \alpha^{\cup} \,;\, Fx \,;\, \varphi$ and $\mathcal{G}x = \psi \,;\, Gx \,;\, \beta^{\cup}$ the theorem specialises to

14 $$\forall f :: \quad \alpha^{\cup} \,;\, F(\psi \,;\, Gf \,;\, \beta^{\cup}) \,;\, \varphi \quad=\quad \psi \,;\, G(\alpha^{\cup} \,;\, Ff \,;\, \varphi) \,;\, \beta^{\cup}$$
$$\Rightarrow \quad (\![\varphi]\!)_F = [\![\psi]\!]_G \qquad \text{cata-ana-EQ2}$$

I see no way to simplify the condition in a useful way. The instantiation $f = id$ in the premise gives the necessary condition

$$\alpha^{\cup} \,;\, F(\psi \,;\, \beta^{\cup}) \,;\, \varphi \quad=\quad \psi \,;\, G(\alpha^{\cup} \,;\, \varphi) \,;\, \beta^{\cup}.$$

Law cata-ana-EQ 12 is a specialisation of 14; this also follows from the fact that Theorem 12 is a special instance of Theorem 13. Law cata-ana-EQ2 is used in the application that now follows.

Application: transpose

15 Preliminaries. We shall define a function *transp* that maps a cons list of streams onto a stream of cons lists. So first of all recall the definitions and nomenclature for cons lists and streams.

$$\begin{array}{llllll} nil \triangledown cons & = & \alpha : & I \dagger L \dashrightarrow L & \text{where} & a\dagger = \underline{1} + \underline{a} \times I \\ hd \vartriangle tl & = & \beta : & S \dashrightarrow I \ddagger S & \text{where} & \ddagger = \times \, . \end{array}$$

Function $transp_a$ is to have type $LSa \to SLa$, and actually $transp\colon LS \dashrightarrow SL$.

I have taken $\ddagger = \times$ instead of $\ddagger = \dagger$ to simplify the formulas. Replacing $\dagger$ by $\times$ would also simplify matters, but too much: in $\mathcal{S}et$ the initial $a\times$-algebra is the trivial identity function on the empty set.

Second, we define two variations of *abide* and mention some laws for them. For arbitrary F and $\dagger$,

$$\begin{array}{lllll} abide_F & = & F\,exl \vartriangle F\,exr \\ abide_\dagger & = & (exl \dagger exl) \vartriangle (exr \dagger exr) & = & abide_{I\dagger I} \\ coabide_F & = & F\,inl \triangledown F\,inr \, . \end{array}$$

These three definitions imply, amongst others,

$$\begin{array}{lllr} Ff \vartriangle Fg & = & F(f \vartriangle g) \,;\, abide_F & \text{ABIDE} \\ (f \dagger g) \vartriangle (h \dagger j) & = & (f \vartriangle h) \dagger (g \vartriangle j) \,;\, abide_\dagger & \text{BIABIDE} \\ Ff \triangledown Fg & = & coabide_F \,;\, F(f \triangledown g) & \text{COABIDE} \end{array}$$

Third, recall the following from paragraphs 3.65 and 3.66:

$$\begin{array}{lll} nils & = & id^\omega \,;\, S\,nil \\ zip & = & [\![\mathit{II}(hd \vartriangle tl) \,;\, abide_{\mathit{II}}]\!] \\ zipwith\text{-}f & = & zip \,;\, Sf \, . \end{array}$$

16 Defining transpose. Looking at the source type of $transp_a\colon LSa \to SLa$ it is obvious to try and express transpose as a catamorphism. We start with an expression that formalises our intuition about transpose.

$$\begin{array}{ll} & nil \,;\, transp' = nils \\ & cons \,;\, transp' = id_{Sa} \times transp' \,;\, zipwith\text{-}cons \\ \equiv & \qquad \text{sum, definition } f \dagger g = id + f \times g \\ & nil \triangledown cons \,;\, transp' = id_{Sa} \dagger transp' \,;\, nils \triangledown zipwith\text{-}cons \\ \equiv & \qquad \text{definition } nil \triangledown cons = \alpha, \text{ cata-CHARN} \end{array}$$

$$\begin{array}{ll} & transp' = (\!|nils \triangledown zipwith\text{-}cons|\!)_{Sa\dagger} \\ \equiv & \qquad \text{definition } nils \text{ and } zipwith \\ & transp' = (\!|(id^\omega ; Snil) \triangledown (zip ; Scons)|\!)_{Sa\dagger} \\ \equiv & \qquad \text{sum} \\ & transp' = (\!|id^\omega + zip ; Snil \triangledown Scons|\!)_{Sa\dagger} \\ \equiv & \qquad \text{coABIDE, } nil \triangledown cons = \alpha \\ & transp' = (\!|id^\omega + zip ; coabide_S ; S\alpha|\!)_{Sa\dagger}. \end{array}$$

Adding the typing and using an auxiliary φ we have

$$\begin{array}{lcll} transp' & = & (\!|\varphi|\!)_{Sa\dagger} & : \ LSa \to SLa \\ \varphi & = & nils \triangledown zipwith\text{-}cons & : \ Sa \dagger SLa \to SLa \\ & = & id^\omega + zip ; coabide_S ; S\alpha . & \end{array}$$

The target type of $transp$, however, suggests to try and express transpose as an anamorphism. Again, the top line formalises our intuition about transpose; we shall later prove that the intuition here is consistent with the one expressed above.

$$\begin{array}{ll} & transp'' = Lhd \vartriangle (Ltl ; transp'') ; (hd \vartriangle tl)^\cup \\ \equiv & \qquad \text{invertibility } \beta = hd \vartriangle tl \text{, product} \\ & transp'' ; \beta = Lhd \vartriangle Ltl ; id_{La} \times transp'' \\ \equiv & \qquad \text{ana-CHARN interchanging left and right hand side, } \ddagger = \times \\ & transp'' = [\!(Lhd \vartriangle Ltl)\!]_{La\ddagger} \\ \equiv & \qquad \text{ABIDE, } hd \vartriangle tl = \beta \\ & transp'' = [\!(L\beta ; abide_L)\!]_{La\ddagger}. \end{array}$$

Adding the typing and using an auxiliary ψ we have

$$\begin{array}{lcll} transp'' & = & [\!(\psi)\!]_{La\ddagger} & : \ LSa \to SLa \\ \psi & = & L\,hd \vartriangle L\,tl & : \ LSa \to L \ddagger LSa \\ & = & L\beta ; abide_L . & \end{array}$$

17 Some typing. Here are some assertions concerning naturality of various object-parametrised morphisms. I shall not prove nor use any of these claims (say $\varepsilon\colon F \dot\to G$) as regards the equalities that they embody (namely $Ff ; \varepsilon_b = \varepsilon_a ; Gf$ for each $f\colon a \to b$), simply because I see no opportunity to use these. What I do use, implicitly, is the claims about the parametrised typing of the parametrised morphisms (namely $\varepsilon_a\colon Fa \to Ga$ for each a).

$transp$	:	$LS \dot\to SL$	
$nils$	:	$\underline{1} \dot\to SL$	
$zipwith\text{-}f$	:	$Sa \times Sb \to Sc$	for $f\colon a \times b \to c$
	:	$SF \times SG \dot\to SH$	for $f\colon F \times G \dot\to H$

$$\begin{array}{lll} zip & : & SX \times SY \dot{\to} S(X \times Y) \\ abide_{I\!I} & : & (W \times X) \times (Y \times Z) \dot{\to} (W \times Y) \times (X \times Z)\,, \end{array}$$

where for readability we have used the abbreviations $W, X, Y, Z = Ex_{4,0}, Ex_{4,1}, Ex_{4,2}, Ex_{4,3}$ in the typing of $abide_{I\!I}$, and $X, Y = Exl\,, Exr$ in the typing of zip.

18 Preparation. By way of preparation we transform the definitions of *transp′* and *transp″* somewhat. The aim, here, is to get an elegant form, fully expressed in α, β and $\dagger, \ddagger$, before we start the proof of the rather complicated premise of cata-ana-EQ2.

For ψ we recognise the opportunity to apply the BANANA SPLIT law.

$$\begin{array}{ll} & \psi \\ = & \quad \text{definition} \\ & Lhd \vartriangle Ltl \\ = & \quad \text{definition sumtype functor} \\ & (\!|hd \dagger id \mathbin{;} \alpha|\!) \vartriangle (\!|tl \dagger id \mathbin{;} \alpha|\!) \\ = & \quad \text{BANANA SPLIT} \\ & (\!|abide_{Sa\dagger} \mathbin{;} (hd \dagger id \mathbin{;} \alpha) \times (tl \dagger id \mathbin{;} \alpha)|\!) \\ = & \quad \text{definition } abide_F = F\,exl \vartriangle F\,exr\text{, functor} \\ & (\!|(id \dagger exl) \vartriangle (id \dagger exr) \mathbin{;} (hd \dagger id) \times (tl \dagger id) \mathbin{;} \alpha \times \alpha|\!) \\ = & \quad \text{product, definition } \ddagger = \times \\ & (\!|(hd \dagger exl) \vartriangle (tl \dagger exr) \mathbin{;} \alpha \ddagger \alpha|\!) \\ = & \quad \{\text{for elegance: aim at combining } hd \text{ and } tl \text{ into } \beta\,\} \\ & \quad \text{BIABIDE} \\ & (\!|(hd \vartriangle tl) \dagger (exl \vartriangle exr) \mathbin{;} abide_{\dagger} \mathbin{;} \alpha \ddagger \alpha|\!) \\ = & \quad \text{definition } \beta = hd \vartriangle tl\text{, product: } exl \vartriangle exr = id \\ & (\!|\beta \dagger id \mathbin{;} abide_{\dagger} \mathbin{;} \alpha \ddagger \alpha|\!). \end{array}$$

For φ we first consider *nils* and *zipwith-cons* separately.

$$\begin{array}{ll} & nils \\ = & \quad \text{definition } nils \\ & id^{\omega} \mathbin{;} S\,nil \\ = & \quad \text{definition iterate } \_^{\omega} \text{ in paragraphs 3.48–49} \\ & [\![split \mathbin{;} id \ddagger id]\!] \mathbin{;} S\,nil \\ = & \quad \text{ana-type-FUSION, product} \\ & [\![nil \vartriangle id]\!]. \end{array}$$

We abbreviate *zipwith-cons* to *zip′*, and remember from paragraph 3.65

$$zip' \quad = \quad [\![I\!I\beta \mathbin{;} abide_{I\!I} \mathbin{;} cons \times id]\!]\,.$$

Big surprise, for φ the dual of Banana Split is applicable! We calculate an elegant expression as follows.

φ

$=$ definition

$\mathit{nils} \triangledown \mathit{zip}'$

$=$ above expressions for nils and zip'

$\llbracket \mathit{nil} \vartriangle \mathit{id} \rrbracket \triangledown \llbracket I\!I\beta \mathbin{;} \mathit{abide}_{I\!I} \mathbin{;} \mathit{cons} \times \mathit{id} \rrbracket$

$=$ dual of Banana Split (Anaba junc?)

$\llbracket (\mathit{nil} \vartriangle \mathit{id}) + (I\!I\beta \mathbin{;} \mathit{abide}_{I\!I} \mathbin{;} \mathit{cons} \times \mathit{id}) \mathbin{;} \mathit{coabide}_{La\ddagger} \rrbracket$

$=$ { for elegance: aim at combining nil and cons into α }
definition $\mathit{coabide}_F = F\mathit{inl} \triangledown F\mathit{inr}$, sum

$\llbracket (\mathit{nil} \vartriangle \mathit{id} \mathbin{;} \mathit{id} \ddagger \mathit{inl}) \triangledown (I\!I\beta \mathbin{;} \mathit{abide}_{I\!I} \mathbin{;} \mathit{cons} \times \mathit{id} \mathbin{;} \mathit{id} \ddagger \mathit{inr}) \rrbracket$

$=$ definition $\ddagger = \times$, product

$\llbracket (\mathit{nil} \vartriangle \mathit{inl}) \triangledown (I\!I\beta \mathbin{;} \mathit{abide}_{I\!I} \mathbin{;} \mathit{cons} \times \mathit{inr}) \rrbracket$

$=$ definition $\beta = \mathit{hd} \vartriangle \mathit{tl}$, law Abide with $F, f, g := I\!I, \mathit{hd}, \mathit{tl}$

$\llbracket (\mathit{nil} \vartriangle \mathit{inl}) \triangledown (I\!I\mathit{hd} \vartriangle I\!I\mathit{tl} \mathbin{;} \mathit{cons} \times \mathit{inr}) \rrbracket$

$=$ { for elegance: aim at combining hd and tl into β }
product

$\llbracket (\mathit{nil} \vartriangle \mathit{inl}) \triangledown ((I\!I\mathit{hd} \mathbin{;} \mathit{cons}) \vartriangle (I\!I\mathit{tl} \mathbin{;} \mathit{inr})) \rrbracket$

$=$ $\vartriangle$ and $\triangledown$ abide, see paragraph 2.15

$\llbracket (\mathit{nil} \triangledown (I\!I\mathit{hd} \mathbin{;} \mathit{cons})) \vartriangle (\mathit{inl} \triangledown (I\!I\mathit{tl} \mathbin{;} \mathit{inr})) \rrbracket$

$=$ sum

$\llbracket (\mathit{id} + I\!I\mathit{hd} \mathbin{;} \mathit{nil} \triangledown \mathit{cons}) \vartriangle (\mathit{id} + I\!I\mathit{tl} \mathbin{;} \mathit{inl} \triangledown \mathit{inr}) \rrbracket$

$=$ definition $\mathit{nil} \triangledown \mathit{cons} = \alpha$, identity $\mathit{inl} \triangledown \mathit{inr} = \mathit{id}$

$\llbracket (\mathit{id} + I\!I\mathit{hd} \mathbin{;} \alpha) \vartriangle (\mathit{id} + I\!I\mathit{tl} \mathbin{;} \mathit{id}) \rrbracket$

$=$ product

$\llbracket (\mathit{id} + I\!I\mathit{hd}) \vartriangle (\mathit{id} + I\!I\mathit{tl}) \mathbin{;} (\alpha \times \mathit{id}) \rrbracket$

$=$ definition $\dagger$ and $\ddagger$

$\llbracket (\mathit{hd} \dagger \mathit{hd}) \vartriangle (\mathit{tl} \dagger \mathit{tl}) \mathbin{;} \alpha \ddagger \mathit{id} \rrbracket$

$=$ law biAbide

$\llbracket (\mathit{hd} \vartriangle \mathit{tl}) \dagger (\mathit{hd} \vartriangle \mathit{tl}) \mathbin{;} \mathit{abide}_\dagger \mathbin{;} \alpha \ddagger \mathit{id} \rrbracket$

$=$ definition $\beta = \mathit{hd} \vartriangle \mathit{tl}$

$\llbracket \beta \dagger \beta \mathbin{;} \mathit{abide}_\dagger \mathbin{;} \alpha \ddagger \mathit{id} \rrbracket$.

The expressions that we have derived for φ and ψ are quite symmetrical:

$$\begin{array}{lll} \varphi & = & [\![\varphi']\!] \qquad \text{where } \varphi' = \beta \dagger \beta \,;\, abide_\dagger \,;\, \alpha \ddagger id \\ \psi & = & (\![\psi']\!) \qquad \text{where } \psi' = \beta \dagger id \,;\, abide_\dagger \,;\, \alpha \ddagger \alpha \end{array}$$

so that

$$\begin{array}{lllll} transp' & = & (\![\varphi]\!) & = & (\![[\![\varphi']\!]]\!) \\ transp'' & = & [\![\psi]\!] & = & [\![(\![\psi']\!)]\!] \,. \end{array}$$

This came as a big surprise, though, in retrospect, some symmetry is to be expected. Laws cata- and ana-SELF give recursive equations for φ and ψ.

19 $\quad \varphi \;=\; \beta \dagger \beta \,;\, abide_\dagger \,;\, \alpha \ddagger id \,;\, id \ddagger \varphi \,;\, \beta^\cup$

20 $\quad \psi \;=\; \alpha^\cup \,;\, id \dagger \psi \,;\, \beta \dagger id \,;\, abide_\dagger \,;\, \alpha \ddagger \alpha.$

21 The proof. We are now ready to prove

$$transp' \;=\; (\![\varphi]\!)_{Sa\dagger} \;=\; [\![\psi]\!]_{La\ddagger} \;=\; transp''\,.$$

To this end we use law cata-ana-EQ2 14 with

$$\begin{array}{lll} \mathcal{F}f & = & \alpha^\cup \,;\, id_{Sa} \dagger f \,;\, \varphi \\ \mathcal{G}f & = & \psi \,;\, id_{La} \ddagger f \,;\, \beta^\cup. \end{array}$$

So, the proof obligation is:

$$\mathcal{F}\mathcal{G}f \;=\; \mathcal{G}\mathcal{F}f$$

for arbitrary f. Some intuitive understanding is provided by interpreting this equation at the point level. That is done in paragraph 22. Here is the formal proof.

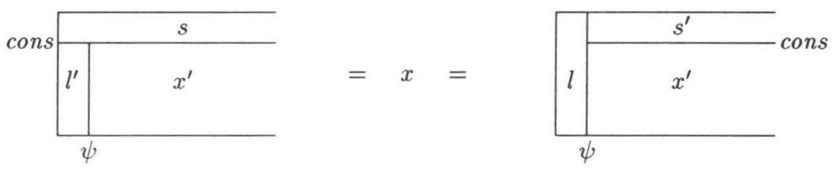
$$\begin{array}{ll}
 & \mathcal{G}\mathcal{F}f = \mathcal{F}\mathcal{G}f \\
\equiv & \qquad \text{definition } \mathcal{F} \text{ and } \mathcal{G} \\
 & \psi \,;\, id \ddagger (\alpha^\cup \,;\, id \dagger f \,;\, \varphi) \,;\, \beta^\cup \;=\; \\
 & \alpha^\cup \,;\, id \dagger (\psi \,;\, id \ddagger f \,;\, \beta^\cup) \,;\, \varphi \\
\equiv & \qquad \text{equations 19 and 20 derived for } \varphi \text{ and } \psi \\
 & \alpha^\cup \,;\, id \dagger \psi \,;\, \beta \dagger id \,;\, abide_\dagger \,;\, \alpha \ddagger \alpha \,;\, id \ddagger (\alpha^\cup \,;\, id \dagger f \,;\, \varphi) \,;\, \beta^\cup \;=\; \\
 & \alpha^\cup \,;\, id \dagger (\psi \,;\, id \ddagger f \,;\, \beta^\cup) \,;\, \beta \dagger \beta \,;\, abide_\dagger \,;\, \alpha \ddagger id \,;\, id \ddagger \varphi \,;\, \beta^\cup \\
\Leftarrow & \qquad \text{functor, Leibniz} \\
 & \beta \dagger id \,;\, abide_\dagger \,;\, \alpha \ddagger \alpha \,;\, id \ddagger \alpha^\cup \,;\, id \ddagger (id \dagger f) \;=\; \\
 & id \dagger (id \ddagger f) \,;\, id \dagger \beta^\cup \,;\, \beta \dagger \beta \,;\, abide_\dagger \,;\, \alpha \ddagger id \\
\equiv & \qquad \text{functor, inverse} \\
 & \beta \dagger id \,;\, abide_\dagger \,;\, \alpha \ddagger (id \dagger f) \;=\; \\
 & \beta \dagger (id \ddagger f) \,;\, abide_\dagger \,;\, \alpha \ddagger id \\
\equiv & \qquad \text{definition } abide_\dagger = (exl \dagger exl) \vartriangle (exr \dagger exr)\text{, definition } \ddagger = \times
\end{array}$$

$$\begin{array}{ll}
& \beta \dagger id \,;\, (exl \dagger exl) \vartriangle (exr \dagger exr) \,;\, \alpha \times (id \dagger f) \;\; = \\
& \beta \dagger (id \times f) \,;\, (exl \dagger exl) \vartriangle (exr \dagger exr) \,;\, \alpha \times id \\
\equiv & \qquad \text{product: } f \,;\, g \vartriangle h \,;\, j \times k = (f \,;\, g \,;\, j) \vartriangle (f \,;\, h \,;\, k) \\
& (\beta \dagger id \,;\, exl \dagger exl \,;\, \alpha) \vartriangle (\beta \dagger id \,;\, exr \dagger exr \,;\, id \dagger f) \;\; = \\
& (\beta \dagger (id \times f) \,;\, exl \dagger exl \,;\, \alpha) \vartriangle (\beta \dagger (id \times f) \,;\, exr \dagger exr \,;\, id) \\
\Leftarrow & \qquad \text{functor, Leibniz} \\
& id \,;\, exl = id \times f \,;\, exl \quad \wedge \quad id \,;\, exr \,;\, f = id \times f \,;\, exr \\
\equiv & \qquad \text{product} \\
& \textbf{true}.
\end{array}$$

This completes the proof that $transp' = transp''$.

22 Interpretation. Here is some interpretation of the proof obligation of law cata-ana-EQ2 for the case considered above. We shall not use this in any way. Formulated at the point level the proof obligation says amongst others the following.

Recall that $\varphi = zip' = zipwith\text{-}cons$ and $\psi = Lhd \vartriangle Ltl$. Let x be an arbitrary nonempty cons list of streams, and consider the following two different ways of composing and decomposing x.

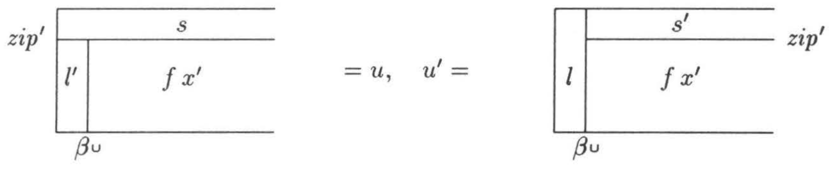


$$\begin{array}{llll}
\alpha^{\cup} \,;\, id \dagger \psi\colon & x \mapsto inr(s,(l',x')), & x = cons(s,y) & \psi\, y = (l',x') \\
\psi \,;\, id \ddagger \alpha^{\cup}\colon & x \mapsto (l, inr(s',x')), & \psi\, x = (l,z) & z = cons(s',x').
\end{array}$$

Let $f\colon x' \mapsto x''$ so that

$$\begin{array}{ll}
id \dagger (id \ddagger f)\colon & inr(s,(l',x')) \mapsto inr(s,(l',x'')) \\
id \ddagger (id \dagger f)\colon & (l, inr(s',x')) \mapsto (l, inr(s',x'')).
\end{array}$$

Let u, u' be the values that are composed and decomposed as follows.



$$\begin{array}{llll} id \dagger \beta^{\cup} ; \varphi: & inr(s,(l',x'')) \mapsto u, & (l',x'') = \beta\, v & zip'(s,v) = u \\ id \ddagger \varphi ; \beta^{\cup}: & (l, inr(s',x'')) \mapsto u', & zip'(s',x'') = w & (l,w) = \beta\, u'. \end{array}$$

Then $u = u'$.

For the formal proof above such an interpretation is neither needed nor useful.

4d Prepromorphisms

A *prepro*-equation is a recursion scheme that differs from the scheme for catamorphisms in that the recursive calls are preceded by some *preprocessing*. There exist simple sufficient conditions under which such an equation has a unique solution. The solution, if it exists, is termed a *prepromorphism.* An important prepromorphism is the so-called f*-cascade.* Of course, dualisation applies here; it provides the solution to the problem of proving the equivalence of two ways of defining f*-iterate.*

The proof technique may be of greater importance than the particular theorem for which it is used here; it uses infinite tuplings of morphisms.

23 Introduction: cascade. Let $\alpha, M = sumtype(\dagger)$. For given $f \colon a \to a$ the **f-cascade**, denoted $f^{\text{\tiny"}}$, is defined by

$$f^{\text{\tiny"}} \quad = \quad (\!|id \dagger Mf ; \alpha|\!)_{a\dagger} \quad : \quad Ma \to Ma\,.$$

Bird calls it **supermap-**f. Taking $\dagger$ to be the functor for cons lists and interpreting in $\mathcal{S}et$, the definition reads

$$f^{\text{\tiny"}}\,[a_0, a_1, \ldots, a_{n-1}] \quad = \quad [f^0 a_0, f^1 a_1, \ldots, f^{n-1} a_{n-1}]$$

where $f^i = f ; f ; \ldots ; f$ (i occurrences of f). Do not confuse f-cascade with the f-iterate f^{ω} of type $a \to Sa$ which was discussed in paragraph 3.48. Also, recall type-cata-FUSION:

$$(\!|f \dagger id ; \varphi|\!) \quad = \quad Mf ; (\!|\varphi|\!)$$

and observe that it is not applicable to f-cascade.

24 A problem. By cata-CHARN f-cascade is the unique solution for x of

$$x \quad = \quad \alpha^{\cup} ; F(x ; Mf) ; \alpha.$$

Now consider the slightly changed equation

$$y \quad = \quad \alpha^{\cup} ; F(Mf ; y) ; \alpha.$$

Is f-cascade also the unique solution for y? Since it is easily verified that

$$Mf ; f^{\text{\tiny"}} \quad = \quad f^{\text{\tiny"}} ; Mf\,,$$

it suffices to show that there is at most one solution y. Below we shall generalise the problem and abstract from the particular morphisms occurring in these equations. This particular problem is then answered in the affirmative in paragraph 36.

25 Generalisation: prepromorphism. Let F be arbitrary and α be μF. Let φ: $U\alpha \to U\alpha$ and ψ be an F-algebra. Consider the following equation in x

$$x \quad = \quad \alpha^{\cup} \,;\, F(\varphi \,;\, x) \,;\, \psi.$$

We call this equation a **prepro-equation.** To consider when it has a unique solution, let us reason informally in $\mathcal{S}et$ and for polynomial functors only. When φ is the identity, the argument to the recursive call is a subvalue, a proper constituent, of the original argument. Therefore, intuitively, the outcome of x is completely determined and well-defined by induction on the argument value: x equals $(\!|\psi|\!)$. (In $\mathcal{S}et$ and for polynomial F, any value in $U\alpha$ is constructed by repeated, finitely many, applications of α.) Now suppose that φ differs from id but still preserves "the structure," in the same way as a 'map' preserves the structure of its argument, and, on lists, any function that does not change the length. Then it is reasonable to expect that the equation has a unique solution, since at the recursive call in the right-hand side the argument structure is a substructure of the original argument, and so by induction on the structure of the argument (rather than on the value of the argument) x's outcome is completely determined and well-defined.

Even more generally, the above reasoning applies if φ has the property that for each argument t, viewed as a 'term' in α, the "complexity" of $\varphi(t)$ is at most that of t. For instance, a left linear binary tree may be transformed by φ into a right linear tree, so that the structure of the tree is changed but its "complexity" isn't. For lack of a better name we call the desired property of φ 'structure preservation'.

26 Structure preservation. One attempt to formalise "structure preservation" is:

$$\varphi \text{ preserves the } \dagger\text{-shape} \quad \equiv \quad \varphi \,;\, \mathit{shape} = \mathit{shape}$$

where $\alpha, M = \mathit{sumtype}(\dagger)$ and $\mathit{shape}_a = M\,!_a$ ($!_a$ being the unique morphism from a into 1). For example, each map Mf preserves the $\dagger$-shape, as shown in paragraph 3.62. However, this definition applies only to bifunctors and not to monofunctors in a sensible way. For suppose monofunctor F is given and bifunctor $\dagger$ is subsequently defined by $x \dagger y = Fy$, so that $F = a\dagger$. Then $\mathit{shape} = M\,! = (\!|\,! \dagger id \,;\, \alpha|\!) = (\!|\alpha|\!) = id$, and so only the identity morphism preserves the $\dagger$-shape.

Another attempt, and at present the best I can think of, is this:

$$\varphi \text{ preserves the } F\text{-structure} \quad \equiv \quad \varphi = (\!|\varepsilon \,;\, \alpha|\!) \quad \text{for some } \varepsilon\text{: } F \dot\to F,$$

where $\alpha = \mu F$. The definition is partly proof-generated and partly suggested by intuition; in particular the right hand side says

$$\varphi \quad = \quad \alpha^{\cup} \,;\, F\varphi \,;\, \varepsilon \,;\, \alpha.$$

The following facts show the generality of the definition.

Facts

27 *For ε: $F \dashrightarrow F$, we have $(\!|\varepsilon \mathbin{;} \alpha|\!) = Mu\,\varepsilon$ (where Mu is the functor discussed in paragraph 3.68 on the factorisation of type functors).*

28 *Structure preservation is closed under composition.*

29 *Let $-, M = sumtype(\dagger)$ and f: $a \to a$. Then Mf preserves the $a\dagger$-structure.*

30 *For lists, morphism $reverse$ preserves the structure, and so does $rotate$.*

Since we won't use these facts, we omit the proofs. I know of no $a\dagger$-structure preserving morphisms other than those mentioned in Fact 29 that preserve the $\dagger$-shape as well.

Characterisation and Fusion

We shall now prove the uniqueness and existence of a solution to the *prepro*-equation, called **prepromorphism**. (I owe the proof of the existence to Lambert Meertens.) We present also some corollaries of the uniqueness. Throughout the sequel we work in an arbitrary category which has enumerable products; the countably infinite tupling $f_0 \vartriangle f_1 \vartriangle \ldots$ is written $\Delta n ::\ f_n$, and the extractions (projections) are denoted ex_n. Endofunctor F is arbitrary, and we assume that an initial algebra $\alpha = \mu F$ exists.

31 Theorem. *Let ε: $F \dashrightarrow F$, let ψ be an F-algebra, and put $\varphi = (\!|\varepsilon \mathbin{;} \alpha|\!)$. Then*

$$x = \alpha^{\cup} \mathbin{;} F(\varphi \mathbin{;} x) \mathbin{;} \psi \quad \equiv \quad x = (\!|\Delta n ::\ F ex_{n+1} \mathbin{;} \varepsilon^n \mathbin{;} \psi|\!) \mathbin{;} ex_0 \qquad \text{prepro-CHARN}$$

where $\_^n$ denotes n-fold composition.

Proof. First we shall prove the implication to the right. Define for all n

$$\begin{array}{lcl} \psi_n & = & \varepsilon^n \mathbin{;} \psi \\ \psi'_n & = & F ex_{n+1} \mathbin{;} \varepsilon^n \mathbin{;} \psi \end{array}$$

Assume that a solution x_0 exists. Define a sequence $(n :: x_n)$ by induction on n:

$$x_{n+1} \quad = \quad \varphi \mathbin{;} x_n\,.$$

By induction on n we show that for all n

$$(\star) \qquad x_n \quad = \quad \alpha^{\cup} \mathbin{;} F x_{n+1} \mathbin{;} \psi_n.$$

Basis: immediate by definition of x_1 and ψ_0, and assumption on x_0.
Step: we calculate

$$\begin{array}{ll} & x_{n+1} \\ = & \qquad \text{definition } x_{n+1} \\ & \varphi \mathbin{;} x_n \\ = & \qquad \text{induction hypothesis} \end{array}$$

$\varphi \,;\, \alpha^{\cup} \,;\, Fx_{n+1} \,;\, \psi_n$

$=$ definition φ and cata-SELF

$\alpha^{\cup} \,;\, F\varphi \,;\, \varepsilon \,;\, Fx_{n+1} \,;\, \psi_n$

$=$ naturality ε, functoriality F

$\alpha^{\cup} \,;\, F(\varphi \,;\, x_{n+1}) \,;\, \varepsilon \,;\, \psi_n$

$=$ definition x_{n+2} and by definition $\psi_{n+1} = \varepsilon \,;\, \psi_n$

$\alpha^{\cup} \,;\, Fx_{n+2} \,;\, \psi_{n+1}$.

This completes the proof of ($\star$). Hence, for each n,

x_n

$=$ just proved: equation ($\star$)

$\alpha^{\cup} \,;\, Fx_{n+1} \,;\, \psi_n$

$=$ product

$\alpha^{\cup} \,;\, F((\Delta n ::\ x_n) \,;\, ex_{n+1}) \,;\, \psi_n$

$=$ functor

$\alpha^{\cup} \,;\, F(\Delta n ::\ x_n) \,;\, F ex_{n+1} \,;\, \psi_n$

$=$ by definition $\psi'_n = F ex_{n+1} \,;\, \psi_n$

$\alpha^{\cup} \,;\, F(\Delta n ::\ x_n) \,;\, \psi'_n$

showing that $(n :: x_n)$ is a collection of *mutumorphisms* that we have discussed in Section 4b. So, by the mutumorphism Theorem 8, or directly by the observation that

$$(\Delta n ::\ x_n) \quad = \quad \alpha^{\cup} \,;\, F(\Delta n ::\ x_n) \,;\, (\Delta n ::\ \psi'_n),$$

all the x's together form a catamorphism:

$$(\Delta n ::\ x_n) \quad = \quad (\!|\Delta n ::\ \psi'_n|\!)$$

hence

$$x_0 \quad = \quad (\!|\Delta n ::\ \psi'_n|\!) \,;\, ex_0 \,.$$

In summary, if a solution x_0 exists, then it is uniquely determined by the above equation.

Second, we show that $(\!|\Delta n :: \psi'|\!) \,;\, ex_0$ is a solution indeed. Putting $y_0 = (\!|\Delta n :: \psi'_n|\!) \,;\, ex_0$ and $y_{n+1} = \varphi \,;\, y_n$ it is easy to show

$y_0 = \alpha^{\cup} \,;\, F(\varphi \,;\, y_0) \,;\, \psi$

$\Leftarrow$

$y_1 = \alpha^{\cup} \,;\, F(\varphi \,;\, y_1) \,;\, \psi$

$\Leftarrow$

$y_2 = \alpha^{\cup} \,;\, F(\varphi \,;\, y_2) \,;\, \psi$

$\vdots$

But, unfortunately, this does not show that y_0 solves the equation. A better argument is needed; the following one has been designed by Lambert Meertens. We abbreviate $(\Delta n :: expr_n)$ to $\Delta expr_n$.

$y_0 = \alpha^{\cup} \,;\, F(\varphi \,;\, y_0) \,;\, \psi$

$\equiv$ definition y_0, and bring α to the left-hand side

$\alpha\,;\,(\!|\Delta\psi'_n|\!)\,;\,ex_0 = F(\varphi\,;\,(\!|\Delta\psi'_n|\!)\,;\,ex_0)\,;\,\psi$

(⋆) $\equiv$ lhs: cata-SELF;
rhs: equation $\varphi\,;\,(\!|\Delta\psi'_n|\!) = (\!|\Delta\psi'_n|\!)\,;\,\Delta ex_{n+1}$ proved below

$F(\!|\Delta\psi'_n|\!)\,;\,\Delta\psi'_n\,;\,ex_0 = F((\!|\Delta\psi'_n|\!)\,;\,\Delta ex_{n+1}\,;\,ex_0)\,;\,\psi$

$\Leftarrow$ functor, Leibniz

$\Delta\psi'_n\,;\,ex_0 = F(\Delta ex_{n+1}\,;\,ex_0)\,;\,\psi$

$\equiv$ product

$\psi'_0 = F ex_1\,;\,\psi$

$\equiv$ definition ψ'_0

true.

The equation used at step (⋆) is the crux of the proof. We prove it as follows, abbreviating Δex_{n+1} $(= \Delta n :: ex_{n+1})$ to *shift*.

$(\!|\Delta\psi'_n|\!)\,;\,shift = \varphi\,;\,(\!|\Delta\psi'_n|\!)$

$\equiv$ definition φ, cata-COMPOSE 26 (noting $\varepsilon\colon F \dot{\to} F$)

$(\!|\Delta\psi'_n|\!)\,;\,shift = (\!|\varepsilon\,;\,\Delta\psi'_n|\!)$

$\Leftarrow$ cata-FUSION

$\Delta\psi'_n\,;\,shift = F\,shift\,;\,\varepsilon\,;\,\Delta\psi'_n$

$\equiv$ at both sides Δ-FUSION: $f\,;\,\Delta g_i = \Delta(f\,;\,g_i)$

$\Delta((\Delta\psi'_n)\,;\,ex_{n+1}) = \Delta(F\,shift\,;\,\varepsilon\,;\,\psi'_n)$

$\equiv$ law: $\Delta f_i = \Delta g_i \;\equiv\; (forall\,i\colon f_i{=}g_i)$.
For all n:

$(\Delta\psi'_n)\,;\,ex_{n+1} = F\,shift\,;\,\varepsilon\,;\,\psi'_n$

$\equiv$ lhs: product, rhs: definition $\psi'_n = F ex_{n+1}\,;\,\varepsilon^n\,;\,\psi$ and naturality ε

$\psi'_{n+1} = F\,shift\,;\,F ex_{n+1}\,;\,\varepsilon^{n+1}\,;\,\psi$

$\equiv$ lhs: definition ψ'_{n+1}, rhs: functor and $shift\,;\,ex_{n+1} = ex_{n+2}$

true.

This completes the proof. □

32 Corollary. *Let $\varepsilon\colon F \dot{\to} F$ and ψ_0, ψ_1 be F-algebras. Let p_0 and p_1 be the F-prepromorphisms determined by ε, ψ_0 and ε, ψ_1 respectively. Then*

$$\psi_0\,;\,f = Ff\,;\,\psi_1 \qquad \Rightarrow \qquad p_0\,;\,f = p_1 \qquad\qquad \text{prepro-FUSION}$$

The proof is a standard calculation.

33 Variations. There are many more schemes for which the above proof technique may show that the prepro-version has a unique solution if the naked version has so. In particular, a "prepro-paramorphism" equation does have a unique solution.

The characterisation theorem gives the unique solution of a prepromorphism equation. This solution has not the form of a catamorphism. With some extra conditions it does.

34 Theorem. *Let* $\varepsilon\colon F \dot{\to} F$ *and* $\psi\colon Fa \to a$ *be arbitrary, and put* $\varphi = (\!|\varepsilon \mathbin{;} \alpha|\!)$. *Furthermore, let* $\chi\colon a \to a$ *be arbitrary, and put* $f = (\!|F\chi \mathbin{;} \psi|\!)$. *Then*

$$x = \alpha^{\cup} \mathbin{;} F(\varphi \mathbin{;} x) \mathbin{;} \psi \quad \equiv \quad x = f \qquad \text{prepro-cata-CHARN}$$

provided that $\varphi \mathbin{;} f = f \mathbin{;} \chi$, *which in turn follows from* $\chi\colon \psi \to_F \varepsilon \mathbin{;} \psi$.

Proof. From the definition of f it follows by cata-CHARN that

$$f \quad = \quad \alpha^{\cup} \mathbin{;} F(f \mathbin{;} \chi) \mathbin{;} \psi$$

and with $f \mathbin{;} \chi = \varphi \mathbin{;} f$ in addition it is immediate that f solves the equation for x. By prepro-CHARN f is then the unique solution. Notice that only the uniqueness part of prepro-CHARN is used. Finally, we derive the sufficient condition for the proviso $f \mathbin{;} \chi = \varphi \mathbin{;} f$ as follows.

$$\begin{array}{ll}
& f \mathbin{;} \chi = \varphi \mathbin{;} f \\
\equiv & \qquad \text{definition } f \text{ and } \varphi \\
& (\!|F\chi \mathbin{;} \psi|\!) \mathbin{;} \chi = (\!|\varepsilon \mathbin{;} \alpha|\!) \mathbin{;} (\!|F\chi \mathbin{;} \psi|\!) \\
\equiv & \qquad \text{rhs: cata-COMPOSE 26 noting that } \varepsilon\colon F \dot{\to} F \\
& (\!|F\chi \mathbin{;} \psi|\!) \mathbin{;} \chi = (\!|\varepsilon \mathbin{;} F\chi \mathbin{;} \psi|\!) \\
\Leftarrow & \qquad \text{cata-FUSION} \\
& \chi\colon F\chi \mathbin{;} \psi \to_F \varepsilon \mathbin{;} F\chi \mathbin{;} \psi \\
\equiv & \qquad \text{naturality } \varepsilon \\
& \chi\colon F\chi \mathbin{;} \psi \to_F F\chi \mathbin{;} \varepsilon \mathbin{;} \psi \\
\Leftarrow & \qquad \text{homo-ADHOC 3.24 (or: fold and unfold } \to_F) \\
& \chi\colon \psi \to_F \varepsilon \mathbin{;} \psi
\end{array}$$

as desired. □

With φ, ψ, χ as in the theorem, it is not necessarily true that for all f, the equation $f \mathbin{;} \chi = \varphi \mathbin{;} f$ holds. If the equation were to hold for all f, then in particular $id \mathbin{;} \chi = \varphi \mathbin{;} id$ so that $\chi = \varphi$. In the case of f*-iterate* in paragraph 37 below χ differs from φ.

Straightforward dualisation gives a characterisation of **postpromorphisms**. (Recall that formula $f\colon \varphi \succ\!\!-_F \psi$ means $\varphi \mathbin{;} Ff = f \mathbin{;} \psi$, thus dualising $f\colon \varphi \to_F \psi$.) Let $\alpha = \nu F$, the final F-co-algebra, assuming it exists.

35 Corollary. *Let* $\varepsilon\colon F \dot{\to} F$, *let* ψ *be an* F*-co-algebra, and put* $\varphi = [\!(\alpha \mathbin{;} \varepsilon)\!]$. *Then*

$$x = \psi \mathbin{;} F(x \mathbin{;} \varphi) \mathbin{;} \alpha^{\cup} \qquad \equiv \qquad x = [\!(\psi \mathbin{;} F\chi)\!] \qquad \text{postpro-ana-CHARN}$$

provided that $\chi \mathbin{;} [\!(\psi \mathbin{;} F\chi)\!] = [\!(\psi \mathbin{;} F\chi)\!] \mathbin{;} \varphi$, *which in turn follows from* $\chi\colon \psi \mathbin{;} \varepsilon \succ\!\!-_F \psi$.

36 Application: cascade. Recall the equations from paragraph 24 that triggered the notion of prepromorphism and the search for its characterisation:

$$\begin{array}{lcl} f^{\blacksquare} & = & \alpha^{\cup} \mathbin{;} F(f^{\blacksquare} \mathbin{;} Mf) \mathbin{;} \alpha \\ y & = & \alpha^{\cup} \mathbin{;} F(Mf \mathbin{;} y) \mathbin{;} \alpha \end{array}$$

where $\alpha, M = \mathit{sumtype}(\dagger)$, and $F = a\dagger$ and $f\colon\ a \to a$. Taking

$$\begin{array}{lcccccl} \varepsilon & & & = & f \dagger \mathit{id} & : & a\dagger \dot{\to} a\dagger \\ \varphi & = & \chi & = & Mf & = & (\!|\varepsilon \mathbin{;} \alpha|\!)\,, \end{array}$$

law prepro-cata-CHARN says that $f^{\blacksquare}$ is the unique solution for y.

37 Application: iterate. Let $\alpha, M = \mathit{prodtype}(\dagger)$. Recall from paragraphs 3.48,49,50 the definition of f^{ω}, for $f\colon\ a \to a$, and consider also the postpro-equation in x:

$$\begin{array}{lcll} f^{\omega} & = & \mathit{split}_{\dagger} \mathbin{;} F(f \mathbin{;} f^{\omega}) \mathbin{;} \alpha^{\cup} & :\quad a \to Ma \\ x & = & \mathit{split}_{\dagger} \mathbin{;} F(x \mathbin{;} Mf) \mathbin{;} \alpha^{\cup}, & \end{array}$$

where $F = a\dagger$ and $\mathit{split}_{\dagger}$ is some natural transformation typed $\mathit{split}_{\dagger}\colon\ I \dot{\to} I \dagger I$. For example, take $\dagger = \times$; then α is the final co-algebra of streams and

$$\mathit{split}_{\times} \;=\; \mathit{split} \;=\; \mathit{id} \vartriangle \mathit{id} \;:\; I \dot{\to} I \times I \;=\; I \dot{\to} I\!I\,.$$

Let us prove that f^{ω} is the unique solution of the equation in x. The two equations have the form

$$\begin{array}{lcl} f^{\omega} & = & \psi \mathbin{;} F(\chi \mathbin{;} f^{\omega}) \mathbin{;} \alpha^{\cup} \\ x & = & \psi \mathbin{;} F(x \mathbin{;} \varphi) \mathbin{;} \alpha^{\cup}, \end{array}$$

where

$$\begin{array}{lcllll} \psi & = & \mathit{split}_{\dagger} & & : & a \to a \dagger a \\ \chi & = & f & & : & a \to a \\ \varphi & = & Mf & = [\![\alpha \mathbin{;} \varepsilon]\!] & : & Ma \to Ma \\ \varepsilon & = & f \dagger \mathit{id} & & : & a\dagger \dot{\to} a\dagger\,. \end{array}$$

Notice that the case considered here is *not* the dual of f-cascade; in particular φ differs from χ. By law postpro-ana-CHARN f^{ω} is the unique solution for x provided that the condition of the law is satisfied. This is easily shown.

$$\begin{array}{ll} & \chi \;:\; \psi \mathbin{;} \varepsilon \succ_{F} \psi \\ \equiv & \qquad \text{definition } \chi, \psi, \varepsilon, F \\ & f \;:\; \mathit{split}_{\dagger} \mathbin{;} f \dagger \mathit{id} \succ_{a\dagger} \mathit{split}_{\dagger} \\ \equiv & \qquad \text{definition } \succ \\ & \mathit{split}_{\dagger} \mathbin{;} f \dagger \mathit{id} \mathbin{;} \mathit{id} \dagger f \;=\; f \mathbin{;} \mathit{split}_{\dagger} \\ \equiv & \qquad \text{functor, naturality } \mathit{split}_{\dagger} \\ & \mathsf{true} \end{array}$$

4e Conclusion

We have investigated several forms of equations that are similar to the equations for ana- and catamorphisms, and that have unique solutions. Such an equations may be used to define an algorithm, namely as the unique solution. In practice these kind of definitions are used indeed; in particular the mutumorphisms (algorithms defined by simultaneous induction) occur often, and also the prepro(cessing) and postpro(cessing) variations of inductive definitions seem quite natural. A FUSION law happens to be valid for the algorithms so defined, thus facilitating to exploit distributivity properties in the derivation of alternative (possibly more efficient) algorithms for the same function. The BANANA SPLIT law is a far reaching generalisation of phenomena like "loop fusion," which is also quite relevant for efficiency improvement.

Some of the alternative induction schemes for defining algorithms are simply a change of view on the datatype involved. There are various ways in which one can look at a cons list, and each of the views brings forth a way to define algorithms on the datatype of cons lists. We have formally described the proof obligations when this technique is applied.

As an aside in this chapter we have investigated conditions under which an algorithm can be expressed both as an anamorphism and as a catamorphism. To illustrate this we have proved the equality of two ways of defining a transpose (from a cons list of streams into a stream of cons lists). The case study shows several aspects that need further improvement and investigation (in our opinion): there seems to be ample opportunity for machine assistence in the calculations, it seems that the naturality properties of the ingredients can be exploited more than we have done, and the avoidance of a "combinatorial explosion" needs further attention (in previous proof attempts there were too many steps that expressed too little each, whereas their combined result was too much for one step).

Chapter 5

Datatype Laws without Signatures

Using the well-known categorical notion of 'functor' one may define the concept of datatype (algebra) without being forced to introduce a signature, that is, names and typings for the individual sorts (types) and operations involved. This has proved to be advantageous for those theory developments where one is not interested in the syntactic appearance of an algebra.

The newly developed categorical notion of 'transformer' allows the same approach to laws: without using signatures one can define the concept of law for datatypes (lawful algebras), and investigate the equational specification of datatypes in a syntax-free way. A transformer is a special kind of functor and also a natural transformation on the level of dialgebras. Transformers are quite expressive, satisfy several closure properties, and are related to naturality and Wadler's Theorems For Free. In fact, any colimit is an initial lawful algebra.

5a Introduction

1 The problem. Most mathematical formalisations of the intuitive notion of 'datatype' define that notion as a (many-sorted) algebra, possibly provided with some (conditional) equations, which we call 'laws'. Such algebras themselves are often formalised with help of the notion of 'signature' or, more categorically and slightly more abstract, with the notion of 'sketch' as described by Barr and Wells [7]. A signature gives the syntactic appearance of the algebra; it gives the names of the sorts (types), the names and arities of the operations and constants, and for each operation a syntactic indication of the types of its arguments and result. No doubt, signatures are indispensable for large-scale programming tasks, and a theory that deals with signatures may be quite useful. Such a theory contains theorems on aspects of name-clashes, renaming, scope rules, persistency and so on. However, sometimes we would like to be able to abstract from syntactic aspects, for example, when investigating the existence of certain kinds of algebras, or the (semantic) relations between algebras. In fact, one should abstract from naming even in the definition of such basic concepts as 'homomorphism'. For the lawless case this is possible indeed, thanks to the notion of *functor*. A functor characterises the type structure of an algebra without naming the sort or any of the operations involved. Functors satisfy just one or two very simple axioms, and —almost unbelievable— that is all that is needed to develop a large body of useful theorems about algebras. The problem for which we propose a solution, is the following.

> Formalise the notion of 'law' (an equation or conditional equation for the operations of an algebra) without introducing signatures, in particular naming and setting up a syntax of terms.

Remarkably, in all texts where functors are used to characterise algebras, signatures (or sketches) are introduced when it comes to laws. Clearly, this is a hindrance to theory development, since it forces to deal with aspects (syntax) that should have been abstracted from. About the use of functors to describe algebras Pierce [60, remark 2.2.3] explicitly says:

> The framework has apparently never been extended to include algebras with equations.

2 The solution. We shall propose a categorical description of 'law' that avoids naming, and is of the same simplicity as the definition of 'functor'. To be specific, each of the two terms of an equation shall be just a mapping T, from (di)algebras to (di)algebras, that satisfies a particular so-called TRANSFORMER property; and such a T is called a *transformer*. A transformer is a special kind of functor, as well as a natural transformation. There are several theorems on transformers that should be true if the notion is to be of any use. In Section 5c we show that transformers can be composed in various ways to form transformers again, and are thus as expressive as the usual syntactic terms in conditional equations. Also, 'laws' are closed under conjunction. Moreover, there is some relation between transformers and Wadler's [74] Theorems For Free theorem and naturality. In Section 5d we give, for each law E, conditions under which the class $\mathcal{A}lg(F,E)$ of F-algebras satisfying E is closed under subalgebras, product algebras and homomorphic images. In Section 5e we give conditions under which $\mathcal{A}lg(F,E)$ has an initial object, the initial F,E-algebra. We also show how to exploit a law E of the initial F,E-algebra in programming. In Section 5f we show that any colimit is in fact an initial F,E-algebra for some suitable choice of F and E. And finally, in Section 5g, we show the transformers in action in the theory of equational specification of datatypes, by proving two little theorems concerning the isomorphy of two differently specified datatypes.

The simplicity of the proofs of the claims above demonstrates the success of our formalisation.

3 Running example: Trees. In this chapter we use the algebra of **binary structures** over a as an example. The default category is $\mathcal{S}et$. The carrier of the algebra is the set Ba that consists of all finite Binary structures with values from a at the tips. There are three functions *nil*, *tip*, and *join*.

nil	:	$\mathit{1} \to Ba$	the nil structure
tip	:	$a \to Ba$	the tip former
$join$	:	$\mathit{II}\, Ba \to Ba$	the join operation, joining two structures.

Formally, the algebra is defined by

$$nil \triangledown tip \triangledown join,\ B \quad = \quad sumtype(\dagger) \qquad\qquad \text{where } a\,\dagger \ = \ \underline{\mathit{1}} + \underline{a} + \mathit{II}$$

so that

$$nil \triangledown tip \triangledown join \quad : \quad \underline{\mathit{1}} + \underline{a} + \mathit{II}B \dot{\to} B\,.$$

The notation x join y is an alternative for $join(x, y)$. The function that sends each structure to its size (number of tips) is defined by

$$\begin{aligned} nil \,;\, size &= zero \\ tip \,;\, size &= one \\ join \,;\, size &= \mathit{II}size \,;\, add \end{aligned}$$

that is,

$$nil \triangledown tip \triangledown join \,;\, size \quad = \quad id \dagger size \,;\, zero \triangledown one \triangledown add$$

so that

$$size \quad = \quad (\!|zero \triangledown one \triangledown add|\!)\,.$$

The name **tree** is an abbreviation of binary structure. In paragraph 12 we shall see that these binary structures are effectively lists, bags, or sets when operation *join* and *nil* satisfy suitable laws.

5b Transformer and Law

4 Abstracting from syntax. Conventionally an equation for algebra φ is just a pair of terms built from variables, the constituent operations of φ, and some fixed standard operations. An equation is valid if the two terms are equal for all values of the variables. We are going to model a syntactic term as a morphism that has the values of the variables as source. For example, the two terms 'x' and 'x join x' (with variable x of type Ba) are modeled by morphisms id and $id \vartriangle id \,;\, join$ of type $Ba \to Ba$. So, an equation for φ is modeled by a pair of terms $(T\varphi, T'\varphi)$, T and T' being mappings of morphisms which we call '*transformer*'. This faces us with the following problem: what properties must we require of an arbitrary mapping T in order that it models a classical syntactic term? Or, rather, what properties of classical syntactic terms are semantically essential, and how can we formalise these as properties of a transformer T? Of course, T has to be well behaved with respect to typing (like functors). And besides that, the resulting morphism

$T\varphi$ should be built out of φ in a way that is independent of the properties of the particular φ itself and its carrier. For example, for I-algebras we disallow the following mappings as transformer.

$$\begin{array}{lcl} T\varphi & = & \textbf{if } \varphi \text{ has carrier } \textit{nat} \textbf{ then } \textit{succ} \textbf{ else } \varphi \\ T\varphi & = & \textbf{if } \varphi \text{ is bijective } \textbf{then} \text{ the inverse of } \varphi \textbf{ else } \varphi\,. \end{array}$$

We disallow these not only for intuitive reasons, but also because with these mappings we cannot prove the things that we want to hold. A tentative definition that a transformer is a natural transformation in the underlying default category does meet our intuitive wish and enables us to prove several desirable theorems, but it makes some terms unexpressible as a transformer (see Theorem 20). So we need a weaker requirement to be imposed on a mapping in order that it can be said to model the intuitive notion of term. A way out is to introduce a syntax of terms, and require T to be expressed in that syntax. That is just what conventionally is done up to now, and what we want to avoid.

5 A property observed. Our solution is to impose a property, saying that homomorphisms are mapped to homomorphisms. This seems to be precisely what is needed to carry the proofs through. And it is also reasonable from an intuitive point of view. Let me try to explain it. (You may skip this informal explanation; the proofs of Theorem 14 and 16 are just the formalisation of the argument here.) Suppose $\varphi\colon a \to a$ and $T\varphi\colon Ha \to Ja$.

Following Meertens [48] we view a term as a box with several input and output gates. Such boxes can be wired together to form composite boxes. You may imagine how the wiring for sequential composition (;) and parallel composition ($\times$) would look like. You can also easily construct boxes for the duplication $id \vartriangle id$, and for the swap $\mathit{exr} \vartriangle \mathit{exl}$. Now imagine a box (term) $T\varphi\colon Ha \to Ja$ built with several copies of a box for $\varphi\colon a \to a$. Suppose you insert on each of the output lines a box for $f\colon a \to b$, thus forming a composite box $T\varphi \,;\, Jf\colon Ha \to Jb$. You can then shift each box for f along the wires in the direction of the input side, through all compositions, until it arrives just after a box for φ. If $\varphi \,;\, f = f \,;\, \psi$ then you can replace the box for φ with one for ψ, and put the box for f just in front of ψ, and continue shifting the box for f along the lines. In this way, eventually, f is shifted to the input gates. Thus, if $\varphi \,;\, f = f \,;\, \psi$ then you may expect that $T\varphi \,;\, Jf = Hf \,;\, T\psi$.

6 Generalisation. Generalising, in the above observation, $\varphi\colon a \to a$ and $\psi\colon b \to b$ to $\varphi\colon Fa \to Ga$ and $\psi\colon Fb \to Gb$, it is reasonable to expect in the same way that $\varphi \,;\, Gf = Ff \,;\, \psi$ implies $T\varphi \,;\, Jf = Hf \,;\, T\psi$. Using dialgebras we can formulate this as:

(a) $\quad f\colon \varphi \to_{F,G} \psi \quad \Rightarrow \quad f\colon T\varphi \to_{H,J} T\psi\,.$

Notice that this formula makes sense even if not all the entities are in one and the same category. The most general typing is easily found: there are categories $\mathcal{A}, \mathcal{B}, \mathcal{C}$, the functors are typed $F, G\colon \mathcal{A} \to \mathcal{B}$ and $H, J\colon \mathcal{A} \to \mathcal{C}$ and $T\colon \mathcal{D}iAlg(F,G) \to \mathcal{D}iAlg(H,J)$. It follows that f is in $\mathcal{A}$, φ, ψ are in $\mathcal{B}$, and $T\varphi, T\psi$ are in $\mathcal{C}$.

We shall now derive two alternative but equivalent formulations of property (a).

Functoriality. Notice that, apparently, T sends $\mathcal{D}iAlg(F,G)$-objects to $\mathcal{D}iAlg(H,J)$-objects. Actually, if we extend T by *defining* $Tf = f$ for each $\mathcal{D}iAlg(F,G)$-morphism f, then property (a) above is one of the axioms for T to be a functor

(b.0) $\quad T \; : \; \mathcal{D}iAlg(F,G) \rightarrow \mathcal{D}iAlg(H,J)\,.$

The other functor axioms are the equations $T\mathit{id} = \mathit{id}$ and $T(f\,;g) = Tf\,;Tg$; these are trivially valid by defining $Tf = f$. Thus extended, T is a functor indeed. That T is the identity on the morphisms in $\mathcal{D}iAlg(F,G)$ can also be formalised as

(b.1) $\quad U'T \; = \; U\,,$

where U, U' are the appropriate forgetful functors,

$$U\colon \mathcal{D}iAlg(F,G) \rightarrow \mathcal{A} \qquad \text{and} \qquad U'\colon \mathcal{D}iAlg(H,J) \rightarrow \mathcal{A}\,.$$

Clearly, a T satisfying (a) can be extended to a T satisfying (b.0) and (b.1), and conversely, a T satisfying (b.0) and (b.1) also satisfies (a).

Naturality. There is still another reading of the typing of T and property (a), namely

$$\begin{array}{lll} T\varphi\colon HU\varphi \rightarrow JU\varphi & & \text{for each } F,G\text{-dialgebra } \varphi \\ HUf\,;T\psi \;=\; T\varphi\,;JUf & & \text{for each } f\colon \varphi \rightarrow_{F,G} \psi\,, \end{array}$$

where $U\colon \mathcal{D}iAlg(F,G) \rightarrow \mathcal{A}$. So, T is a natural transformation in $\mathcal{C}$ from HU to JU,

(c) $\quad T \; : \; HU \dot{\rightarrow} JU\,.$

And each T satisfying (c) also satisfies (a). This was observed by Ross Paterson and Peter de Bruin. The latter also pointed out, when this text was almost finished, that transformers are a —slight— generalisation of the *semantic operations* described by Manes [44]. Manes investigates a relation with *syntactic operations*, but doesn't discuss most topics of this chapter.

In the following definition we choose one of the three equivalent ways (a), (b.0,b.1) and (c) to characterise transformers.

Definitions (Transformer, Law)

7 *Let* $F,G\colon \mathcal{A} \rightarrow \mathcal{B}$ *and* $H,J\colon \mathcal{A} \rightarrow \mathcal{C}$ *be functors. Then a* **transformer** *of* **type** $(F,G) \rightarrow (H,J)$ *is: a mapping* T *from* F,G*-dialgebras to* H,J*-dialgebras satisfying*

$$f\colon \varphi \rightarrow_{F,G} \psi \quad\Rightarrow\quad f\colon T\varphi \rightarrow_{H,J} T\psi \qquad\qquad \text{TRANSFORMER}$$

that is,

$$\varphi\,;Gf = Ff\,;\psi \quad\Rightarrow\quad T\varphi\,;Jf = Hf\,;T\psi$$

for all a,b, $f\colon a \rightarrow_{\mathcal{A}} b$, $\varphi\colon Fa \rightarrow_{\mathcal{B}} Ga$, *and* $\psi\colon Fb \rightarrow_{\mathcal{B}} Gb$.
The pairs (F,G) *and* (H,J) *are called the* **source type** *and* **target type** *respectively.*

8 *A* **law** *is: a pair of transformers of the same type, called the* type of the law. *For a law* $E = (T, T')$ *we say* E **holds for** φ *if:* $T\varphi = T'\varphi$. *Alternatively we also say* $E\varphi$ **holds** *or* φ **satisfies** E*; a more formal notation would be* $\models E\varphi$ *or* $\varphi \models E$.

9 *A* **conditional law** *is: a pair* E, E' *of laws, both having the same source type, that is, both being applicable to the same dialgebras. Such a law holds for* φ *if:* $E\varphi$ *implies* $E'\varphi$. *(We shall hardly discuss conditional laws.)*

Often we will take $\mathcal{A} = \mathcal{B} = \mathcal{C}$ in applications of transformers, so that F, G, H, J are endofunctors on the default category $\mathcal{C}$. It is straightforward to extend the definitions in such a way that transformers and laws accept several arguments rather than one. Actually, this is already covered by the above definition by taking $\mathcal{B}$ to be a suitable product category. For example, when $\mathcal{B} = \mathcal{B}' \times \mathcal{B}'$, then the transformer gets as argument a pair from $\mathcal{B}'$. This will occur in Section 5g.

10 Use of laws. If a law is prescribed for an F-algebra φ, then of course the law must have source type (F, I), that is, $G = I$. The definition of law and transformer may seem unnecessarily general for this application. However, in composing transformers we need the more general form with $G \neq I$, even though the entire composite transformer has $G = I$; see Theorem 16. A dual remark holds for co-algebras. As regards to the target type a similar observation holds; in this case either $H = I$ or $J = I$ depending on the use of the law. We illustrate both possibilities for the use of a law in the following example; yet another use, related to the first one, is discussed in Section 5g.

11 Example (Trees continued) Consider the law "x join y = y join x," which we shall formalise later. Here the result type of the two terms, viewed as functions of x and y, is $I\!I Ba \to Ba$, where Ba is the carrier. So the transformers that model these terms have target type $(I\!I, I)$, that is, $J = I$ (and $H = I\!I$). The law induces an equivalence relation on Ba that is a congruence for the algebra, namely the least equivalence relation that contains all pairs (x join y, y join x) (as indicated by the law) and is closed under the operations of the algebra, meaning that with (x, x') and (y, y') it also contains ($tip\, x$, $tip\, x'$) and (x join y, x' join y'). Imposing the law on the algebra means to identify equivalent elements and to consider the induced quotient algebra.

Now consider the law "$size\, x \bmod 2 = 0$" (also formalised later). Here the result type of the two terms, viewed as functions of x, is $Ba \to nat$, so in this case the transformers have target type $(B, \underline{nat})$, that is, $H = I$ (and $J = \underline{nat}$; recall $\underline{x}$ is the constant mapping, or functor). Imposing the law on the algebra means to leave out from the carrier the trees with odd size and to look for an "induced subalgebra" (which might not exist at all). □

In the sequel we shall illustrate our notion of transformer and law mainly for the case $G = I = J$: applicable to algebras and meant to identify elements of the carrier. We are in fact primarily interested in the rôle of the TRANSFORMER property , since we conjecture this to be the heart of the formalisation of the semantics of terms. Further applications of transformers and laws await future research.

12 Example (Trees continued) By making *nil* neutral for *join* (that is, making *nil* the identity for *join*) it behaves properly as 'empty': joining *nil* to a structure yields the same structure again.

By further imposing associativity of *join* the trees become effectively lists or sequences, known as *join lists*: since $x\,\mathrm{join}(y\,\mathrm{join}\,z) = (x\,\mathrm{join}\,y)\mathrm{join}\,z$, the parentheses may be omitted, and that structure can be denoted by x join y join z , the usual notation for a list.

Bags result by imposing commutativity of *join* as well: since x join $y = y$ join x , the order in which the elements are joined to a structure is insignificant.

Finally, *sets* are obtained if *join* is made absorptive (idempotent) in addition: since x join $x = x$, the multiplicity of the elements in (the denotation of) a structure is insignificant, as for sets.

Meertens [47] attributes this observation to H.J. Boom, and Backhouse [3] calls these types the Boom-hierarchy. We shall show how the laws can be expressed as pairs of transformers. The laws are applicable to every $a\dagger$-algebra, not only to the initial one (the trees). Also the law for 'even size' of trees is formalised; this one has a feature not present in the others.

Let $\varphi = e \triangledown f \triangledown \oplus$: $\imath + a + I\!I b \to b$ be an arbitrary $a\dagger$-algebra. Observe that the constituent operations of φ can be expressed as follows.

$$\begin{array}{lclcl} e & = & in_{3,0} \,;\, \varphi & : & \imath \to b \\ f & = & in_{3,1} \,;\, \varphi & : & a \to b \\ \oplus & = & in_{3,2} \,;\, \varphi & : & I\!I b \to b\,. \end{array}$$

So when we say $T\varphi = \ldots e \ldots \oplus \ldots$, we actually mean the right hand side that is obtained by substituting the above definitions for $e, f,$ and $\oplus$. We discuss the simplest laws first.

Absorptivity. In order to express $x \oplus x = x$ for all x in b, take

$$T\varphi \;=\; split \,;\, \oplus \qquad \text{and} \qquad T'\varphi \;=\; id\,,$$

where $split = id \vartriangle id$. Here and in the following examples, Theorems 16, 17, and 18 imply the validity of the TRANSFORMER property for both T and T' on account of the way they are composed out of basic transformers. But it may be instructive to verify the property at least once explicitly. We do it here for T, the verification for T' is trivial.

Consider two arbitrary $a\dagger$-algebras $\varphi = d \triangledown g \triangledown \oplus$ and $\psi = e \triangledown h \triangledown \otimes$. Suppose f is a homomorphism from φ to ψ,

$$f \qquad : \quad \psi \to_{a\dagger} \psi$$

that is,

$$\begin{array}{lcl} d \,;\, f & = & e \\ g \,;\, f & = & h \\ \oplus \,;\, f = I\!I f \,;\, \otimes\,. \end{array}$$

Then f is a homomorphism from $T\varphi$ to $T\psi$,

$$f\colon T\varphi \to_{I,I} T\psi$$

$\equiv$ definition T, definition $\to$

$$\begin{array}{ll} & split \,;\, \oplus \,;\, f = f \,;\, split \,;\, \otimes \\ \equiv & \quad \text{lhs: assumption on } f \text{, rhs: naturality } split\colon I \dot{\to} I\!I \\ & split \,;\, I\!I f \,;\, \otimes = split \,;\, I\!I f \,;\, \otimes \\ \equiv & \quad \text{Leibniz} \\ & \textbf{true}. \end{array}$$

Commutativity. To express $x \oplus y = y \oplus x$ for all x, y in b, take

$$T\varphi \;=\; \oplus \qquad \text{and} \qquad T'\varphi \;=\; swap \,;\, \oplus,$$

where $swap = exr \vartriangle exl$.

Neutrality. To express $e \oplus x = x$ for all x in b, take

$$T\varphi \;=\; (! \,;\, e) \vartriangle id \,;\, \oplus \qquad \text{and} \qquad T'\varphi \;=\; id\,.$$

Here $!\colon b \to 1$ is the unique morphism into the unit type 1.

Associativity. To express $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for all x, y, z in b, take

$$T\varphi \;=\; \oplus \times id \;;\; \oplus \qquad \text{and} \qquad T'\varphi \;=\; assoc \;;\; id \times \oplus \;;\; \oplus$$

where

$$assoc \;=\; (exl \,;\, exl) \vartriangle ((exl \,;\, exr) \vartriangle exr) \;:\; (X \times Y) \times Z \dot{\to} X \times (Y \times Z)$$

Here functors X, Y, Z stand for $Ex_{3,0}, Ex_{3,1}, Ex_{3,2}$.

Even size. A problem in expressing $(size\, x) \bmod 2 = 0$ is that $size$ is not an operation of the algebra. Given that φ is the initial algebra, $size$ is just $(\!|\varphi \to zero \triangledown one \triangledown add|\!)$, as we have shown earlier. However, T should be applicable to every algebra φ, not just an initial one. In fact, it is a problem what "$size$" means at all if φ is not initial. One way out is this. First extend the algebra with an additional operation ψ specified by the 'defining equations' for $size$: "$\varphi \,;\, \psi = F\psi \,;\, zero \triangledown one \triangledown add$". This is discussed in detail in Section 5g and gives an F, G-bialgebra (φ, ψ) for some G. Then form, for arbitrary F, G-bialgebra (φ, ψ), the law E suggested by

$$E(\varphi, \psi) \;=\; \text{"}\; \varphi \,;\, \psi \,;\, mod2 = zero \;\text{"}\,.$$

Transformers are applicable to bialgebras indeed, by a suitable instantiation of $\mathcal{A}, \mathcal{B}$ and the functors in the definition of transformer. □

After all these examples one might wonder whether there are morphism mappings that have type $(F, G) \to (H, J)$ for some F, G, H, J and are not transformers.

13 Fact. *The morphism mappings given at the beginning of the section are not transformers; the* TRANSFORMER *property is not valid for them.*

5c Expressiveness of transformers and laws

We shall see in this section that the TRANSFORMER property for a mapping of type $(F, G) \to (H, J)$ follows from the Theorems For Free theorem (provided it is applicable to the mapping). Further, the usual syntactic ways to compose terms are also applicable to transformers: they are closed under composition and substitution, and the identity mapping and each functor and constant mapping is a transformer. Thus transformers are at least as expressive as syntactic terms. Also, natural transformations of a higher type are transformers, but not conversely. And, finally, laws are closed under conjunction.

* * *

14 Theorem. *Let T be a morphism mapping of type $\forall\alpha :: (F\alpha \to G\alpha) \to (H\alpha \to J\alpha)$, and suppose that the Theorems For Free theorem of Wadler [74] is applicable to T. Then T is a transformer of type $(F, G) \to (H, J)$.*

Proof. We use the notation of Wadler [74] except for our choice of identifiers and the order of composition:

Each function f denotes a relation, namely $(x,y){\in}f \;\equiv\; f(x){=}y$. Composition $;$ is extended to relations: $(x,z) \in R\,;S$ iff there exists an y for which $(x,y) \in R$ and $(y,z) \in S$. For relations R and S and relation mapping F, expressions $R \to S$ and $\forall r :: Fr$ denote a relation too:

$$\begin{array}{rcl}(f,g) \in (R \to S) & \equiv & R\,; g \subseteq f\,; S \\ & \equiv & \forall x, y :: \;(x,y) \in R \;\Rightarrow\; (f\,x, g\,y) \in S \\ (T,T') \in (\forall r :: Fr) & \equiv & \forall a, b,\ R\colon a{\Leftrightarrow}b :: \;(T_a, T'_b) \in FR\,.\end{array}$$

Here, $a{\Leftrightarrow}b$ is the type of relations containing pairs (x,y) with $x \in a$ and $y \in b$. All morphisms (functions) are required to be total, so that $f \subseteq g$ equivales $f = g$.

The task is to prove that TRANSFORMER is valid for T. For this we argue

$$\begin{array}{ll} & T\colon \forall\alpha :: (F\alpha \to G\alpha) \to (H\alpha \to J\alpha) \\ \Rightarrow & \quad \text{Theorems for Free — applicability assumed} \\ & (T,T) \in \forall r :: (Fr \to Gr) \to (Hr \to Jr) \\ \equiv & \quad \text{definition } \forall \\ & \forall\text{reln } R\colon a \Leftrightarrow b.\ (T_a, T_b) \in (FR \to GR) \to (HR \to JR) \\ \equiv & \quad \text{definition } \to \text{ (second alternative)} \\ & \forall\text{reln } R\colon a \Leftrightarrow b.\ \forall\varphi,\psi.\ (\varphi,\psi) \in (FR \to GR) \;\Rightarrow\; (T_a\varphi, T_b\psi) \in (HR \to JR) \\ \equiv & \quad \text{definition } \to \text{ (first alternative) at both sides} \\ & \forall\text{reln } R\colon a \Leftrightarrow b.\ \forall\varphi,\psi.\ FR\,;\psi \subseteq \varphi\,; GR \;\Rightarrow\; HR\,; T_b\psi \subseteq T_a\varphi\,; JR \\ \Rightarrow & \quad \text{taking } R\colon a \Leftrightarrow b \text{ to be a function } f\colon a \to b \\ & \forall\text{fctn } f\colon a \to b.\ \forall\varphi,\psi.\ Ff\,;\psi = \varphi\,; Gf \;\Rightarrow\; Hf\,; T_b\psi = T_a\varphi\,; Jf \end{array}$$

which is exactly the required TRANSFORMER property. □

One condition on T for the applicability of the Theorems For Free theorem is that T is lambda-definable (there may be more conditions on the category — this is not clear to me). In the definition of transformer each morphism mapping is allowed, even those that are not lambda-definable. Theorems For Free suggests that TRANSFORMER is a crucial property (and provides an alternative rationale for requiring this property to hold for transformers). Moreover, working in a 'functional' categorical setting, it seems that Theorems For Free suggests no stronger property for transformers.

15 Composite transformers. Here follow some theorems showing how transformers may be composed to form transformers again.

16 Theorem. *The following equations define transformers of the type indicated, provided that T, T' are transformers of type $(F,G) \to (H,J)$ and $(F',G') \to (H',J')$ respectively, and that the well-formedness conditions at the right hold.*

$$\begin{array}{lcllll}
I\varphi & = & \varphi & I & : & (F,G) \to (F,G) & \\
\underline{f}\varphi & = & f & \underline{f} & : & (F,G) \to (\underline{\mathrm{src} f, \mathrm{tgt} f}) & \\
\underline{\varepsilon}\varphi & = & \varepsilon & \underline{\varepsilon} & : & (F,G) \to (H,J) & \varepsilon\colon H \dot{\to} J \\
(T;T')\varphi & = & T\varphi \,;\, T'\varphi & (T;T') & : & (F,G) \to (H,J') & \begin{cases} (F,G) = (F',G') \\ J = H' \end{cases} \\
(T \circ T')\varphi & = & T(T'\varphi) & (T \circ T') & : & (F',G') \to (H,J) & (H',J') = (F,G) \\
(\!|\_|\!)_F\, \varphi & = & (\!|\varphi|\!)_F & (\!|\_|\!) & : & (F,I) \to (\underline{U\mu F}, I) & \mu F \text{ exists.}
\end{array}$$

Proof. The correct typing is immediate for all these transformers. As regards the TRANSFORMER property for $(\!|\_|\!)$ we argue as follows. Let $a = U\mu F$. Then

$$\begin{array}{ll}
 & \underline{a}f \,;\, (\!|\_|\!)\varphi = (\!|\_|\!)\psi \,;\, If \\
\equiv & \qquad \text{definition } (\!|\_|\!),\ \underline{a},\ \text{and } I \\
 & (\!|\varphi|\!) = (\!|\psi|\!) \,;\, f \\
\Leftarrow & \qquad \text{cata-FUSION} \\
 & \psi \,;\, f = Ff \,;\, \varphi.
\end{array}$$

This fact is a special instance of cata-TRANSFORMER 57: take $x \dagger y = Fy$, so that $I \dagger I = F$ and $L = \underline{U\mu F}$.

As regards the TRANSFORMER property of the composite $T \circ T'$ we argue:

$$\begin{array}{ll}
 & T(T'\varphi) \,;\, Jf = Hf \,;\, T(T'\psi) \\
\Leftarrow & \qquad \text{TRANSFORMER } T,\ \text{noting that } (F,G) = (H',J') \\
 & T'\varphi \,;\, J'f = H'f \,;\, T'\psi \\
\Leftarrow & \qquad \text{TRANSFORMER } T' \\
 & \varphi \,;\, G'f = F'f \,;\, \psi.
\end{array}$$

Rephrased with the $\to_{F,G}$ notation, this calculation is but a special instance of the proof that the composition of functors is a functor again.

The other parts are proved similarly to $T \circ T'$. Actually, that $\underline{f}$ is a transformer follows also from the fact that $\underline{\varepsilon}$ is a transformer, since $f\colon a \to b$ implies $f\colon \underline{a} \dot{\to} \underline{b}$. □

17 Theorem. *Let T be a transformer of type $(F,G) \to (H,J)$, and K an endofunctor on the source category of F,G,H,J. Then T is also a transformer of type $(FK,GK) \to (HK,JK)$.*

Proof. The typing is clearly correct. As regards the TRANSFORMER property we argue:

$$\begin{array}{ll} & f\colon T\varphi \to_{HK,JK} T\psi \\ \equiv & \qquad \text{unfold, fold} \\ & Kf\colon T\varphi \to_{H,J} T\psi \\ \Leftarrow & \qquad \text{TRANSFORMER for } T \text{ of type } (F,G) \to (H,J) \\ & Kf\colon \varphi \to_{F,G} \psi \\ \equiv & \qquad \text{unfold, fold} \\ & f\colon \varphi \to_{FK,GK} \psi \end{array}$$

as required. □

The next theorem shows that each functor is a transformer. Remember that *Exl* and *Exr* denote the extraction (projection) functors from a product category to the component categories respectively.

18 Theorem. *Let $K\colon \mathcal{B} \to \mathcal{C}$ be a functor. Put $X,Y = Exl, Exr$, both being functors of type $\mathcal{B} \times \mathcal{B} \to \mathcal{B}$. Then K is a transformer of type $(X,Y) \to (KX,KY)$.*

Proof. The typing requirement for K is met: taking $\mathcal{A} = \mathcal{B} \times \mathcal{B}$,

$$\begin{array}{ll} & \forall a \text{ in } \mathcal{A},\ \varphi\colon Xa \to_{\mathcal{B}} Ya ::\ \ K\varphi\colon KXa \to_{\mathcal{C}} KYa \\ \equiv & \qquad \text{definition } \mathcal{A} \text{ and } X,Y \\ & \forall b,c \text{ in } \mathcal{B},\ \varphi\colon b \to_{\mathcal{B}} c ::\ \ K\varphi\colon Kb \to_{\mathcal{C}} Kc \\ \equiv & \qquad \text{functoriality of } K \\ & \textsf{true}. \end{array}$$

To check the TRANSFORMER property, we argue:

$$\begin{array}{ll} & f\colon K\varphi \to_{KX,KY} K\psi \\ \Leftarrow & \qquad \text{general theorem (even for arbitrary } X,Y\text{)} \\ & f\colon \varphi \to_{X,Y} \psi. \end{array}$$

It may be instructive to spell out this implication. Observe that a morphism f in $\mathcal{B} \times \mathcal{B}$ has the form $f = (g, h)$ for some morphisms g, h in $\mathcal{B}$. The TRANSFORMER property thus reads

$$(g,h)\colon\ \varphi \rightarrow_{Exl\,,Exr}\ \psi \quad \Rightarrow \quad (g,h)\colon\ K\varphi \rightarrow_{K\,Exl\,,\ K\,Exr}\ K\psi$$

that is,

$$\varphi\ ;\ h = g\ ;\ \psi \quad \Rightarrow \quad K\varphi\ ;\ Kh = Kg\ ;\ K\psi\,.$$

Indeed, this is valid for each functor K. □

From all these theorems we conclude that for all conventional laws there is no need to check TRANSFORMER explicitly: the transformers of such laws are built entirely by the compositions of the theorems. In particular this holds for morphisms and natural transformations like projections, injections, split, junc, product and sum and so on.

19 Naturality of transformers. Before we realised that transformers are natural transformations as explained in paragraph 6 we were looking for naturality properties in the way reported here. As a motivation, notice that a transformer T maps morphisms of type $Fa \rightarrow Ga$ into morphisms of type $Ha \rightarrow Ja$. In a sense, transformers are natural transformations from 'functor' $F \rightarrow G$ to 'functor' $H \rightarrow J$. Let us first make precise what we mean.

The infix written bifunctor

$$\_ \rightarrow \_ \quad : \quad \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{S}et \qquad (\ ^{op} \text{ is explained below})$$

is defined as follows (it is the Hom-functor). For objects a, b and morphisms $f\colon\ a \rightarrow b$ and $g\colon\ c \rightarrow d$,

$$\begin{array}{lcl}(a \rightarrow b) & = & \{x \mid\ x\colon\ a \rightarrow_{\mathcal{C}} b\} \\ (f \rightarrow g) & = & \lambda(x ::\ \ f\ ;\ x\ ;\ g)\ :\ \ (b \rightarrow c) \rightarrow (a \rightarrow d)\,.\end{array}$$

(The interchange of a and b in the type of $(f \rightarrow g)$ means that $\_ \rightarrow \_$ is contravariant in its first argument, indicated by ^{op} in the typing.) Notice that $x\colon\ a \rightarrow b$ equivales $x \in (a \rightarrow_{\mathcal{C}} b)$ (thus justifying our choice of notation). For readability put $X, Y = Exl\,, Exr$. Recall also the convention that $(F \dagger G)x = Fx \dagger Gx$, which we shall use with $\rightarrow$ for $\dagger$.

Consider now a natural transformation $T\colon\ (FX \rightarrow GY) \dot{\rightarrow} (HX \rightarrow JY)$. Working out in detail what naturality means we find

$$\begin{array}{cl} & T\colon\ (FX \rightarrow GY) \dot{\rightarrow} (HX \rightarrow JY) \\ \equiv & \qquad \text{ntrf: for all } (f,g)\colon\ (b,c) \rightarrow_{\mathcal{C}^{op}\times\mathcal{C}} (a,d) \ \ (\text{so } f\colon\ a \rightarrow_{\mathcal{C}} b) \\ & (Ff \rightarrow Gg)\ ;\ T_{ad} = T_{bc}\ ;\ (Hf \rightarrow Jg) \\ \equiv & \qquad \text{extensionality in } \mathcal{S}et\text{: for all } \varphi \in (b \rightarrow c) \\ & ((Ff \rightarrow Gg)\ ;\ T_{ad})\varphi = (T_{bc}\ ;\ (Hf \rightarrow Jg))\varphi \\ \equiv & \qquad \text{application, composition, hom-functor} \end{array}$$

$$T_{ad}(Ff \mathbin{;} \varphi \mathbin{;} Gg) = Hf \mathbin{;} T_{bc}\varphi \mathbin{;} Jg.$$

That is, natural transformation T satisfies a two-sided fusion law. (An adjunction between F and J is nothing but such a natural transformation that has an inverse, so that necessarily $G = J = I$.)

20 Theorem. *Suppose $T\colon (FX \to GY) \dot{\to} (HX \to JY)$. Define mapping T' by $T'\varphi = T_{U\varphi,U\varphi}$. Then T' is a transformer of type $(F,G) \to (H,J)$. The converse is not true, that is, there exist transformers that cannot be written this way.*

Proof. As regards the typing requirement, the statement $\varphi\colon Fa \to Ga$ implies clearly $T'\varphi\colon Ha \to Ja$. It remains to verify the TRANSFORMER property. Let φ, ψ be F,G-dialgebras with carriers a, b respectively, and let $f\colon a \to b$. Then

$$\begin{array}{ll}
 & T'\varphi \mathbin{;} Jf = Hf \mathbin{;} T'\psi \\
\equiv & \qquad \text{definition } T'\text{, identity} \\
 & H id_a \mathbin{;} T_{a,a}\varphi \mathbin{;} Jf = Hf \mathbin{;} T_{b,b}\psi \mathbin{;} J id_b \\
\equiv & \qquad \text{in lhs: naturality } T \text{ with } a,b,c,d,f,g := a,a,a,b,id_a,f\text{,} \\
 & \qquad \text{in rhs: naturality } T \text{ with } a,b,c,d,f,g := a,b,b,b,f,id_b \\
 & T_{a,b}(F id_a \mathbin{;} \varphi \mathbin{;} Gf) = T_{a,b}(Ff \mathbin{;} \psi \mathbin{;} G id_b) \\
\Leftarrow & \qquad \text{Leibniz, identity} \\
 & \varphi \mathbin{;} Gf = Ff \mathbin{;} \psi.
\end{array}$$

To show that the converse is not true, consider arbitrary $\eta\colon H \dot{\to} J$. Then by Theorem 16 η is a transformer of type $(F,G) \to (H,J)$. It is not a natural transformation of type $(FX \to GY) \dot{\to} (HX \to JY)$ since the typing is not correct; this is also apparent from the two-sided fusion law that now simplifies to

$$\eta \quad = \quad Hf \mathbin{;} \eta \mathbin{;} Jg$$

which should hold for each a, b, c, d and $f\colon a \to b$ and $g\colon c \to d$ — clearly impossible in general. For a counterexample, take $\eta = id\colon I \dot{\to} I$. □

21 Conjunction. If E_0 and E_1 are two laws with the same source type, then by 'a conjunction' of E_0 and E_1 we mean a law E such that for all φ: $E\varphi \equiv E_0\varphi \wedge E_1\varphi$. We shall show that there are two ways of representing the conjunction of laws. The two ways yield laws of different target type.

For mappings $T_i\colon (F,G) \to (H_i,J)$ ($i = 0,1$) we define $T_0 \triangledown T_1$ by

$$(T_0 \triangledown T_1)\varphi \quad = \quad T_0\varphi \triangledown T_1\varphi \;:\; H_0a + H_1a \to Ja$$

for any a and $\varphi\colon Fa \to Ga$. It follows by homo-SUM that $T_0 \triangledown T_1$ is a mapping of type $(F,G) \to (H_0{+}H_1, J)$. The composite $S_0 \vartriangle S_1$ is defined similarly, and we have

$$S_0 \vartriangle S_1 \;:\; (F,G) \to (H, J_0{\times}J_1)$$

for S_i: $(F, G) \to (H, J_i)$. Of course we need to assume that the category has sums or products, respectively.

22 Theorem. *Let T_i, T_i': $(F, G) \to (H_i, J)$ be transformers for $i = 0, 1$. Then $T_0 \triangledown T_1$ is a transformer, and*

$$(T_0 \triangledown T_1)\varphi = (T_0' \triangledown T_1')\varphi \quad\equiv\quad T_0\varphi = T_0'\varphi \;\wedge\; T_1\varphi = T_1'\varphi\,.$$

Similarly, for S_i, S_i': $(F, G) \to (H, J_i)$, mapping $S_0 \vartriangle S_1$ is a transformer, and

$$(S_0 \vartriangle S_1)\varphi = (S_0' \vartriangle S_1')\varphi \quad\equiv\quad S_0\varphi = S_0'\varphi \;\wedge\; S_1\varphi = S_1'\varphi\,.$$

Proof. The equivalences follow from the properties for product and sum. For the TRANSFORMER property of $T_0 \triangledown T_1$, let $\varphi \,;\, Gf = Ff \,;\, \psi$. Then

$$\begin{array}{ll}
 & (T_0 \triangledown T_1)\varphi \,;\, Jf \\
= & \\
 & (T_0\varphi \,;\, Jf) \triangledown (T_1\varphi \,;\, Jf) \\
= & \qquad \text{TRANSFORMER } T_0,\ T_1 \\
 & (H_0 f \,;\, T_0\psi) \triangledown (H_1 f \,;\, T_1\psi) \\
= & \\
 & (H_0 + H_1)f \,;\, (T_0 \triangledown T_1)\psi.
\end{array}$$

The proof for $S_0 \vartriangle S_1$ is similar. Law homo-SUM 3.18 and homo-PROD 3.19 express properties very similar to these transformer properties. □

So, if two laws $E_0 = (T_0, T_0')$ and $E_1 = (T_1, T_1')$ have typing T_i, T_i': $(F, I) \to (H_i, I)$, and are used to "identify elements in the carriers" of F-algebras as explained in paragraphs 10 and 11, then $E = (T_0 \triangledown T_1, T_0' \triangledown T_1')$ is a conjunction of such a type that it may be used for the same purpose as E_0 and E_1. The $\vartriangle$-form of the conjunction of laws is to be used if the laws are used to "leave out elements from the carriers" as explained in paragraph 11.

Of course, there are also arbitrary infinite conjunctions of laws, provided the category has arbitrary infinite sums or products.

5d One half of a Birkhoff characterisation

23 Birkhoff characterisation. Let F and H be endofunctors and $E = (T, T')$ be a law of type $(F, I) \to (H, I)$, fixed throughout this section. We define $\mathcal{A}lg(F, E)$ as the full subcategory of $\mathcal{A}lg(F)$ containing all and only those F-algebras for which law E is valid. A "Birkhoff characterisation" is a characterisation of the classes (subcategories) that can be specified by means of a single law. For example, a characterisation might be an equivalence like: for any class $\mathcal{A}$ of F-algebras,

$\mathcal{A} = \mathcal{A}lg(F, E)$ for some law E

if and only if

$\mathcal{A}$ is closed under subalgebras, homomorphisms, and products.

We shall give one half of such an equivalence (the easy half): some closure properties of $\mathcal{A}lg(F,E)$. (I've been unable to prove the converse.) Some care is needed in defining subalgebras and homomorphic images since we wish to work in an arbitrary category, and not just in $\mathcal{S}et$ where several properties hold that are not valid in, say, *CPO*. The notions of subalgebra and homomorphic image that we define are dual to each other.

24 Subalgebra. Given F-algebras φ and ψ, we say φ is a **subalgebra** of ψ if: there exists an $f\colon \varphi \to_F \psi$ which is monic in $\mathcal{C}$. A subcategory $\mathcal{A}$ of $\mathcal{A}lg(F)$ is **closed under subalgebras** if: each subalgebra of an algebra in $\mathcal{A}$ is in $\mathcal{A}$ too. More in spirit with the position that in any category the morphisms are important and the objects play only an auxiliary rôle, we define also another, related, property. $\mathcal{A}$ is **closed under incoming monos** if: f is a morphism in $\mathcal{A}$ whenever $f\colon \varphi \to_F \psi$ is monic in $\mathcal{C}$ and ψ is in $\mathcal{A}$. For a full subcategory of $\mathcal{A}lg(F)$, closure under subalgebras equivales closure under incoming monos.

25 Theorem. *$\mathcal{A}lg(F,E)$ is closed under subalgebras (i.e., under incoming monos).*

Proof. Suppose $f\colon \varphi \to_F \psi$ is monic, and ψ is in $\mathcal{A}lg(F,E)$. We show that $E\varphi$ holds.

$T\varphi = T'\varphi$

$\Leftarrow$ $\quad$ f monic

$T\varphi \,;\, f = T'\varphi \,;\, f$

$\equiv$ $\quad$ TRANSFORMER (condition '$\varphi \,;\, f = Ff \,;\, \psi$' is satisfied) at both sides

$Hf \,;\, T\psi = Hf \,;\, T'\psi$

$\Leftarrow$ $\quad$ Leibniz, and ψ in $\mathcal{A}lg(F,E)$

true.

So φ is in $\mathcal{A}lg(F,E)$ and, since $\mathcal{A}lg(F,E)$ is a full subcategory of $\mathcal{A}lg(F)$, f is a morphism in $\mathcal{A}lg(F,E)$ as well. □

26 Homomorphisms. Consider $f\colon \varphi \to_F \psi$. Working in $\mathcal{S}et$ the homomorphic image of φ under f is the algebra ψ restricted to the range of function f. A generalisation to arbitrary categories is problematic, since categorically there are no points available. Working with *varieties*, as Lehmann [38, 39] does, the corresponding closure property says that with φ also ψ is in the class. That is certainly not true for $\mathcal{A}lg(F,E)$, as we shall argue after the theorem. Our way out is to consider epic homomorphisms. We define: A subcategory $\mathcal{A}$ of $\mathcal{A}lg(F)$ is **closed under homomorphic images** if: ψ is in $\mathcal{A}$ whenever there exists an $f\colon \varphi \to_F \psi$ which is epic in $\mathcal{C}$ and φ is in $\mathcal{A}$. And, $\mathcal{A}$ is **closed under outgoing epis** if: f is a morphism in $\mathcal{A}$ whenever $f\colon \varphi \to_F \psi$ is epic and φ is in $\mathcal{C}$. For a full subcategory of $\mathcal{A}lg(F)$, closure under homomorphic images equivales closure under outgoing epis.

27 Theorem. *Suppose H (from the type of E) preserves epis. Then $\mathcal{A}lg(F,E)$ is closed under homomorphic images (that is, under outgoing epis).*

Proof. Let $f\colon \varphi \to_F \psi$ be epic, with φ in $\mathcal{A}lg(F,E)$.

$$\begin{array}{ll}
 & T\psi = T'\psi \\
\Leftarrow & \qquad Hf \text{ is epic} \\
 & Hf \mathbin{;} T\psi = Hf \mathbin{;} T'\psi \\
\equiv & \qquad \text{TRANSFORMER (condition `}f\colon \varphi \to_F \psi\text{' is satisfied) at both sides} \\
 & T\varphi \mathbin{;} f = T'\varphi \mathbin{;} f \\
\equiv & \qquad \text{assumption: } \varphi \text{ in } \mathcal{A}lg(F,E) \\
 & \textsf{true}.
\end{array}$$

So ψ is in $\mathcal{A}lg(F,E)$, and since $\mathcal{A}lg(F,E)$ is a full subcategory of $\mathcal{A}lg(F)$, f is a morphism in $\mathcal{A}lg(F,E)$. □

For later use we mention the following result; its proof is part of the above one.

28 Lemma. *Let φ in $\mathcal{A}lg(F,E)$, and $f\colon \varphi \to_F \psi$. Then $E\psi$ holds "on the range of f", that is, $Hf \mathbin{;} T\psi = Hf \mathbin{;} T'\psi$.*

The requirement that a functor (like H in the theorem) preserves epis, is a mild one. In $\mathcal{S}et$ all polynomial functors preserve epis. Lehmann [39] argues that preservation of epis is an important property.

29 Homomorphisms do not preserve laws. It is now clear why homomorphisms do not preserve the validity of laws: outside the range of the homomorphism nothing can be inferred for the target algebra. (This may be a good reason to work with the variety $V_F(E\mu F)$ instead of $\mathcal{A}lg(F,E)$, see Definition 34 and Theorem 35.) For example, imagine in $\mathcal{S}et$ the algebra $zero \triangledown add\colon \imath + \mathbb{I}\, nat \to nat$ of finite naturals, where $zero$ is a neutral element for add. Form another algebra by adjoining a fictitious element ω to nat, and extend operation add as follows: for any natural x, $add'(x,\omega) = add'(\omega,x) = x$ and $add'(\omega,\omega) = \omega$. The injection of the original algebra into this new one is a homomorphism, but in the new algebra $zero$ is no longer neutral for add'.

30 Product. For F-algebras φ, ψ (with carriers a and b say), and F-homomorphisms f, g with common source in $\mathcal{A}lg(F)$ we define

$$\begin{array}{lcll}
\varphi \times_F \psi & = & (F\,exl \mathbin{;} \varphi) \vartriangle (F\,exr \mathbin{;} \psi) & : \quad F(a \times b) \to (a \times b) \\
f \vartriangle_F g & = & f \vartriangle g & \\
exl_F, exr_F & = & exl, exr\,. &
\end{array}$$

It is then readily verified that $\times_F$, $\vartriangle_F$, exl_F, exr_F form a categorical product in $\mathcal{A}lg(F)$, and we omit the subscript F in these operations since no confusion can result. In particular, exl: $\varphi \times \psi \to_F \varphi$, that is,

$$(*) \qquad \varphi \times \psi \,;\, exl \;=\; F\,exl \,;\, \varphi\,,$$

and similarly for exr. The generalisation to arbitrary products is straightforward; the proviso being that the default category has arbitrary products.

31 Theorem. *$\mathcal{A}lg(F, E)$ is closed under products.*

Proof. We consider only binary products.

$$\begin{array}{ll}
 & T(\varphi \times \psi) = T'(\varphi \times \psi) \\
\Leftarrow & \qquad \text{the projections are jointly monic in } \mathcal{C} \\
 & T(\varphi \times \psi) \,;\, exl = T'(\varphi \times \psi) \,;\, exl \qquad \text{and} \\
 & T(\varphi \times \psi) \,;\, exr = T'(\varphi \times \psi) \,;\, exr \\
\equiv & \qquad \text{TRANSFORMER at both sides (condition is satisfied: see } (*) \text{ above)} \\
 & H\,exl \,;\, T\varphi = H\,exl \,;\, T'\varphi \qquad \text{and} \\
 & H\,exr \,;\, T\psi = H\,exr \,;\, T'\psi \\
\Leftarrow & \qquad \text{Leibniz, } \varphi, \psi \text{ in } \mathcal{A}lg(F, E) \\
 & \mathsf{true}.
\end{array}$$

□

5e Initial algebras with laws

Let F be an endofunctor for which the initial algebra $\alpha = \mu F$ exists. Let E be a law of type $(F, I) \to (H, I)$ for some endofunctor H. As before, $\mathcal{A}lg(F, E)$ is the full subcategory of $\mathcal{A}lg(F)$ of algebras for which E holds. We are interested in an algebra that is initial in $\mathcal{A}lg(F, E)$; we shall denote it by $\mu(F, E)$.

32 Example (Trees continued) Take E to be the law such that $E(e \triangledown f \triangledown \oplus)$ expresses both the neutrality of e for $\oplus$, and the associativity of $\oplus$. Then $\mu(F, E)$, if it exists, is the algebra of join lists. Put $nil' \triangledown tip' \triangledown join' = \mu(F, E)$ and let $e \triangledown f \triangledown \oplus$ be another (F, E)-algebra. Now, by definition of initiality, the recursive equations

$$\begin{array}{lcl}
nil' \,;\, h & = & e \\
tip' \,;\, h & = & f \\
join' \,;\, h & = & I\!I h \,;\, \oplus
\end{array}$$

have precisely one solution for h, denoted $(\!| nil' \triangledown tip' \triangledown join' \to e \triangledown f \triangledown \oplus |\!)_{F,E}$. Actually, the equations imply that e is neutral for $\oplus$, and $\oplus$ is associative, at least on the range of h: see Lemma 28. □

33 Induced congruence. We explain here informally in terms of $\mathcal{S}et$ the notion of induced congruence; in Section 2f we have given a discussion in category speak. Let φ be an F-algebra with carrier a, and f,g be morphisms with target a and common source; think in particular of $\varphi,(f,g) = \alpha,(T\alpha,T'\alpha)$. The pair (f,g) induces an equivalence relation p on a, namely the least equivalence relation on a that contains all (fx,gx); categorically, this is a morphism p with $f \,;\, p = g \,;\, p$ that has some initiality property. The target of p may be denoted $a/(f,g)$, thus $p\colon a \to a/(f,g)$.

We say that an equivalence relation p for (f,g) is a congruence for φ if p-related elements are mapped by φ to p-related results; this is almost the same as saying that p is a homomorphism from φ. Given φ and (f,g), the induced congruence is the least equivalence relation that contains all pairs (fx,gx) and is a congruence for φ; the target of p may be denoted $\varphi/(f,g)$, thus $p\colon \varphi \to_F \varphi/(f,g)$.

Thus, when α and E are given, a construction of $\mu(F,E) = \alpha/(T\alpha,T'\alpha)$ requires a construction of the congruence p for algebra α induced by $(T\alpha,T'\alpha)$. Our construction of p in Section 2f simulates the well-known one for $\mathcal{S}et$, and assumes properties of the default category that are not obviously satisfied by category CPO. Since we are also interested in applications to CPO, see Chapter 6, and want to be truly general, we shall use a result from Lehmann [38, 39].

34 Definition. *For a pair (f,g) of morphisms with common source and with the carrier of α as their common target, the (f,g)-*variety *$V_F(f,g)$ is the full subcategory of $\mathcal{A}lg(F)$ of algebras φ for which $f \,;\, (\!|\varphi|\!) = g \,;\, (\!|\varphi|\!)$.*

Notice that $\varphi \mapsto f \,;\, (\!|\varphi|\!)$ is a transformer, so that $V_F(f,g)$ equals $\mathcal{A}lg(F,E')$ where E' is the law determined by the transformers $\varphi \mapsto f \,;\, (\!|\varphi|\!)$ and $\varphi \mapsto g \,;\, (\!|\varphi|\!)$.

35 Theorem (Lehmann [39]) *For $\mathcal{C} = \mathcal{S}et$ and for each $\mathcal{C}$ that satisfies the conditions of Lehmann's theorem, such as CPO and several other order-enriched categories, and assuming that F preserves epis, any variety $V_F(f,g)$ has an initial object, denoted $\alpha/(f,g)$. Moreover $(\!|\alpha/(f,g)|\!)$ is epic in $\mathcal{C}$.*

The following result enables us to exploit Lehmann's theorem.

36 Theorem. *Suppose that H (of the type of E) preserves epis. Then $\mathcal{A}lg(F,E)$ is a full subcategory of $V_F(E\alpha)$, and contains each φ of $V_F(E\alpha)$ for which $(\!|\varphi|\!)$ is epic in $\mathcal{C}$.*

Proof. Since both $\mathcal{A}lg(F,E)$ and $V_F(E\alpha)$ are full subcategories of $\mathcal{A}lg(F)$, we have to show, for the first claim, that each φ in $\mathcal{A}lg(F,E)$ is also in $V_F(E\alpha)$. This implication is shown as follows. Let φ be arbitrary in $\mathcal{A}lg(F)$. Then

φ is in $V_F(E\alpha)$

$\equiv$ definition 34 of $V_F(E\alpha)$, and φ is in $\mathcal{A}lg(F)$

$T\alpha \,;\, (\!|\varphi|\!) = T'\alpha \,;\, (\!|\varphi|\!)$

$\equiv$ TRANSFORMER at both sides (condition is satisfied: $(\!|\varphi|\!)\colon \alpha \to_F \varphi$)

$$\begin{array}{lll}
 & & H\llparenthesis\varphi\rrparenthesis \,;\, T\varphi = H\llparenthesis\varphi\rrparenthesis \,;\, T'\varphi \\
(*) & \Leftarrow & \quad \text{Leibniz} \\
 & & T\varphi = T'\varphi \\
 & \equiv & \quad \text{definition of } \mathcal{A}lg(F,E)\text{, and } \varphi \text{ is in } \mathcal{A}lg(F) \\
 & & \varphi \text{ is in } \mathcal{A}lg(F,E).
\end{array}$$

For the second claim, let φ be in $V_F(E\alpha)$ such that $\llparenthesis\varphi\rrparenthesis$ is epic in $\mathcal{C}$. Then the $\Leftarrow$ in step $(*)$ above can be strengthened to $\equiv$ since $\llparenthesis\varphi\rrparenthesis$ is epic and H is assumed to preserve epis. Hence φ is in $\mathcal{A}lg(F,E)$ as well. □

37 Corollary. *Suppose both F and H preserve epis, and $\mathcal{C}$ satisfies the conditions of Lehmann's theorem [39], see 35. Then $\mathcal{A}lg(F,E)$ has an initial object, denoted $\mu(F,E)$.*

Proof. By Theorem 35 $V_F(E\alpha)$ has an initial object $\alpha/E\alpha$, and $\llparenthesis\alpha/E\alpha\rrparenthesis$ is epic in $\mathcal{C}$. (Here the conditions on F and $\mathcal{C}$ are used.) Then by Theorem 36 $\alpha/E\alpha$ is also in $\mathcal{A}lg(F,E)$, and moreover it is also initial in $\mathcal{A}lg(F,E)$. (Here it is used that H preserves epis.) □

* * *

Although the following is independent of the precise nature of laws and transformers, we cannot resist the temptation to include it. The theorem is useful for the development of efficient programs, as we shall explain afterwards. We put $\alpha = \mu F$ and $\beta = \mu(F,E)$, and write $\llparenthesis\alpha \to \varphi\rrparenthesis_F$ as $\llparenthesis\varphi\rrparenthesis$, and $\llparenthesis\beta \to \varphi\rrparenthesis_{F,E}$ as $\llparenthesis\varphi\rrparenthesis_E$.

38 Theorem (Meertens) *Suppose (α and also) β exists, and suppose $\llparenthesis\beta\rrparenthesis$ has a pre-inverse u. Then*

$$\begin{array}{lcl}
\llparenthesis\varphi\rrparenthesis_E & = & u \,;\, \llparenthesis\varphi\rrparenthesis \\
\llparenthesis\varphi\rrparenthesis & = & \llparenthesis\beta\rrparenthesis \,;\, u \,;\, \llparenthesis\varphi\rrparenthesis
\end{array}$$

for each φ in $\mathcal{A}lg(F,E)$.

Proof. For the first claim we argue

$$\begin{array}{ll}
 & \llparenthesis\varphi\rrparenthesis_E \;=\; u \,;\, \llparenthesis\varphi\rrparenthesis \\
\equiv & \quad u \text{ is pre-inverse of } \llparenthesis\beta\rrparenthesis \\
 & u \,;\, \llparenthesis\beta\rrparenthesis \,;\, \llparenthesis\varphi\rrparenthesis_E \;=\; u \,;\, \llparenthesis\varphi\rrparenthesis \\
\Leftarrow & \quad \text{Leibniz} \\
 & \llparenthesis\beta\rrparenthesis \,;\, \llparenthesis\varphi\rrparenthesis_E \;=\; \llparenthesis\varphi\rrparenthesis \\
\Leftarrow & \quad \text{cata-}\textsc{Fusion} \\
 & \llparenthesis\varphi\rrparenthesis_E \colon \beta \to_F \varphi \\
\equiv & \quad \text{cata-}\textsc{Self} \\
 & \textsf{true}.
\end{array}$$

The second claim is an immediate corollary:

$$\begin{array}{ll} & (\!|\beta|\!) \,;\, u \,;\, (\!|\varphi|\!) \\ = & \quad \text{just shown} \\ & (\!|\beta|\!) \,;\, (\!|\varphi|\!)_E \\ = & \quad \text{cata-}\textbf{FUSION}\text{ (condition is satisfied: } (\!|\varphi|\!)_E\text{: } \beta \rightarrow_F \varphi\text{)} \\ & (\!|\varphi|\!). \end{array}$$

The existence of β (hence the well definedness of $(\!|\_|\!)_E$) is guaranteed by the previous theorems if F and H preserve epis. □

39 Application. In $\mathcal{S}et$, the carrier of β consists of the $\simeq$-equivalence classes of the carrier of α , where $\simeq$ is the least equivalence that contains the pairs $((T\alpha)z, (T'\alpha)z)$ for all z . A pre-inverse u of $(\!|\alpha \rightarrow \beta|\!)$ chooses for each equivalence class a representative in the class. So the theorem says that $(\!|\beta \rightarrow \varphi|\!)_E$ at x may be computed by computing $(\!|\alpha \rightarrow \varphi|\!)$ at a representative of x instead. In this way the operational efficiency of a program may be improved.

40 Example (Trees continued) Let $E(e \triangledown id \triangledown \oplus)$ express that $\oplus$ is an associative operation with neutral element e, and suppose that the law holds for $e \triangledown id \triangledown \oplus$. The value of $(\!|e \triangledown id \triangledown \oplus|\!)_E$ at arguments $x\,\text{join}'\,(y\,\text{join}'\,z)$ and $(x\,\text{join}'\,y)\,\text{join}'\,z$ is

$$\begin{array}{lll} (\!|e \triangledown id \triangledown \oplus|\!)_E(x\,\text{join}'\,(y\,\text{join}'\,z)) & = & x \oplus (y \oplus z) \\ (\!|e \triangledown id \triangledown \oplus|\!)_E((x\,\text{join}'\,y)\,\text{join}'\,z) & = & (x \oplus y) \oplus z\,. \end{array}$$

Due to associativity both results are the same, yet the computations as suggested by the right hand sides may differ operationally. For example, the first alternative is more efficient if $x \oplus y$ takes time linear in $size\,x$, and $size\,(x \oplus y) = size\,x + size\,y$. (This is valid in most functional programming languages for $\oplus$ equal to the concatenation of lists.) Thus associativity may be exploited. More generally, let u be the function that sends $x\,\text{join}'\,y\,\text{join}' \ldots \text{join}'\,z$ (with arbitrary parenthesisation) to $x\,\text{join}\,(y\,\text{join}\,(\ldots\text{join}\,z))$ (with parenthesation to the right). The theorem asserts that $(\!|e \triangledown id \triangledown \oplus|\!)_E = u \,;\, (\!|e \triangledown id \triangledown \oplus|\!)$, and by the argument above we know that the catamorphism in the right hand side is more efficient than that in the left hand side. (It is quite easy to express u explicitly. In an actual program transformation u might disappear completely, namely when this transformation is but one step in a large series of steps.) □

41 Another application. In a similar way the second claim of the theorem asserts that if φ satisfies E, then "within the argument" of $(\!|\varphi|\!)$ α may be manipulated as if it satisfies E, that is, $T\alpha \,;\, (\!|\varphi|\!) = T'\alpha \,;\, (\!|\varphi|\!)$. This is shown as follows.

$$\begin{array}{ll} & T\alpha \,;\, (\!|\varphi|\!) = T'\alpha \,;\, (\!|\varphi|\!) \\ \equiv & \quad \text{second claim of the theorem: } (\!|\varphi|\!) = (\!|\beta|\!) \,;\, u \,;\, (\!|\varphi|\!) \end{array}$$

$T\alpha \,;\, (\!|\beta|\!) \,;\, u \,;\, (\!|\varphi|\!) = T'\alpha \,;\, (\!|\beta|\!) \,;\, u \,;\, (\!|\varphi|\!)$

$\Leftarrow$ Leibniz

$T\alpha \,;\, (\!|\beta|\!) = T'\alpha \,;\, (\!|\beta|\!)$

$\equiv$ TRANSFORMER (condition is satisfied: $(\!|\beta|\!)$: $\alpha \rightarrow_F \beta$) at both sides

$H(\!|\beta|\!) \,;\, T\beta = H(\!|\beta|\!) \,;\, T'\beta$

$\equiv$ Leibniz, $E\beta$ holds

true.

5f Each colimit is an initial lawful algebra

Lambert Meertens has made the following observation. For an arbitrary colimit we can construct an endofunctor F and a law E such that (the "∇" of) the colimit is an initial (F, E)-algebra, provided the category has arbitrary sums. This is further evidence for the expressiveness of our notion of law. We shall first perform the construction for coequalisers, and then for colimits in general.

42 Coequalisers. Let f, g be a parallel pair with target a, and let p be a coequaliser of f, g. This means, by definition, that $f \,;\, p = g \,;\, p$ and for each q with $f \,;\, q = g \,;\, q$ there exists a morphism, which we denote $p\backslash q$, such that

$$p \,;\, x = q \quad \equiv \quad x = p\backslash q \qquad\qquad \text{coequaliser-CHARN}$$

Now take $F = \underline{a}$, the constant functor mapping any morphism onto id_a. Take

$$E \;=\; (T, T') \qquad \text{with} \qquad Tq \;=\; f \,;\, q \qquad \text{and} \qquad T'q \;=\; g \,;\, q$$

for each q: $Fb \rightarrow b \;=\; a \rightarrow b$. Then, in the notation of Theorem 16, $T = \underline{f}; I$ and similarly $T' = \underline{g}; I$, and so E is a law (by that same theorem). Further, Ep holds, and p: $F(\text{tgt}\, p) \rightarrow \text{tgt}\, p$. So p is an (F, E)-algebra. To show the initiality of p we shall prove cata-CHARN, deriving along the way a definition for the required $(\!|p \rightharpoonup q|\!)$.

x: $p \rightarrow_F q$

$\equiv$ definition $\rightarrow_F$ and $F = \underline{a}$

$p \,;\, x = q$

$\equiv$ coequaliser-CHARN, noting that $f \,;\, q = g \,;\, q$

$x = p\backslash q$

$\equiv$ **defining** $(\!|p \rightharpoonup q|\!) = p\backslash q$

$x = (\!|p \rightharpoonup q|\!)$.

43 Colimits. We generalise the above construction to arbitrary colimits. The p and q above become cocones γ, δ or algebras γ', δ' below, the f and g become (the arrows in) the diagram D, and law E is going to express "the commutativity of all triangles". First we give a formalisation of colimit that suits the present purpose well.

Let D be a diagram in $\mathcal{C}$. A cocone for D is a family $\delta = (a \text{ in } D :: \delta_a)$ such that

(a) $\qquad \forall f\colon a \to b \text{ in } D :: \quad \delta_a = f \mathbin{;} \delta_b .$

A cocone γ is a colimit for D if for any cocone δ for D there exists a morphism, which we denote $\gamma\backslash\delta$, such that

(b) $\qquad \forall(a \text{ in } D :: \quad \gamma_a \mathbin{;} x = \delta_a) \quad \equiv \quad x = \gamma\backslash\delta$ $\qquad$ colimit-CHARN

44 The construction. Take $F = \underline{\Sigma D}$, where $\Sigma D =$ (the carrier of) the sum of all objects in D. Similarly to the Trees example each F-algebra $\delta'\colon \Sigma D \to d$ can be written as $\delta' = \nabla(a \text{ in } D :: in_a \mathbin{;} \delta')$. We design E such that $E\delta$ equivales (a) above:

$$E \quad = \quad \text{the conjunction of the laws } (T_a, T'_{f,b}) \text{ for all } f\colon a \to b \text{ in } D$$

where

$$\begin{aligned} T_a\delta' &= in_a \mathbin{;} \delta' \\ T'_{f,b}\delta' &= f \mathbin{;} in_b \mathbin{;} \delta' . \end{aligned}$$

Indeed, $\delta' = \nabla(a :: \delta_a)$ is an (F, E)-algebra iff $\delta = (a :: \delta_a)$ is a cocone for D. Moreover, by Theorem 16 T_a and $T'_{f,b}$ are transformers, so that $(T_a, T'_{f,b})$ is a law, and by Theorem 22 the conjunction E can be expressed as a law.

Let $\gamma = (a :: \gamma_a)$ be a colimit for D. We claim that $\gamma' = \nabla(a :: \gamma_a)$ is initial in $\mathcal{A}lg(F, E)$. To show this let $\delta' = \nabla(a :: \delta_a)$ be an arbitrary (F, E)-algebra. Then, as argued above, $\delta = (a :: \delta_a)$ is a cocone for D, so $\gamma\backslash\delta$ satisfying (b) exists. It is now readily shown that $\gamma\backslash\delta$ taken as $(\!|\gamma' \to \delta'|\!)$ meets the requirement of cata-CHARN:

$$\begin{aligned}
& x\colon \gamma' \to_F \delta' \\
\equiv & \qquad \text{definition } \to_F \\
& \gamma' \mathbin{;} x = Fx \mathbin{;} \delta' \\
\equiv & \qquad \text{definition } F \text{ as a constant functor} \\
& \gamma' \mathbin{;} x = \delta' \\
\equiv & \qquad \text{definition } \gamma' \text{ and } \delta' \text{ and sum} \\
& \forall a \text{ in } D :: \quad \gamma_a \mathbin{;} x = \delta_a \\
\equiv & \qquad \text{colimit-CHARN: (b) above} \\
& x = \gamma\backslash\delta \\
\equiv & \qquad \text{definition `} (\!|\gamma' \to \delta'|\!) \text{'} \\
& x = (\!|\gamma' \to \delta'|\!).
\end{aligned}$$

So γ' is initial indeed.

5g Equational specification of datatypes

45 Datatype of stacks. In paragraph 3.77 we have shown that a 'datatype' like the usual *stack* is a bialgebra (φ, ψ) (which in turn is a particular dialgebra). To be specific, for *stack* we have

$$\begin{array}{lllllll} \varphi & = & \mathit{empty} \triangledown \mathit{push} & : & 1 + a \times b \to b & = & Fb \to b \\ \psi & = & \mathit{isempty} \vartriangle \mathit{top} \vartriangle \mathit{pop} & : & b \to \mathit{bool} \times a \times b & = & b \to Gb \end{array}$$

where a is the type of the stacked values and b is the type of the stacks themselves, and apparently $F = 1 + \underline{a} \times I$ and $G = \underline{\mathit{bool}} \times \underline{a} \times I$. Often for such 'datatypes' some law E is imposed that "defines" the ψ-part in terms of the φ-part. For *stack* the laws are

$$\begin{array}{lll} \mathit{empty} \,;\, \mathit{isempty} & = & \mathit{true} \\ \mathit{push} \,;\, \mathit{isempty} & = & \mathit{false} \\ \mathit{push} \,;\, \mathit{top} & = & \mathit{exl} \\ \mathit{push} \,;\, \mathit{pop} & = & \mathit{exr}\,. \end{array}$$

Written as two equations:

$$\begin{array}{lll} \mathit{empty} \,;\, \psi & = & id_1 \,;\, \mathit{true} \vartriangle \ldots \vartriangle \ldots \\ \mathit{push} \,;\, \psi & = & id_a \times \psi \,;\, (! \,;\, \mathit{false}) \vartriangle \mathit{exl} \vartriangle (\mathit{exr} \,;\, \mathit{ex}_{3,1} \vartriangle \mathit{ex}_{3,2} \,;\, \mathit{push}) \end{array}$$

where on the dots there have to be expressions of type $1 \to a$ and $1 \to b$ respectively, defining the top and pop of an empty stack. (It is outside the scope of our current interest to discuss this aspect in detail.) We can even combine the two equations into one, thus obtaining a law

$$E(\varphi, \psi) \quad = \quad \text{“}\ \varphi \,;\, \psi \ = \ F\psi \,;\, T\varphi\ \text{”}$$

for some transformer T of type $(F, I) \to (FG, G)$. Theorem 46 below asserts that for a law of this form, with arbitrary transformer T, the 'datatype' (initial bialgebra) (φ, ψ) is isomorphic to the initial F-algebra (the φ-part) to which $(\![T\varphi]\!)$ (the ψ-part) is added as a derived operation. Since for the F above the initial F-algebra is known as the cons lists over a, we find that the datatype *stack* is semantically just the algebra of cons lists with some additional derived operations, "destructors" in this case.

Notation. Category $\mathcal{B}iAlg(F, G; E)$ is the full subcategory of $\mathcal{B}iAlg(F, G)$ of those bialgebras that satisfy law E.

46 Theorem. *Let T be a transformer of type $(F, I) \to (FG, G)$, and suppose that $\alpha = \mu F$ exists. Let E be the law suggested by*

$$E(\varphi, \psi) \quad = \quad \text{“}\ \varphi \,;\, \psi \ = \ F\psi \,;\, T\varphi\ \text{”} \qquad \text{for } F, G\text{-bialgebra } (\varphi, \psi)\,.$$

Then $(\alpha, (\![T\alpha]\!))$ is initial in $\mathcal{B}iAlg(F, G; E)$.

Proof. (Observe that law E is well-formed; the type of both sides of the equation is $Fa \to Ga$ where a is the carrier of the argument.) Let (φ, ψ) be a F,G-bialgebra for which E holds. We shall show that

$$x\colon\ (\alpha, (\![T\alpha]\!)) \ \to_{\mathcal{B}i\mathcal{A}lg(F,G)}\ (\varphi,\psi) \qquad \equiv \qquad x = (\![\varphi]\!)$$

thus establishing the existence and uniqueness of an F,G-bi-homomorphism, namely $(\![\varphi]\!)$, from $(\alpha, (\![T\alpha]\!))$ to (φ, ψ).

$$\begin{array}{lll}
 & & x\colon\ (\alpha, (\![T\alpha]\!)) \ \to_{\mathcal{B}i\mathcal{A}lg(F,G)}\ (\varphi,\psi) \\
 & \equiv & \quad \text{definition } \mathcal{B}i\mathcal{A}lg(F,G) \\
 & & x\colon \alpha \to_F \varphi \ \wedge\ x\colon (\![T\alpha]\!) \to_{G,I} \psi \\
 & \equiv & \quad \text{cata-CHARN} \\
 & & x = (\![\varphi]\!) \ \wedge\ (\![\varphi]\!)\colon (\![T\alpha]\!) \to_{G,I} \psi \\
(*) & \equiv & \quad \text{below} \\
 & & x = (\![\varphi]\!).
\end{array}$$

It remains to justify step $(*)$. For this we argue

$$\begin{array}{ll}
 & (\![\varphi]\!)\colon (\![T\alpha]\!) \to_{G,I} \psi \\
\equiv & \quad \text{definition } \to_{G,I} \\
 & (\![T\alpha]\!) \,;\, G(\![\varphi]\!) = (\![\varphi]\!) \,;\, \psi \\
\equiv & \quad \text{rhs: cata-FUSION (condition `}\psi\colon \varphi \to_F T\varphi\text{' follows from } E(\varphi,\psi)\text{)} \\
 & (\![T\alpha]\!) \,;\, G(\![\varphi]\!) = (\![T\varphi]\!) \\
\Leftarrow & \quad \text{cata-FUSION} \\
 & T\alpha \,;\, G(\![\varphi]\!) = FG(\![\varphi]\!) \,;\, T\varphi \\
\Leftarrow & \quad \text{TRANSFORMER} \\
 & \alpha \,;\, (\![\varphi]\!) = F(\![\varphi]\!) \,;\, \varphi \\
\equiv & \quad \text{cata-SELF} \\
 & \mathsf{true}.
\end{array}$$

□

Similarly one may specify an $F+G$-algebra $\varphi \triangledown \psi$ by forcing the ψ-part to be determined by the φ-part. In this case ψ is an additional derived operation that is a "constructor", like φ.

47 Theorem. *Let T be a transformer of type $(F,I) \to (G,I)$, and suppose that $\alpha = \mu F$ exists. Let E be the law suggested by*

$$E(\varphi \triangledown \psi) \quad \equiv \quad \text{“}\ \psi \ = \ T\varphi\ \text{”}.$$

Then $\alpha \triangledown T\alpha$ is initial in $\mathcal{A}lg(F+G; E)$.

Proof. We show initiality of $\alpha \triangledown T\alpha$ by establishing cata-CHARN. Let $\varphi \triangledown \psi$ be an arbitrary $F+G$-algebra for which E holds. Then

$$\begin{array}{lll}
 & & x\colon\ \alpha \triangledown T\alpha \ \ \dot{\to}_{F+G} \ \ \varphi \triangledown \psi \\
 & \equiv & \quad \text{definition } \dot{\to}_{F+G} \\
 & & x\colon\ \alpha \dot{\to}_F \varphi \ \ \wedge \ \ x\colon\ T\alpha \dot{\to}_G \psi \\
 & \equiv & \quad \text{cata-CHARN} \\
 & & x = (\!|\varphi|\!) \ \ \wedge \ \ (\!|\varphi|\!)\colon\ T\alpha \dot{\to}_G \psi \\
(*) & \equiv & \quad \text{below} \\
 & & x = (\!|\varphi|\!).
\end{array}$$

Step $(*)$ is verified as follows.

$$\begin{array}{ll}
 & (\!|\varphi|\!)\colon\ T\alpha \dot{\to}_G \psi \\
\equiv & \quad \text{law } E \text{ holds for } \varphi \triangledown \psi \\
 & (\!|\varphi|\!)\colon\ T\alpha \dot{\to}_G T\varphi \\
\Leftarrow & \quad \text{TRANSFORMER} \\
 & (\!|\varphi|\!)\colon\ \alpha \dot{\to}_F \varphi \\
\equiv & \quad \text{cata-SELF} \\
 & \textsf{true}.
\end{array}$$

□

It is straightforward to combine both theorems, and generalise to the case of triples (φ, ψ, χ) where φ is an F,G-dialgebra, ψ is an H-algebra, and χ is a J-co-algebra, all with the same carrier, and ψ, χ being determined in terms of φ by means of a law.

5h Conclusion

We have proposed a semantical, categorical, characterisation of what a term (as used in conditional equations) is: the TRANSFORMER property. The property is almost as simple as the defining property of functor, and a mapping that satisfies the TRANSFORMER property is called 'transformer'. The reasonability of the proposal has been shown by various theorems on the expressiveness of transformers. The simplicity of various proofs dealing with laws is further evidence of the success of the notion of transformer.

The notion of transformer seems to allow for a great simplification of the theory of equational specification of datatypes as far as only semantic aspects are concerned. Compare for example the exposition in Section 5g with current literature on 'equational specification of datatypes' such as Ehrig and Mahr's book [17]. Our formalism is entirely directed to the semantics (of algebras, or datatypes), whereas signatures and other syntactic aspects are prominently present in Ehrig and Mahr's formalism. As a result, even in the discussions of purely semantic aspects they are forced to take into account irrelevant aspects like scope rules —appearing in the decision to incorporate a parameter algebra into the result algebra— and sharing of implementations —appearing in the notion of persistency— and so on. This gives a lot of unnecessary junk and confusion, and such a treatise is in no way initial. The use of transformer avoids the introduction of non-semantic aspects. Much more in this area can be done.

Thanks to the formalisation of the notion of law, one can now formulate conjectures, statements and proofs about them in general. For example, since long there was a feeling that so-called lifting preserves the validity of all "algebraic" laws; for instance, a lifted commutative operation is itself commutative as well. Recently, Meertens and Van der Woude have been able to formally prove this conjecture — using the notion of transformer.

Chapter 6

Order-enriched categories

An order-enriched category is a category each of whose morphism sets is a pointed cpo, so that monotone mappings on the morphisms have a –unique– least solution. Examples are *CPO* and $CPO_\perp$, the latter containing only the *strict* morphisms of the former. For a wide class of functors F an F-algebra exists that is initial with respect to the strict morphisms, and in general not with respect to the nonstrict morphisms. For such functors the catamorphism constructor "$(\!|\ |\!)$" can be extended to nonstrict morphisms while still satisfying catamorphism-like properties.

In an order-enriched category the inverse of an initial F-algebra is a final F-co-algebra. This fact enables a formalisation of an important programming paradigm, the use of virtual datastructures, that is problematic to describe otherwise.

This chapter reports about joint work with Erik Meijer [24] and Erik Meijer and Ross Paterson [51]. Since a lot of the program calculation laws will be proved and used by Erik Meijer in his thesis [50], we set out to discuss the theoretical foundations and some pragmatic issues only. We call an equation of the form $x = \mathcal{F}x$ a fixed point equation (since each solution of the equation is a fixed point of $\mathcal{F}$, and conversely).

6a The main theorem

1 Fixed point equations. In the preceding chapters we have been dealing with equations having precisely one solution: the cata-, ana-, mutu-, para-, prepro-, or postpromorphism. There were no particular properties required of the underlying category. (Initial algebras exist for polynomial functors if the category is an ω-category.) In this chapter we investigate the consequences of working in a category where fixed point equations

$$x \quad = \quad \mathcal{F}x$$

do have a –unique– least solution for monotone $\mathcal{F}$.

Considering fixed points is readily motivated. First, in practice many algorithms are expressed in such a form, that is, with arbitrary recursion and not only the kind of induction provided by the cata-, ana-, etc. equations. In fact, almost all programming languages explicitly allow an equation $x = \mathcal{F}x$ as a definition of x (namely, defining x as the least fixed point of $\mathcal{F}$). Secondly, *if* we wish that the formalism to express algorithms has universal computing power, *then* more general fixed points should be expressible, since the lambda calculus and any other formalism for computable functions has a way to express fixed points too.

We hasten to say that many algorithms that nowadays seem to require arbitrary recursion are as well expressible by a more restricted form of recursion, namely by an equation that has provably a unique solution. I conjecture that several of the fixed point equations (recursive functions) discussed by Sijtsma [67] do have a unique solution. The prepro-equations in Section 4d originated from this very conjecture.

2 Order-enriched categories. There is a vast amount of literature presenting the theory for least fixed points, mostly in the setting of Denotational Semantics; see for example Schmidt [66] and the references he gives. Basically, in a pointed cpo (ω-complete partial order with a least element $\perp$) each monotone function has a least fixed point, and for a ω-continuous function there is an effective way to compute or approximate the least fixed point, namely by ω-repeated unfolding. (This generalises to other kinds of completeness and continuity.) A category for which the set of morphisms from a to b is a pointed cpo, for each a and b, is called an O-category (**order-enriched**). Moreover, we call the category an $O\perp$-category if, in addition, each least morphism $\perp_{a,b}\colon a \to b$ is a post-zero of composition, that is, $f \,;\, \perp_{b,c} = \perp_{a,c}$ for all $f\colon a \to b$ and c.

From now on we adhere to the convention that

$\mathcal{C}$ ranges over $O\perp$-categories, and is the default category.

Category CPO is the prime example of an $O\perp$-category; its objects are complete partially ordered sets with a least element $\perp$, and the morphisms are the continuous functions between the objects. The pointed-cpo structure on the morphisms is induced by that of the objects: $f \sqsubseteq g \;\equiv\; \forall(x :: \;\; fx \sqsubseteq gx)$, and $\perp_{a,b} = \lambda(x \in a :: \;\; \perp_b)$. Order-enriched categories have been studied extensively by Wand [75], Smyth and Plotkin [68], and Bos and Hemerik [12]. Pierce [60] gives an overview of some of the results.

For later use we also define $\mathcal{C}\perp$; it is the subcategory of $\mathcal{C}$ that has the same objects as $\mathcal{C}$ and as morphisms only the strict morphisms of $\mathcal{C}$. A morphism $f\colon b \to c$ is **strict** if: $\perp_{a,b} \,;\, f = \perp_{a,c}$ for all a. For CPO this coincides with the usual notion of strictness, namely $f(\perp_b) = \perp_c$; the proof is easy and omitted. Clearly, $\mathcal{C}\perp$ is an $O\perp$-category if $\mathcal{C}$ is. Notice that

the categorical dual of a strictness assertion is vacuously true, since the equation $\bot ; f = \bot$ dualises to $f ; \bot = \bot$, and this latter equation is true on account of the definition of $O\bot$-category above.

Finally, a function (morphism mapping of a functor) is **locally continuous** if: for all a, b the restriction of the function to the set of morphisms from a to b is continuous (with respect to the partial order of the order-enriched category).

3 Notation. Since we shall be considering both $\mathcal{C}$ and $\mathcal{C}\bot$ at the same time, the notation $\mathcal{A}lg(F)$ may be ambiguous. It is not, if we take care that each functor has just one source and target —as it should be— for then the underlying category of $\mathcal{A}lg(F)$ is just src$F$. However, we wish to consider a strictness preserving functor F on $\mathcal{C}$ also as a functor on $\mathcal{C}\bot$ without a change in notation, for that would be cumbersome. Hence we indicate the underlying category explicitly, and write $\mathcal{A}lg(\mathcal{C}; F)$ and $\mathcal{A}lg(\mathcal{C}\bot; F)$. Another reading is this: the pair $(\mathcal{C}; F)$ is an endofunctor on $\mathcal{C}$, whereas the pair $(\mathcal{C}\bot; F)$ is an endofunctor on $\mathcal{C}\bot$, and for both of them the mapping on objects and morphisms is given by the mapping F; when applying functor $(\mathcal{C}\bot; F)$, say, we immediately unfold the definition of $(\mathcal{C}\bot; F)$ and write F in its place.

Even though in $\mathcal{C}$ and $\mathcal{C}\bot$ least fixed points exist for monotone morphism-mappings, it it still worthwhile to look for initial algebras and final co-algebras since the cata- and ana-equations bring forth nice calculation properties. We present a theorem that is well known, except perhaps for the finality claim; a consequence of the finality claim is that $U\mu F = U\nu F$ (or rather $U\mu F \cong U\nu F$). Section 6b is devoted to the proof of the theorem and a generalisation to arbitrary order-enriched categories.

4 Theorem (Reynolds [63]) *Let $\mathcal{C}$ be CPO, and F be a functor on $\mathcal{C}\bot$ that is locally continuous. Then $\mathcal{A}lg(\mathcal{C}\bot; F)$ has an initial object α, and for all φ in $\mathcal{C}$ (strict and nonstrict)*

$$x = \alpha^{\cup} ; Fx ; \varphi \quad \Rightarrow \quad x \sqsupseteq (\!|\varphi|\!) \qquad \text{cata-LEAST}$$

and the comparison is not necessarily an equality. The inverse $\alpha^{\cup}$ is final in $\mathcal{C}o\mathcal{A}lg(\mathcal{C}\bot; F)$, and even in $\mathcal{C}o\mathcal{A}lg(\mathcal{C}; F)$ if F is a locally continuous functor on the whole of $\mathcal{C}$.

5 Discussion. Law cata-LEAST is just an additional property of catamorphisms here; cata-CHARN is valid in *CPO*$\bot$ by definition of initiality.

In *CPO* all polynomial functors are locally continuous; see paragraph 13. Also each type functor (map functor) M induced by $\dagger$ is readily shown to be locally continuous if $\dagger$ is; the proof is a routine cpo continuity proof, and is much simpler than a proof of the categorical ω-cocontinuity of M; see Fokkinga and Meijer [24].

In view of the theorem strictness will play an essential rôle when discussing catamorphisms or cata-like concepts.

6b Hylomorphisms

This section contains the proof of the theorem. Hylomorphisms are least fixed points of a particular form, and are a useful tool for the proof and for programming in general.

6 Hylomorphisms. Let default category $\mathcal{C}$ be an arbitrary $O\bot$-category. Equations of the form $x = \varphi \,;\, Fx \,;\, \psi$ play an important rôle; these specialise to both the cata- and ana-equation. (In the previous chapters we didn't consider these equations since in general there is no unique solution, and it is only in the present context that the notion of 'least' solution makes sense.) So, by definition, the F**-hylomorphism** of φ, ψ, denoted $\llbracket \varphi, \psi \rrbracket_F$, is: the least solution of that equation, that is,

$$\llbracket \varphi, \psi \rrbracket_F \quad = \quad \text{least solution of } x = \varphi \,;\, Fx \,;\, \psi \,.$$

For this to make sense as a definition in $\mathcal{C}$ it is required that F is a mapping such that $Fx\colon Fa \to Fb$ for each $x\colon a \to b$, and that F is continuous as a function of x (locally continuous if F is a functor). There is no need, here, for the two other functor axioms to be satisfied. Moreover, φ must be an F-co-algebra and ψ an F-algebra.

$$\varphi \text{ is } F\text{-co-algebra and } \psi \text{ is } G\text{-algebra} \quad \Rightarrow \quad \llbracket \varphi, \psi \rrbracket_F\colon U\varphi \to U\psi \qquad \text{hylo-TYPE}$$

(Erik Meijer has coined the name hylomorphism; hylo comes from the Greek ύλη meaning "matter", after the Aristotelian philosophy that form(= generated) and matter(= reduced) are one.) Here are some laws for F-hylomorphisms, for a locally continuous endofunctor F on $\mathcal{C}$.

$$\varphi, \psi \text{ strict, } F \text{ preserves strictness} \quad \Rightarrow \quad \llbracket \varphi, \psi \rrbracket \text{ strict} \qquad \text{hylo-STRICT}$$

$$f\colon \varphi \succ\!\!-_F \psi \;\wedge\; g\colon \chi \to_F \omega \;\wedge\; g \text{ strict} \quad \Rightarrow \quad f \,;\, \llbracket \psi, \chi \rrbracket \,;\, g = \llbracket \varphi, \omega \rrbracket \qquad \text{hylo-FUSION}$$

$$\psi \,;\, \chi = id \quad \Rightarrow \quad \llbracket \varphi, \psi \rrbracket \,;\, \llbracket \chi, \omega \rrbracket = \llbracket \varphi, \omega \rrbracket \qquad \text{hylo-COMPOSE}$$

$$\varepsilon\colon F \mathrel{\dot\to} G \quad \Rightarrow \quad \llbracket \varphi,\ \varepsilon \,;\, \psi \rrbracket_F = \llbracket \varphi \,;\, \varepsilon,\ \psi \rrbracket_G \qquad \text{hylo-SHIFT}$$

Meijer [50] gives the –simple– proofs; using least fixed point induction, also called Scott-deBakker induction, for all but hylo-SHIFT.

7 Proof of the theorem. Reynolds [63] proves Theorem 4, except for the last statement, for the particular case $\mathcal{C} = CPO$. Schmidt [66, Chapter 11] gives a readable and precise account of Reynolds' proof. The crux of the proof is Scott's inverse limit construction; it yields a strict F-algebra α such that the inverse $\alpha^{\cup}$ exists and is strict, and $id_{U\alpha} = \llbracket \alpha^{\cup}, \alpha \rrbracket$.

> *Aside.* Actually, $(\alpha^{\cup}, \alpha)$ is initial in the category of fixed points of F_{PR} in $\mathcal{C}_{PR}$, where $\mathcal{C}_{PR}$ is the category of *PR*ojection pairs, or retractions, of $\mathcal{C}$. A retraction from a to b is a pair $(f\colon a \to b,\ g\colon b \to a)$ satisfying $f \,;\, g = id_a$ and $g \,;\, f \sqsubseteq id_b$. It is easy to see that both components of a retraction are strict. In the inverse limit construction only strict functions play a rôle; F is applied only to strict functions, and the construction is carried out entirely in $\mathcal{C}\bot$. (Formally, $\mathcal{C}_{PR} = (\mathcal{C}\bot)_{PR}$.)

Now, this result implies the other positive parts of the theorem in a nice way. First we show that taking $[\![\alpha^{\cup}, \varphi]\!]$ as $(\![\varphi]\!)$ makes cata-CHARN true on $\mathcal{C}_\perp$. We do so by establishing the equivalence by circular implication as follows. For each strict x,

$$\begin{array}{ll}
 & x = [\![\alpha^{\cup}, \varphi]\!] \\
\equiv & \qquad \text{above: } id = [\![\alpha^{\cup}, \alpha]\!] \\
 & [\![\alpha^{\cup}, \alpha]\!] \,;\, x = [\![\alpha^{\cup}, \varphi]\!] \\
\Leftarrow & \qquad \text{hylo-FUSION, noting that } x \text{ is assumed to be strict} \\
 & \alpha \,;\, x = Fx \,;\, \varphi \\
\Leftarrow & \qquad \text{fixed point property, definition hylo as a fixed point} \\
 & x = [\![\alpha^{\cup}, \varphi]\!].
\end{array}$$

Moreover, for strict φ, hylomorphism $[\![\alpha^{\cup}, \varphi]\!]$ is itself strict by hylo-STRICT and strictness of $\alpha^{\cup}$, hence it is in $\mathcal{A}lg(\mathcal{C}_\perp; F)$ as required. So, α is initial in $\mathcal{A}lg(\mathcal{C}_\perp; F)$.

The proof that $\alpha^{\cup}$ is final in $CoAlg(\mathcal{C}_\perp; F)$, or even in $CoAlg(\mathcal{C}; F)$ if F is locally continuous functor on $\mathcal{C}$, is entirely dual. (The strictness conditions disappear by dualisation, as observed in paragraph 2.) Actually, from the initiality of α in $\mathcal{A}lg(\mathcal{C}_\perp; F)$ it follows that $id = [\![\alpha^{\cup}, \alpha]\!]$, hence, by the above argument, that $\alpha^{\cup}$ is final in $CoAlg(\mathcal{C}_\perp; F)$.

An example in paragraph 9 confirms that, in general, α is not initial in $\mathcal{A}lg(CPO; F)$.

8 Abstracting from *CPO*. The inverse limit construction employed by Reynolds does not depend on specific properties of category *CPO*. Abstracting from *CPO*, the conditions on the category and the functor are:

> $\mathcal{C}$ is a localised $O_\perp$-category, with $\mathcal{C}_{PR}$ being an ω-category.
> F is a locally continuous functor on $\mathcal{C}_\perp$.

Smyth and Plotkin [68] and Bos and Hemerik [12] generalise Reynolds' proof discussed in paragraph 7 to this more general setting (and define the notion 'localised'), except, maybe, for the equation $id = [\![\alpha^{\cup}, \alpha]\!]$. (I couldn't find this equation in their papers, but it is certainly derivable from their by-products.)

9 Strictness inevitable. There is no simple modification to the equational characterisation of $(\![\varphi]\!)$ that makes it valid for nonstrict φ as well. Thus initiality in $\mathcal{A}lg(\mathcal{C}_\perp; F)$ is the best possible result. In particular we shall prove the following discrepancies, for some $\mathcal{C}, F, \alpha$ that meet the conditions of the theorem, and some φ, x. Here \$ is some "strictifying" function; the only properties that we assume of \$ are $\$\varphi = \varphi$ for strict φ, and $\$\varphi$ is strict for each φ.

$$\begin{array}{llcl}
\text{(a)} & [\![\alpha^{\cup}, \varphi]\!] = x & \not\equiv & \alpha \,;\, x = Fx \,;\, \varphi \\
\text{(b)} & [\![\alpha^{\cup}, \varphi]\!] = x & \not\equiv & \alpha \,;\, x = Fx \,;\, \varphi \ \wedge\ x \text{ strict} \\
\text{(c)} & [\![\alpha^{\cup}, \varphi]\!] = x & \not\equiv & \alpha \,;\, x = Fx \,;\, \$\varphi \\
\text{(d)} & [\![\alpha^{\cup}, \varphi]\!] = x & \not\equiv & \alpha \,;\, x = Fx \,;\, \$\varphi \ \wedge\ x \text{ strict}.
\end{array}$$

True enough for all φ and x in $\mathcal{C}$

(e) $\quad [\![\alpha^{\cup}, \$\varphi]\!] = x \quad \equiv \quad \alpha \,;\, x = Fx \,;\, \$\varphi \ \wedge\ x$ strict,

but in view of the assumed properties of \$ this equivales initiality of α in $\mathcal{A}lg(\mathcal{C}_{\perp}; F)$, which we already know.

For (a) and (c) observe that in general a fixed point (x in the rhs) need not be the least fixed point (x in the lhs). Specifically, let $\mathcal{C} = CPO$ and $F = I$ (hence $U\alpha = \{\perp\}$ and $\alpha = \perp = \alpha^{\cup}$). Furthermore, let a be a cpo with an element $\bullet$ different from $\perp_a$, and take $\varphi = id_a$ (which is strict). Then both the constant functions $x := \underline{\perp_a}$ and $x := \underline{\bullet}$ satisfy the right hand side, and since they differ they do not both satisfy the equality of the left hand side. So for at least one of them the discrepancy is true.

For (b) and (d) observe that the left hand side does not imply 'x strict'. Specifically, let $\mathcal{C} = CPO$, let a be a cpo with an element $\bullet$ different from $\perp_a$, and take $F = \underline{a}$ and $\varphi = \underline{\bullet}$: $a \to a$. Take $x = [\![\alpha^{\cup}, \varphi]\!] = \varphi$, which is not strict. So the left hand side is true, the right hand side is false, and therefore the discrepancy is true.

As an aside we conclude that Theorem 4 is invalidated when '$\mathcal{A}lg(CPO_{\perp}; F)$' is replaced by 'the category $ContAlg(CPO; F)$ of continuous F-algebras'. Here $ContAlg(CPO; F)$ is: the subcategory of $\mathcal{A}lg(CPO; F)$ in which the morphisms are the strict morphisms of CPO. (Category $\mathcal{A}lg(CPO_{\perp}; F)$ is a full subcategory of $ContAlg(CPO, F)$.) The counterexample is given in (b) above. This contradicts the *claim* of Reynolds [63], namely that $ContAlg(CPO, F)$ has an initial object α with $[\![\alpha^{\cup}, \varphi]\!]$ being the unique morphism from α to φ. Indeed, his *proof* shows initiality of α in $\mathcal{A}lg(CPO_{\perp}; F)$, since he treats the (possibly singleton) collection φ_s (for $s \in S$) of operations of an algebra, as a single operation $\nabla(s \in S :: \ \varphi_s)$, with ∇ defined as the $\triangledown$-operation for the separated sum in paragraph 13 below. For the separated sum, $\nabla(s \in S :: \ \varphi_s)$ is strict, even if the φ_s are not.

6c Consequences

10 A programming paradigm. The equality $\nu F = (\mu F)^{\cup}$ (which is what the theorem implies for $\mathcal{C}_{\perp}$) allows a formalisation of a programming paradigm that has been —up to now— problematic to formalise otherwise.

Suppose a function f: $a \to b$ has been specified in some way or another, and it is requested to design an algorithm, a program, for f. The following procedure is sometimes (often?) applicable. Invent a datastructure, tree say, that can be built in an easy way from f's input, and from which f's output is easily obtained. "Easy" means: in a homomorphic way, say the datastructure is generated by an anamorphism from the input, and the datastructure is reduced by a catamorphism to the output. Thus

$$f \quad = \quad generate \,;\, reduce \quad : \quad a \to b$$

where

$$\begin{array}{lcll} generate & = & [\![divide]\!]_F & : \ a \to c \\ reduce & = & (\![conquer]\!)_F & : \ c \to b\,, \end{array}$$

for some operations *divide* and *conquer*. Here, c is the type of the intermediate data-structure. These equations imply that $U\nu F = c = U\mu F$. In case the initial F-algebra has an inverse that is a final F-co-algebra too, there is no contradiction here. But in $\mathcal{S}et$ and in other categories this is not the case in general, and the above equations together are sometimes absurd.

The equality $\nu F = (\mu F)^{\cup}$ is sufficient but not necessary. What is really needed is that the "range" of the F-anamorphism *generate* is embeddable in the carrier of the initial F-algebra, so that the proper expression for f would read *generate* ; ***embed*** ; *reduce*. I do not know under what conditions on the category and the functor such an embedding exists, except for the case at hand: order-enriched categories.

Since the inverse $\alpha^{\cup}$ of the initial algebra α is the final co-algebra β, function f itself satisfies a fixed point equation too.

$$\begin{array}{lcl} f & = & generate \mathbin{;} reduce \\ & = & divide \mathbin{;} F generate \mathbin{;} \beta^{\cup} \mathbin{;} \alpha^{\cup} \mathbin{;} F reduce \mathbin{;} conquer \\ & = & divide \mathbin{;} F(generate \mathbin{;} reduce) \mathbin{;} conquer \\ & = & divide \mathbin{;} F f \mathbin{;} conquer\,. \end{array}$$

Actually, by hylo-SPLIT 12 function f is the least fixed point of this equation. The operational evaluation of the f so defined does no longer refer to the intermediate datastructure of type c. Thus that value is a *virtual datastructure* that has been of great help in the algorithm design but does not exist at all during program execution.

The name 'virtual datastructure' has been coined by Doaitse Swierstra [71]; he used the method to derive a linear time algorithm for the so-called low segment problem and related problems (In the low segment problem, fx = the longest low segment in list x; a segment is *low* if: the maximum value in it is at most its length).

11 Extending "catamorphisms". Theorem 4 asserts, for a large class of functors F, the existence of an algebra that is initial in $\mathcal{A}lg(\mathcal{C}_{\perp}; F)$ but not necessarily initial in $\mathcal{A}lg(\mathcal{C}; F)$. So function $(\![\ ]\!)$ is not applicable to nonstrict φ. Law cata-LEAST suggests the possibility to *extend* function $(\![\ ]\!)$ to nonstrict φ, with the definition that $(\![\varphi]\!)$ is the least solution x of the equation $x = \alpha^{\cup} \mathbin{;} Fx \mathbin{;} \varphi$.

These observations raise the question whether $\mathcal{C}_{\perp}$ or $\mathcal{C}$ should be taken as the universe of discourse, and whether function $(\![\ ]\!)$ should be extended to nonstrict φ. Here are three alternatives.

(a) Take $\mathcal{C}$ as universe of discourse and extend $(\![\ ]\!)$.
(b) Take $\mathcal{C}$ as universe of discourse and don't extend $(\![\ ]\!)$.
(c) Take $\mathcal{C}_{\perp}$ as universe of discourse.

Let us discuss these in turn.

Ad (a). Obviously, cata-CHARN does not hold in $\mathcal{C}$, since in general α is not initial in $\mathcal{Alg}(\mathcal{C}; F)$. Moreover, as shown in paragraph 9, there is no simple modification to cata-CHARN that makes it valid for the extended $\llparenthesis\ \rrparenthesis$ as well. Nevertheless, it will turn out that cata-CHARN and the other laws are valid for the extended $\llparenthesis\ \rrparenthesis$ provided strictness is assumed of only *some* of the ingredients of the law.

Ad (b). When functions may be nonstrict and $\llparenthesis\ \rrparenthesis$ is not extended, then in the laws for $\llparenthesis\ \rrparenthesis$ strictness is to be required for *all* the ingredients. This gives laws that are less applicable than in the first alternative.

Ad (c). The last alternative is to take $\mathcal{C}_\perp$ as the universe of discourse, and discard nonstrict functions altogether. This is the approach followed by Ross Paterson [58, 59]. He did not notice the initiality asserted by Theorem 4, but only weak initiality.

We shall explore alternative (a) somewhat further.

12 Laws for the "extended catamorphisms". To illustrate the consequences of extending the notation $\llparenthesis\ \rrparenthesis$ to nonstrict algebras (for which it is no longer a catamorphism) we list here without proof some of those laws. Meijer [50] discusses them in detail.

$\llparenthesis\varphi\rrparenthesis = \llbracket\alpha^{\cup}, \varphi\rrbracket$ (= least solution of $x = \alpha^{\cup} ; Fx ; \varphi$)			cata-LFP
φ strict	$\Rightarrow$	$\llparenthesis\varphi\rrparenthesis$ strict	cata-STRICT
$x\colon \alpha \to_F \varphi \ \wedge\ x$ strict	$\equiv$	$x = \llparenthesis\varphi\rrparenthesis \ \wedge\ \varphi$ strict	cata-CHARN
$x, y\colon \alpha \to_F \varphi \ \wedge\ \varphi$ strict	$\Rightarrow$	$x = y$	cata-UNIQ
$x\colon \varphi \to_F \psi \ \wedge\ x$ strict	$\Rightarrow$	$\llparenthesis\varphi\rrparenthesis ; x = \llparenthesis\psi\rrparenthesis$	cata-FUSION
$\varphi ; \psi = id$	$\Rightarrow$	$\llparenthesis\varphi\rrparenthesis ; [\!(\psi)\!] = id$	cata-ana-ID
$\llbracket\varphi, \psi\rrbracket = [\!(\varphi)\!] ; \llparenthesis\psi\rrparenthesis$			hylo-SPLIT

Notice that hylo-SPLIT only makes sense if $U\nu F = U\mu F$, which is implied by the theorem.

13 Sum in *CPO*. There are two kinds of disjoint union in *CPO*. The *coalesced sum* $a \oplus b$ identifies $\perp_a$ and $\perp_b$ in the union into $\perp_{a+b}$. The *separated sum* $a + b$ keeps $\perp_a$ and $\perp_b$ apart in the union and adds a new bottom element $\perp_{a+b}$. (Further details are not relevant here, and may be obvious anyway.) Category *CPO* has no categorical sum, whereas the coalesced sum $\oplus$ is a categorical sum in $CPO_\perp$. Manes and Arbib [45] explain this in detail. Both $+$ and $\oplus$ are locally continuous functors on $CPO_\perp$, and $+$ is a locally continuous functor on *CPO* too, whereas $\oplus$ is not a functor on *CPO*.

What kind of sum to choose? The coalesced sum $\oplus$, being a categorical sum, has nice calculation properties. The separated sum $+$ corresponds closely to the disjoint union in fully lazy functional languages (as explained below). The carrier of $\mu(\underline{1} + I)$ has an infinite element (the number 'infinite' is the limit of all $succ^n(\perp)$), whereas the carrier of $\mu(\underline{1} \oplus I)$ is the flat cpo with elements $0, 1, 2, \ldots$ ordered besides each other and above $\perp$. Sometimes infinite elements are wanted (in particular for lists) and sometimes they are unwanted (think of the algebra of natural numbers), so the language designer might choose to provide both. Lehmann and Smyth [40] discuss this phenomenon extensively.

Even though the separated sum $+$ is not a categorical sum in $CPO_\perp$, it satisfies exactly the $\triangledown$-CHARN property in $CPO_\perp$, and therefore it has almost the same calculation properties as the coalesced sum $\oplus$ has. To be specific, here is the definition of the separated sum. (In this definition variables A, B denote sets (pointed cpo's) and a, b, x, y denote elements.)

$$\begin{array}{lcl}
A + B & = & \{\perp\} \ \cup \ \{0\} \times A \ \cup \ \{1\} \times B \\
x \sqsubseteq y & \equiv & x = \perp \ \vee \ (x_0 = y_0 \ \wedge \ x_1 \sqsubseteq y_1) \\
f \triangledown g & = & \perp \mapsto \perp \ \cup \ (0, a) \mapsto f(a) \ \cup \ (1, b) \mapsto g(b) \\
\mathit{inl} & = & a \mapsto (0, a) \\
\mathit{inr} & = & b \mapsto (1, b) \,.
\end{array}$$

Now recall the equivalence $\triangledown$-CHARN that characterises the categorical sum:

$$\mathit{inl} \,;\, f = g \ \wedge \ \mathit{inr} \,;\, f = h \quad \equiv \quad f = g \triangledown h \,.$$

The equivalence does not hold on CPO since CPO has no sums; indeed, the right hand side determines f completely for given g, h but the left hand side does not determine the outcome of $f(\perp)$. The equivalence does hold on $CPO_\perp$, that is, when f, g, h range over strict morphisms, as is easily verified. Yet the separated sum is not a categorical sum in $CPO_\perp$; this is because *inl* and *inr* as defined above are not in $CPO_\perp$ since they are not strict.

In an operational interpretation the difference between the separated and coalesced sum is explained as follows. In case of the separated sum, value $(0, \perp_a)$ as input for a program f means that the tag 0 is fully determined and f can use this information to produce already some part of its output, e.g., the tag of its result, or the complete result if it is independent of the actual tagged value. In case of the coalesced sum, however, $(0, \perp_a)$ is identified with $(1, \perp_b)$ into $\perp_{a+b}$, and for program f there is no information at all, not even the information that the tag of the input is 0. Clearly both sums are implementable.

Similar observations hold for the cartesian product and smashed product.

6d Conclusion

A quick glance at the literature shows without any doubt that many algorithms are defined as least solutions of equations $x = \mathcal{F}x$ for which it is unknown whether there is a unique solution, or for which it is known that there are several solutions. As long as these kind of algorithms are being derived a theory that deals with this phenomenon is needed. Our joint study shows that the very elegant laws for initiality get lost in the setting of lazy evaluation (full CPO), but similar properties still do hold at the cost of —nasty— strictness conditions. (It is, in fact, unwise to call the extended catamorphism still a catamorphism, since it is not a catamorphism in the sense of paragraph 3.26 and 3.30.) In particular, equational reasoning may still be possible to a large extent.

The alternative is to reason and program with strict functions only, taking $CPO_\perp$ as the universe of discourse. After all, this does not prohibit infinite datastructures (streams) as we have already seen in the previous chapters, and the additional advantage of $CPO_\perp$ over $\mathcal{S}et$, say, is the existence of least fixed points. As in the previous chapters the resulting theory is elegant, but it is not applicable to modern functional languages since these do contain nonstrict functions. (The wish that "substitution of equals for equals" doesn't change the semantics of programs, together with the presence of nonterminating programs, requires functions to be nonstrict. For example, suppose that f has been defined by the syntactic declaration '$fx = 3$', suggesting that fx and 3 may be substituted for each other. If really fx equals 3 for all x, even when x denotes $\perp$, then f is nonstrict.)

Appendix A

Category theory

There are several good introductory texts to category theory for computing scientists; for example, Goldblatt [27, Chapter 2,3,9], Barr and Wells [7], and Pierce [60]. These are strongly recommended, in that order. This appendix introduces only what is needed to read the main text (and also explains a bit of my perception of the concepts involved).

1 **Definition.** A **category** is: the following data, subject to the axioms listed in paragraph 3.

- A collection of things called **objects**.
- A collection of things called **morphisms**, sometimes called arrows.
- Two functions from morphisms to objects, called **source** and **target** function.
- A binary partial operation on morphisms, called **composition**.
- For each object a a distinguished morphism, called **identity** on a.

Actually, these data define (the basic terms of) the **categorical language** in which properties of the category can be stated. If you happen to know what the objects really are, you may use those aspects in your statements, but then you are *not* working categorically. Category $\mathcal{S}et$ is: the category whose objects are sets, whose morphisms are typed total functions, and whose composition and identities are function composition and identity functions respectively; these do satisfy the axioms listed below. Thus, doing set theory categorically enforces the strait jacket of expressing everything with function composition only, without using explicit arguments (set elements) and function application. Once mastered it is an elegant way of expressing. We shall often interpret our results in $\mathcal{S}et$.

2 **Notation.** Here is some default notation for categories. The name of a category may and should be added as a subscript or otherwise in order to avoid confusion when there are several categories under discussion. Let $\mathcal{A}$ be a category. The source and target function are denoted src and tgt. The infix written composition is denoted $;$ or $\circ$ or not

at all (juxtaposition) with the convention $f \mathbin{;} g = g \circ f = g f$. Within a term denoting a morphism, the symbol $;$ has **strongest separation** power (binds weakest); juxtaposition has strongest binding power, more than any binary operation symbol. Let a be an object in $\mathcal{A}$; then the identity on a is denoted id_a, and the subscript is omitted if it is clear or can be derived from the context. Formula $f\colon a \to_{\mathcal{A}} b$ means that a and b are objects in $\mathcal{A}$ and f is a morphism in $\mathcal{A}$ with source a and target b; we also say that f is a morphism **from** a **to** b, and that $a \to b$ is the **type** of f in $\mathcal{A}$.

3 Axioms. With the above notation the axioms read, for a category $\mathcal{A}$:

- If $f\colon a \to_{\mathcal{A}} b$ and $g\colon b \to_{\mathcal{A}} c$, then $f \mathbin{;} g : \ a \to_{\mathcal{A}} c$.
- For each object a of $\mathcal{A}$, $id_a\colon a \to_{\mathcal{A}} a$.
- Composition is associative.
- Whenever $f\colon a \to_{\mathcal{A}} b$, then $id_a \mathbin{;} f = f = f \mathbin{;} id_b$.

The axioms are so basic that we shall nowhere invoke or mention them explicitly. Whenever we write a composition, we assume that the free variables are typed in such a way that the composition is defined, that is, the sources and targets match.

4 Discussion. The notion of category has been designed to formalise in a *uniform* way the intuitive notion of (various kinds of) mathematical structure. Concrete examples of structure are: no structure at all, partial order, and complete partial order; abstract structures like diagrams and algebras turn up in Chapter 2 and 3 respectively. The data and axioms of a category should fix the structure of interest. However, the above axioms are so weak that not only many structures can be rendered as a category, but also categories exist for which there is no interpretation as formalising a known or reasonable mathematical structure: each directed graph is a category (if all the finite compositions of arrows are adjoined as arrows).

Uniformity is achieved by modeling structure "externally," that is, via the structure preserving transformations only, and not as an aspect of an object in isolation. Formulas — expressing properties of a category— can only be built by the data of the category, mainly composition of morphisms, using logical connectives and quantifications. Thus, whereas the objects may be thought to carry the structure, it is the morphisms that effectively represent the structure. (Hence the word morphism: $\mu o \rho \varphi \eta$ means form or structure.) For example, suppose you want to study complete partially ordered sets, and thus take sets as the objects of the category. If you take *all* functions between sets as morphisms, then the equations in this category do not state properties of the structure you are interested in. However, if you take precisely the monotonic functions as the morphisms, then in this category the equations do say something about the partial order: the monotonic functions are precisely those that preserve the partial order. To investigate the completeness (existence of limits) too, you should take the continuous (= limit preserving) functions as **morphisms**. Of course, you may investigate all three at the same time: categories need not be disjoint.

The axioms on the morphisms and composition are motivated by the observation that for all (?) mathematical structures the structure-preserving transformations do satisfy them. As we've said, they are very weak. By imposing extra axioms, still in the categorical language, the categories may have more of the properties you are interested in. For example, a *topos* is a category whose extra axioms give those properties of objects that are quite characteristic of *sets*; and this is done by "external" means only: set membership is not used at all. We shall nowhere need the axioms for a topos. As a result, our results are often very general, and hence very weak at the same time. Nevertheless, I myself was surprised that theorems relevant for transformational programming could be proved for categories admitting interpretations that no human being can ever imagine.

5 **Isomorphism.** Let $\mathcal{A}$ be a category, and a, b be objects in $\mathcal{A}$. Then a and b are called **isomorphic** in $\mathcal{A}$ if: there exist morphisms $f\colon a \to_{\mathcal{A}} b$ and $g\colon b \to_{\mathcal{A}} a$ that are each others inverse, that is, $f \mathbin{;} g = id_a$ and $g \mathbin{;} f = id_b$. In this case we write $a \cong_{\mathcal{A}} b$ and $f\colon a \cong_{\mathcal{A}} b$, and say f and g are **isomorphisms**. If a and b are isomorphic and $f\colon a \to_{\mathcal{A}} b$, then there is precisely one $g\colon b \to_{\mathcal{A}} a$ satisfying $f \mathbin{;} g = id_a$ and $g \mathbin{;} f = id_b$.

If P is a property of objects that holds for precisely one entire class of isomorphic objects, then we often speak of **the** P**-object** rather than of an object satisfying P; we also say that the object is *unique up to isomorphism*. For example, in $\mathcal{S}et$ 'the one-point set' is unique up to isomorphism; it is denoted $\mathit{1}$.

Discussion. Isomorphic objects are often called 'abstractly the same' since for most categorical purposes one is as good as the other: each morphism to/from the one can be extended to a morphism to/from the other using the morphisms that establish the isomorphism. (The preceding sentence is informal intuition; I do not know of a formalisation of the idea.) For example, in $\mathcal{S}et$ all sets of the same cardinality are isomorphic, hence 'abstractly the same'. If you want to distinguish some of them on account of structural properties, a partial order say, you should not take $\mathcal{S}et$ as the category but another one for which the morphisms better reflect your intention.

6 **Functor.** A functor is a mapping from one category to another that preserves the categorical structure. Functors form a tool to abstract from the source and target structure of a morphism, see paragraph 8.

Formally, let $\mathcal{A}$ and $\mathcal{B}$ be categories; then a **functor** from $\mathcal{A}$ to $\mathcal{B}$ is: a mapping F that sends objects of $\mathcal{A}$ to objects of $\mathcal{B}$, and morphisms of $\mathcal{A}$ to morphisms of $\mathcal{B}$ in such a way that

$$\begin{array}{lcll} Ff & : & Fa \to_{\mathcal{B}} Fb & \text{whenever } f\colon a \to_{\mathcal{A}} b \\ F id_a & = & id_{Fa} & \text{for each object } a \text{ in } \mathcal{A} \\ F(f \mathbin{;} g) & = & Ff \mathbin{;} Fg & \text{whenever } f \mathbin{;} g \text{ is well defined} \end{array}$$

Formula $F\colon \mathcal{A} \to \mathcal{B}$ means that $\mathcal{A}$ and $\mathcal{B}$ are categories and F is a functor from $\mathcal{A}$ to $\mathcal{B}$. The **identity** functor from $\mathcal{A}$ to $\mathcal{A}$ is denoted I; it is the identity on both the objects and the morphisms. For functors $G\colon \mathcal{B} \to \mathcal{C}$ and $F\colon \mathcal{A} \to \mathcal{B}$, the composite GF is defined by $(GF)x = G(Fx)$ for all objects and morphisms x of $\mathcal{A}$; thus defined GF is a functor of type $\mathcal{A} \to \mathcal{C}$, and we write just GFx without parentheses. (Since functors "preserve the structure" one may expect that they form the morphisms of a category; the objects of that category are categories. Indeed, with the above definitions for I and FG the axioms are fulfilled.) Let $\mathcal{A}$ and $\mathcal{B}$ be categories, and b an object in $\mathcal{B}$; then the **constant** functor $\underline{b}\colon \mathcal{A} \to \mathcal{B}$ is defined by $\underline{b}a = b$ and $\underline{b}f = id_b$ for all objects a and morphisms f in $\mathcal{A}$. An **endofunctor** is: a functor whose source and target category are the same.

7 Bifunctor. A **bifunctor** is a functor that takes two arguments rather than one. We shall use $\dagger$ and $\ddagger$ as variables (infix operation symbols) for bifunctors. The axioms for a bifunctor read:

$$\begin{array}{lcl} f\colon a \to b \ \wedge \ g\colon c \to d & \Rightarrow & f \dagger g\colon a \dagger c \to b \dagger d \\ (f \,;\, g) \dagger (h \,;\, j) & = & f \dagger h \,;\, g \dagger j \\ id_a \dagger id_b & = & id_{a\dagger b}\,. \end{array}$$

Actually, a bifunctor $\dagger$ from $\mathcal{A}$, $\mathcal{B}$ to $\mathcal{C}$ is a normal functor $\dagger\colon \mathcal{A} \times \mathcal{B} \to \mathcal{C}$, where category $\mathcal{A} \times \mathcal{B}$ is the so-called product category of $\mathcal{A}$ and $\mathcal{B}$. The definition of the **product category** of two categories is straightforward: everything is coordinatewise. Important bifunctors are $\times$ and $+$; in $\mathcal{S}et$, $a \times b$ and $a + b$ denote the cartesian product (with **extractions** *exl*, *exr*) and disjoint union (with **injections** *inl*, *inr*) respectively. Applied to functions the bifunctors $\times$ and $+$ yield "componentwise acting" functions:

$$\begin{array}{lcl} f \times g \,;\, exl & = & exl \,;\, f \\ inl \,;\, f + g & = & f \,;\, inl\,, \end{array}$$

and similarly for *exr*, *inr*. (Rewrite these equations with explicit arguments and function application to see the usual definitions.)

If $\dagger$ is a bifunctor and F, G are functors, then $F \dagger G$ denotes the functor defined by $(F \dagger G)x = Fx \dagger Gx$ for all objects and morphisms x. In particular, $I\!I = I \times I$; it maps each x onto $x \times x$. The **polynomial functors** are those that can be written using I, all $\underline{a}$, $\times$ and $+$, and functor composition only.

8 Source and target structure. Functors are a tool to abstract from the particular source and target structure of morphisms (operations, functions, algebras). For example, a binary operation on a has type $a \times a \to a$, that is, $I\!Ia \to Ia$. As another example, function $n \mapsto (n \operatorname{div} 10,\ n \bmod 10)$ has type $I nat \to I\!I nat$. More generally, a morphism of type $Fa \to Ga$ has a source type whose 'structure' is given by functor F; the structure of its target type is given by G. In a typing $Fa \to Ga$ you may always read I or $I\!I$ for F and G in order to get a less abstract typing. (Notice also that a may be an object from a product category, so that effectively F and G may take several arguments.)

9 Duality. In $\mathcal{S}et$ the cartesian product and disjoint union are in a sense dual to each other, and each statement about one of these can be dualised to a statement about the other (though dualisation need not preserve validity of the statements). For example, consider the statement, theorem in $\mathcal{S}et$, that a function $f\colon a \to b \times c$ is fully determined by the two composites $f ; exl\colon a \to b$ and $f ; exr\colon a \to c$. Dualisation gives the statement, also a theorem in $\mathcal{S}et$, that a function $f\colon b + c \to a$ is fully determined by the two composites $inl ; f\colon b \to a$ and $inr ; f\colon c \to a$.

Dualisation $\mathcal{D}$ of a statement in the language of category theory is easy. Define the dual $\mathcal{D}f$ of a term f in the categorical language by

$$\begin{array}{lcl} \mathcal{D}(\mathrm{src} f) & = & \mathrm{tgt}(\mathcal{D}f) \\ \mathcal{D}(\mathrm{tgt} f) & = & \mathrm{src}(\mathcal{D}f) \\ \mathcal{D}(id_a) & = & id_a \\ \mathcal{D}(f ; g) & = & \mathcal{D}g ; \mathcal{D}f . \end{array}$$

Then, for each definition expressed in the categorical language, of a concept or construction *xxx*, you obtain another concept, often called *co-xxx* if no better name suggests itself, by dualising each term in the definition. Also, for each equation $f = g$ provable from the above axioms of category theory (hence valid for all categories), the equation $\mathcal{D}f = \mathcal{D}g$ is provable too. Thus dualisation cuts work in half, and gives each time two concepts or theorems for the price of one. In this sense the cartesian product and disjoint union are dual to each other in category $\mathcal{S}et$.

Another easy way of dualising a term denoting a morphism is simply replacing each $;$ by $\circ$. However, the presence of both compositions for the same morphisms is not practical.

10 Naturality. Let $F, G\colon \mathcal{A} \to \mathcal{B}$ be functors. In the terminology of paragraph 8 each Fa denotes a structured type and F denotes the structure itself. A 'transformation' from structure F to structure G is, informally, something existing in $\mathcal{B}$ that provides for a way to go from Fa to Ga, for each a. The transformation is natural if, in addition, every two morphisms Ff and Gf are the *same* modulo the inevitable transformation between their sources and targets.

Formally, a **transformation** from F to G is: a family ε of morphisms

$$\varepsilon_a \quad : \quad Fa \to_{\mathcal{B}} Ga \quad \text{for each } a \text{ in } \mathcal{A}.$$

A transformation ε from F to G is **natural**, denoted $\varepsilon\colon F \dot{\to} G$, if:

$$Ff ; \varepsilon_b \quad = \quad \varepsilon_a ; Gf \quad \text{for each } f\colon a \to_{\mathcal{A}} b.$$

This formula is (so natural that it is) easy to remember: member $\varepsilon_{\mathbf{target}\, f}$ has type $F(\text{target } f) \to G(\text{target } f)$ and therefore occurs at the target side of an occurrence of f; similarly $\varepsilon_{\mathbf{source}\, f}$ occurs at the source side of an f. Moreover, since ε is a transformation *from* F *to* G, functor F occurs at the source side of an ε and functor G at the target side.

The notation εa is an alternative for ε_a, and uses ε as a function.

11 Example natural transformations. Natural transformations are all over the place. Here are a few examples in $\mathcal{S}et$.

The family id of all identities id_a: $a \to a$ is a natural transformation id: $I \dot{\to} I$. Indeed, for each f: $a \to b$

$$If \mathbin{;} id_b \quad = \quad id_a \mathbin{;} If .$$

The family $split_a$: $a \to a \times a$, defined by $split_a(x) = (x, x)$, is a natural transformation $split$: $I \dot{\to} I\!I$. Indeed, for each f: $a \to b$

$$f \mathbin{;} split_b \quad = \quad split_a \mathbin{;} I\!I f$$

that is,

$$split_b(fx) \quad = \quad (fy, fz) \text{ where } (y,z) = split_a\, x, \quad \text{for all } x \in a .$$

The family of extractions $exl_{a,b}$: $a \times b \to a$ is a natural transformation exl: $X \times Y \dot{\to} X$, where $X, Y = Exl, Exr$: $\mathcal{S}et \times \mathcal{S}et \to \mathcal{S}et$ are the obvious extraction functors for product categories. Indeed, for each (f,g): $(a,b) \to (c,d)$

$$f \times g \mathbin{;} exl_{c,d} \quad = \quad exl_{a,b} \mathbin{;} f$$

that is,

$$exl_{c,d}(fx, gy) \quad = \quad f(exl_{a,b}(x,y)), \quad \text{for } (x,y) \in \; a \times b .$$

The family $swap_{a,b}$: $a \times b \to b \times a$ defined by $swap_{a,b}(x,y) = (y,x)$, is a natural transformation $swap$: $X \times Y \dot{\to} Y \times X$, where again $X, Y = Exl, Exr$. Indeed, for each (f,g): $(a,b) \to (c,d)$

$$f \times g \mathbin{;} swap_{c,d} \quad = \quad swap_{a,b} \mathbin{;} g \times f$$

that is,

$$swap_{c,d}(fx, gy) \quad = \quad (gu, fv) \text{ with } (u,v) = swap_{a,b}(x,y), \quad \text{all } (x,y) \in a \times b .$$

For the datatype of lists we define La = 'the set of lists over a' and Lf = the map $[x, y, z, \ldots] \mapsto [fx, fy, fz, \ldots]$. (Thus defined L is a functor.) Family $join_a$: $I\!I La \to La$ (joining two lists of type La) is a natural transformation $join$: $I\!I L \dot{\to} L$ since for each f: $a \to b$

$$I\!I Lf \mathbin{;} join_b \quad = \quad join_a \mathbin{;} Lf$$

that is,

$$(Lf)x \;\text{join}_b\; (Lf)y \quad = \quad (Lf)\,(x \,\text{join}_a\, y), \quad \text{for all } (x,y) \in La \times La .$$

12 Omitting subscripts. Let $\mathcal{A}$ be a category, let F, G: $\mathcal{A}^m \to \mathcal{A}^n$ be functors, and ε: $F \dot{\to} G$ be a natural transformation (here $\mathcal{A}^k$ is a k-fold product category). Then we shall write just ε in a term denoting a morphism in $\mathcal{A}$, thereby omitting the subscripts. This is common practice, and can probably be *formally* justified since —I conjecture— there is an algorithm that, given the types of the natural transformations and the morphisms occurring in a term, yields for each occurrence in the term the most general typing such that the entire composite is well typed (at all compositions the target and source type match). I haven't checked this in general; for polynomial functors I suspect that Milner's [52] *polymorphic type inference* algorithm does the job.

The omission of subscripts applies to all the examples given above. When a so-called cocone is considered as a natural transformation its subscripts cannot be omitted; this occurs in Chapter 2. Also, in Chapter 5 a so-called transformer T is considered as a natural transformation, and its subscript φ cannot be omitted since the outcome of T_φ is a morphism expressed in φ.

13 Laws for naturality. Write εa instead of ε_a, and define $(F\varepsilon)a = F(\varepsilon a)$ and $(\varepsilon G)a = \varepsilon(Ga)$. It follows that $(F\varepsilon)G = F(\varepsilon G)$, and parentheses are not needed. Furthermore, define $(\varepsilon \mathbin{;} \eta)a = \varepsilon a \mathbin{;} \eta a$. Some useful laws for natural transformations read:

$id\colon I \dashrightarrow I$			ntrf-ID
$\varepsilon\colon F \dashrightarrow G \ \wedge\ \eta\colon G \dashrightarrow H$	$\Rightarrow$	$\varepsilon \mathbin{;} \eta\colon F \dashrightarrow H$	ntrf-COMPOSE
$\varepsilon\colon F \dashrightarrow G$	$\Rightarrow$	$H\varepsilon J\colon HFJ \dashrightarrow HGJ$	ntrf-FTR
$f\colon a \rightarrow b$	$\equiv$	$\underline{f}\colon \underline{a} \dashrightarrow \underline{b}$	ntrf-CONST
$\varepsilon_i\colon F_i \dashrightarrow G_i \quad (i=0,1)$	$\Rightarrow$	$\varepsilon_0 \dagger \varepsilon_1\colon F_0 \dagger F_1 \dashrightarrow G_0 \dagger G_1$	ntrf-BIFTR

As announced in paragraph 3 it is assumed that the terms make sense; this constraints the typing of the variables substantially. Law ntrf-BIFTR is a specialisation of ntrf-FTR (using product categories and bifunctors). Further 'specialisations' may be obtained when εa is written without its 'subscript', and hence εG is written as ε. (From Laws ntrf-ID and ntrf-COMPOSE assert that natural transformations form the morphisms of a category, where *functors* are the objects.)

Here is an example (due to Roland Backhouse) of the use of the laws. Consider the usual datatype of lists with Lf denoting the map $[a_0, a_1, \ldots] \mapsto [fa_0, fa_1, \ldots]$. Suppose $inits, tails\colon L \dashrightarrow LL$ and $flatten\colon LL \dashrightarrow L$. Define $segs = inits \mathbin{;} Ltails \mathbin{;} flatten$. (Here we write natural transformations without the subscript; otherwise we would have written $segs = inits \mathbin{;} Ltails \mathbin{;} flatten L$.) Then $segs\colon L \dashrightarrow LL$, as shown by

$segs\colon L \dashrightarrow LL$

$\equiv$ definition *segs*

$inits \mathbin{;} Ltails \mathbin{;} flatten\colon L \dashrightarrow LL$

$\Leftarrow$ ntrf-COMPOSE

$inits\colon L \dashrightarrow LL \ \wedge\ Ltails\colon LL \dashrightarrow LLL \ \wedge\ flatten\colon LLL \dashrightarrow LL$

$\Leftarrow$ ntrf-FTR on middle and right conjunct

$inits\colon L \dashrightarrow LL \ \wedge\ tails\colon L \dashrightarrow LL \ \wedge\ flatten\colon LL \dashrightarrow L$

$\equiv$ assumptions

true.

Actually, the assumptions are valid, and therefore the conclusion too.

14 Epic, monic. For completeness' sake two more definitions. A morphism f is **epic** or an **epimorphism** if: $f \,;\, x = f \,;\, y \;\Rightarrow\; x = y$ for all x, y (of the right type). Dually, f is **monic** or a **monomorphisms** if: $x \,;\, f = y \,;\, f \;\Rightarrow\; x = y$ for all x, y. In $\mathcal{S}et$ a monomorphism is injective, and an epimorphism is surjective; there exist $\mathcal{S}et$-like categories where this is not true.

Samenvatting

In *Law and Order in Algorithmics* onderzoeken we een methode om computerprogramma's te maken. Om te voorkomen dat het onderzoek te ingewikkeld wordt, laten we een heleboel aspekten buiten beschouwing, zowel van programma's zelf als ook van het maken van programma's. Bijvoorbeeld, we bekommeren ons niet om de snelheid en benodigde computercapaciteit van programma's; we laten uitsluitend het invoer-uitvoer effect een rol spelen in onze beschouwingen. Voor het maken van een programma gaan we ervan uit dat er al een preciese beschrijving bekend is van het gewenste invoer-uitvoer effect; we houden ons dus niet bezig met de vraag hoe zo'n beschrijving tot stand komt.

Algorithmics. De onderzochte methode om computerprogramma's te maken, gaat als volgt. Een programma wordt afgeleid uit de beschrijving van het gewenste invoer-uitvoer effect door stap voor stap die beschrijving te veranderen (te transformeren) totdat uiteindelijk een vorm bereikt wordt die zelf een bevredigend computerprogramma is. In het bijzonder onderzoeken we díe aanpak waarbij de stappen heel erg lijken op de rekenstappen die bij de algebra op school gehanteerd worden. Bijvoorbeeld, in de algebra geldt: $(a+b)(a-b) = a^2 - b^2$ (we noemen dit een *rekenregel*), zodat bij het rekenen de uitdrukking $(a+b)(a-b)$ vervangen mag worden door $a^2 - b^2$, en omgekeerd. Een rekenregel is louter een gelijkheid van twee uitdrukkingen; de uitdrukkingen zijn verschillend van vorm, maar hebben wel dezelfde uitkomst. De theorie en praktijk van het *rekenen met programma's* heet **algoritmiek** (engels: algorithmics). Meertens [47] en Bird [9] hebben de eerste invulling aan algoritmiek gegeven. Ons werk is een aanvulling op de theorie ervan.

Law... . Een rekenregel noemen we ook wel **wet** (engels: law). In de rekenkunde zijn veel wetten voor getallen bekend en in gebruik. Het *systematisch ontdekken en gebruiken van wetten voor programma's* is het hoofddoel van ons onderzoek. In Hoofdstuk 3–6 doen we dit onderzoek, en maken daarbij gebruik van begrippen uit de categorie-theorie (een tak van wiskunde).

In Hoofdstuk 2 hebben we het systematisch ontdekken en gebruiken van wetten ook toegepast op categorie-theorie zelf. Het resultaat daarvan is een manier om in categorie-theorie bewijzen te leveren die een alternatief is voor bestaande methoden.

In Hoofdstuk 3 beschrijven we programma's met behulp van begrippen uit de categorie-theorie. Deze beschrijving is sterk beinvloed door het werk van Malcolm [42] en Hagino [29], en bevat op zich geen nieuwe resultaten.

In Hoofdstuk 4 beschouwen we een paar soorten 'recursieve' programma's. Voor deze programma's liggen een aantal wetten nogal voor de hand. We *bewijzen* de geldigheid van die wetten formeel, en geven een toepassing ervan.

> Een programma is *recursief* als het als onderdeel in zichzelf voorkomt of benoemd wordt, net zoals het spiegelbeeld bij twee spiegels die tegenover elkaar staan: ieder spiegelbeeld komt in zichzelf voor. Recursieve programma's komen in de praktijk veel voor. Bij een recursief programma kán het zo zijn dat er bij sommige invoer mogelijk niets aan uitvoer geproduceerd wordt, doordat de computerberekening in een oneindige lus (recursie!) raakt. Bij de soorten van recursieve programma's van Hoofdstuk 4 treedt dit gevaar niet op; dus door die recursieve programma's wordt er, bij iedere invoer, uitvoer geproduceerd.

In Hoofstuk 5 stellen we een karakterisering voor van het begrip 'wet' en onderzoeken de eigenschappen ervan. Hiermee hebben we een gereedschap ontwikkeld dat het mogelijk maakt om over wetten-in-het-algemeen stellingen te formuleren en te bewijzen. We geven daarvan een paar eenvoudige voorbeelden.

... **and order.** Getuige de resultaten van Hoofdstukken 3 en 4 is het goed mogelijk te rekenen met beperkt-recursieve programma's. In de theorie voor algemenere soorten recursie is het begrip **ordening** (engels: order), een begrip uit de wiskunde, haast onmisbaar.

In Hoofdstuk 6 onderzoeken we het ontdekken en gebruiken van wetten in situaties waarbij zo'n ordening aanwezig is. Daarmee breiden we de theorie uit tot algemenere vormen van recursie.

* * *

Ieder hoofdstuk begint met een korte technische samenvatting, en eindigt met een conclusie waarin op de behaalde resultaten teruggeblikt wordt.

Index

References

[1] L. Aiello, G. Attardi, and G. Prini. Towards a more declarative programming style. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*. North-Holland, 1978. Proceedings IFIP TC-2 Conference, St. Andrews, Canada, August 1977.

[2] M. Arbib and E. Manes. *Arrows, Structures and Functors — The Categorical Imperative*. Academic Press, 1975.

[3] R.C. Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS8810, Dept of Math and Comp Sc, University of Groningen, 1988.

[4] R.C. Backhouse. Naturality of homomorphisms. In *International Summerschool on Constructive Algorithmics*, 1989. Held on Ameland, The Netherlands, September 1989.

[5] R.C. Backhouse et al. A relational theory of datatypes. Presented at workshop [18], 1991.

[6] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[7] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[8] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984. Addendum: Ibid. 7(3):490–492, 1985.

[9] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer Verlag, 1987. Also Technical Monograph PRG–56, Oxford University, Computing Laboratory, Programming Research Group, October 1986.

[10] R.S. Bird. Lecture notes on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*. International Summer School directed by F.L. Bauer [et al.], Springer Verlag, 1989. NATO Advanced Science Institute Series (Series F: Computer and System Sciences Vol. 55).

[11] R.S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[12] R. Bos and C. Hemerik. An introduction to the category-theoretic solution of recursive domain equations. Technical Report Computing Science Notes 88/15, Eindhoven University of Technology, Eindhoven, The Netherlands, October 1988.

[13] R.M. Burstall and P.J. Landin. Programs and their proofs: An algebraic approach. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 17–43. Edinburgh University Press, 1969.

[14] G. Cousineau, P.L. Curien, and M. Mauny. The categorical abstract machine. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lect. Notes in Comp. Sc.* Springer Verlag, 1985.

[15] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11(1):1–30, 1978.

[16] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer Verlag, 1990.

[17] H. Ehrig and B. Mahr. *Fundamentals of Equational Specification 1 — equations and initial semantics.* Springer Verlag, 1985.

[18] *EURICS Workshop on Calculational Theories of Program Structure.* Utrecht University, September 1991.

[19] M.M. Fokkinga. Transformatie van specificatie tot implementatie. In *Colloquium Software Specificatie Technieken*, pages 53–86. Academic Service, Schoonhoven, The Netherlands, 1987.

[20] M.M. Fokkinga. Using underspecification in the derivation of some optimal partition algorithms. CWI, Amsterdam, March 1990.

[21] M.M. Fokkinga. An exercise in transformational programming: Backtracking and Branch-and-Bound. *Science of Computer Programming*, 16:19–48, 1991.

[22] M.M. Fokkinga, J.T. Jeuring, L. Meertens, and E. Meijer. A translation from attribute grammars to catamorphisms. *The Squiggolist*, 2(1):20–26, March 1991.

[23] M.M. Fokkinga and L. Meertens. Map-functor factorized. *The Squiggolist*, 2(1):17–19, March 1991.

[24] M.M. Fokkinga and E. Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.

[25] A.J.M. van Gasteren. *On the Shape of Mathematical Arguments.* PhD thesis, Eindhoven University of Technology, 1988.

[26] J.A. Goguen. How to prove inductive hypotheses without induction. In W. Bibel and R. Kowalski, editors, *Proc. 5th Conference on Automated Deduction*, number 87 in Lect. Notes in Comp. Sc., pages 356–373, 1980.

[27] R. Goldblatt. *Topoi — the Categorical Analysis of Logic*, volume 98 of *Studies in Logic and the Foundations of mathematics*. North-Holland, 1979.

[28] J. Gray and A. Scedrov, editors. *Categories in Computer Science and Logic*, volume 92 of *Contempory mathematics*. 1989.

[29] T. Hagino. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, 1987.

[30] T. Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, number 283 in Lect. Notes in Comp. Sc., pages 140–157. Springer Verlag, 1987.

[31] C.A.R. Hoare. Notes on an Approach to Category Theory for Computer Scientists. In M. Broy, editor, *Constructive Methods in Computing Science*, pages 245–305. International Summer School directed by F.L. Bauer [et al.], Springer Verlag, 1989. NATO Advanced Science Institute Series (Series F: Computer and System Sciences Vol. 55).

[32] J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison-Wesley, 1990.

[33] J.T. Jeuring. Deriving algorithms on binary labelled trees. In P.G.M. Apers, D. Bosman, and J. van Leeuwen, editors, *Proceedings SION Computing Science in the Netherlands*, pages 229–249, 1989.

[34] J.T. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming concepts and Methods*, pages 247–266. North-Holland, 1990.

[35] J.T. Jeuring. The derivation of on-line algorithms, with an application to finding palindromes. To appear in Algorithmica, 1992.

[36] G. Jones. Constructing the fast fourier transform by calculating with the algorithm. Contribution in [70].

[37] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in advanced mathematics*. Cambridge University Press, 1986.

[38] D.J. Lehmann. On the algebra of order — extended abstract. In *19th Symposium on the Foundations of Computer Science (FOCS)*, pages 214–220. IEEE, 1978.

[39] D.J. Lehmann. On the algebra of order. *Journal of Computer and System Sciences*, 21:1–23, 1980.

[40] D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, 14:97–139, 1981.

[41] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, 1971.

[42] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, University of Groningen, The Netherlands, 1990.

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

[44] E.G. Manes. *Algebraic Theories*, volume 26 of *Graduate Text in Mathematics*. Springer Verlag, 1987.

[45] E.G Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Text and Monographs in Computer Science. Springer Verlag, 1986.

[46] L. Meertens. First steps towards the theory of Rose Trees. Draft Report, CWI, Amsterdam, 1987.

[47] L. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker and J.C. van Vliet, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[48] L. Meertens. Constructing a calculus of programs. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, Lect. Notes in Comp. Sc., pages 66–90. Springer Verlag, 1989. LNCS 375.

[49] L. Meertens. Paramorphisms. Technical Report CS-R9005, CWI, Amsterdam, February 1990. To appear in Formal Aspects of Computing.

[50] E. Meijer. *Calculating Compilers*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1992.

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

[52] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[53] O. de Moor. Indeterminacy in optimization problems. In *International Summer School on Constructive Algorithmics*, 1989. Held on Ameland, The Netherlands, September 1989.

[54] O. de Moor. Inverses in program synthesis. In *International Summer School on Constructive Algorithmics*, 1989. Held on Ameland, The Netherlands, September 1989.

[55] O. de Moor. List partitions. In *International Summer School on Constructive Algorithmics*, 1989. Held on Ameland, The Netherlands, September 1989.

[56] O. de Moor. Categories, relations and dynamic programming. Forthcoming PhD Thesis, 1991.

[57] H. Partsch. Transformational program development in a particular problem domain. *Science of Computer Programming*, 7:99–241, 1986.

[58] R. Paterson. *Reasoning about Functional Programs.* PhD thesis, University of Queensland, Department of Computer Science, September 1987.

[59] R. Paterson. Operators. In *Lecture Notes Workshop on Constructive Algorithmics*, September 1990. Organized by CWI Amsterdam, Utrecht University, University of Nijmegen, and held at Hollum (Ameland), The Netherlands.

[60] B.C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, Ma, 1991. Originally planned for Computing Surveys. Also Tech Report CMU-CS-90-113, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 125213.

[61] D. Pitt, A. Abramsky, A. Poigné, and D. Rydeheard, editors. *Category Theory and Computer Science*, volume 240 of *Lect. Notes in Comp. Sc.* Springer Verlag, 1986.

[62] D. Pitt, A. Poigné, and D. Rydeheard, editors. *Category Theory and Computer Science*, volume 283 of *Lect. Notes in Comp. Sc.* Springer Verlag, 1987.

[63] J.C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, July 1977.

[64] J.C. Reynolds. Using category theory to design implicit conversions and generic operators. In N.D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, volume 94 of *Lect. Notes in Comp. Sc.* Springer Verlag, 1980.

[65] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory.* Prentice Hall, 1988.

[66] D.A. Schmidt. *Denotational Semantics — A Methodology for Language Development.* Allyn and Bacon, 1986. Also: Wm. C. Brown Publishers, Dubuque, Iowa, USA, 1988.

[67] B.A. Sijtsma. *Verification and Derivation of Infinite-list Programs.* PhD thesis, University of Groningen, 1988.

[68] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–785, November 1982.

[69] M. Spivey. A categorical approach to the theory of lists. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lect. Notes in Comp. Sc. Springer Verlag, 1989.

[70] *Lecture Notes International Summer School on Constructive Algorithmics*, September 1989. Organized by CWI Amsterdam, Utrecht University, University of Nijmegen, and held at Hollum (Ameland), The Netherlands.

[71] S.D. Swierstra. Virtuele datastructuren. In *Computing Science in the Netherlands 1988*, 1988.

[72] N. Verwer. Homomorphisms, factorisation and promotion. *The Squiggolist*, 1(3):45–54, 1990. Also technical report RUU-CS-90-5, Utrecht Univerity, 1990.

[73] E. Voermans and J.C.S.P. van der Woude. Relational tupling. In *Calculational Theories of Program Structure*, 1991. EURICS Workshop, organised by T.U.E. and R.U.U.

[74] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989. FPCA '89, Imperial College, London.

[75] M. Wand. Fixed point constructions in order enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.

[76] *Workshop on Constructive Algorithmics: the role of relations in program development*, September 1990. Organized by CWI Amsterdam, Utrecht University, University of Nijmegen, and held at Hollum (Ameland), The Netherlands.

[77] G.C. Wraith. Categorical datatypes — a critique of Hagino's thesis. Unpublished note, November 1988.

Curriculum Vitae

Martinus Maria Fokkinga

25 dec 1948:	geboren te Nijmegen
1961–1967:	Gymnasium-β Canisius College te Nijmegen en Franciscus College te Rotterdam
1967–1972:	Studie wiskunde (met lof) Rijksuniversiteit Utrecht
1972–1974:	Wetenschappelijk medewerker Vakgroep Informatica, afdeling Wiskunde Technische Hogeschool Delft
1972–19??:	Wetenschappelijk medewerker, hoofdmedewerker en universitair docent Faculteit Informatica Universiteit Twente
1988–1991:	vanuit de Universiteit Twente als onderzoeker gedetacheerd bij Afdeling Algoritmiek en Architectuur CWI (Centrum voor Wiskunde en Informatica), Amsterdam