Algebra of Program Termination

Henk Doornbos* Roland Backhouse†

July 30, 2001

Abstract

Well-foundedness and inductive properties of relations are expressed in terms of fixed points. A class of fixed point equations, called "hylo" equations, is introduced. A methodology of recursive program design based on the use of hylo equations is presented. Current research on generalisations of well-foundedness and inductive properties of relations, making these properties relative to a datatype, is introduced.

*EverMind, Westerkade 15/4, 9718 AS Groningen, The Netherlands

†School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, England

1 Introduction

Central to computing science is the development of practical programming methodologies. Characteristic of a programming methodology is that it involves a *discipline* designed to maximise confidence in the reliability of the end product. The discipline constrains the construction methods to those that are demonstrably simple and easy to use, whilst still allowing sufficient flexibility that the creative process of program construction is not impeded. A well-established methodology is the combined use of invariant relations and well-founded relations in the design of iterative algorithms in sequential programming (see sections 2.2 and 2.5), the constraint being the restriction of the methodology to the design of **while** statements rather than allowing arbitrary use of **goto** statements.

In this paper we develop an algebra of terminating computations based on fixed point calculus and relation algebra. We begin by formulating properties of well-foundedness (of a relation) and admitting induction in terms of fixed points. This discussion is motivated by the methodology of designing iterative algorithms mentioned above. We then explore the termination of recursive programs. Here we argue for a practical discipline of recursive programming based on a class of recursive fixed-point equations called "hylo" equations (a notion first introduced by Meijer [MFP91]). The notion of a relation admitting induction is generalised to the notion of "F-reductivity", where the parameter F captures the data structure underlying the recursion.

2 Imperative Programming and Well-founded Relations

An important application of fixed point calculus is in the study of well-founded relations. Well-founded relations are fundamental to program termination and to inductive proofs of program properties.

Before we can begin to discuss how to express well-foundedness in terms of fixed points we need to introduce relation algebra. In relation algebra, relations are not viewed as sets of pairs but just as values in (the carrier of) an algebraic system. In this way, relation algebra expresses the properties of the most fundamental operations on relations without reference to the elements being related. For example, relation algebra encapsulates the properties of the converse $R^\cup$ of relation R without reference to the so-called *pointwise* definition of $R^\cup$:

$$(x,y) \in R^\cup \equiv (y,x) \in R \quad .$$

Instead the converse operation is given an axiomatic definition, which includes for ex-

ample the property that it is its own inverse:

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

This *point-free* axiomatisation is the key to formulating the notions of well-foundedness and admitting induction in terms of fixed points.

A continuous motivational thread throughout this section will be the formulation of a methodology for the design of iterative programs in terms of relation algebra. We begin in section 2.1 with a brief introduction to relation algebra, just enough to be able to present a concise formulation of the use of invariant properties in section 2.2. This latter discussion raises several issues which motivates the introduction of the domain and division operators in section 2.3.

In section 2.4 we begin with the standard pointwise definition of well-foundedness which we then reformulate in a point-free definition. We then go on to derive an equivalent but more compact definition. The same process is followed for the notion of admitting induction. We recap the standard pointwise definition, reformulate this in a point-free manner and then derive a more compact but equivalent definition. We then discuss the equivalence of admitting induction and being well-founded.

2.1 Relation Algebra

For us, a (non-deterministic) program is an input-output relation. The convention we use when defining relations is that the input is on the right and the output on the left. The convention is thus that used in functional programming and not that used in sequential programming. For example, the relation $<$ on numbers is a program that maps a number into one smaller than itself. The function succ is the relation between natural numbers such that $m\langle succ\rangle n$ equivales $m = n+1$. It is thus the program that maps a natural number into its successor.

A relation is a set of ordered pairs. In discussions of the theory of imperative programming the "state space" from which the elements of each pair are drawn often remains anonymous. This reflects the fact that type structure is often not a significant parameter in the construction of imperative programs, in contrast to functional programs where it is pervasive. One goal here is to combine the functional and imperative programming paradigms. For this reason, we adopt a typed algebra of relations (formally an "allegory" [Fv90]). A relation is thus a triple consisting of a pair of types I and J, say, and a subset of the cartesian product $I \times J$. We write $R \in I \leftarrow J$ (read R has type I from J), the left-pointing arrow indicating that we view J as the set of all possible inputs and I as the set of possible outputs. I is called the *target* and J the *source* of the relation R, and $I \leftarrow J$ (read I from J) is called its *type*.

We write $x[\![R]\!]y$ if the pair (x,y) is an element of relation R. (As is usual in mathematics, we omit the brackets when R is denoted by a symbol and it is easy to parse the resulting expression — as in, for example, $x<y$.) We use a raised infix dot to denote relational composition. Thus $R\circ S$ denotes the composition of relations R and S (the relation defined by $x[\![R\circ S]\!]z$ equivales $\exists\langle y:: x[\![R]\!]y \wedge y[\![S]\!]z\rangle$). The composition $R\circ S$ is only defined when the source of R equals the target of S. Moreover, the target of $R\circ S$ is the target of R, and the source of $R\circ S$ is the source of S. Thus, $R\circ S \in I\leftarrow K$ if $R\in I\leftarrow J$ and $S\in J\leftarrow K$. The converse of relation R is denoted by $R^\cup$. Thus, $x[\![R^\cup]\!]y$ equivales $y[\![R]\!]x$. The type rule is that $R^\cup\in I\leftarrow J$ equivales $R\in J\leftarrow I$.

Relations of the same type are ordered by set inclusion denoted in the conventional way by the infix $\subseteq$ operator. The relations of a given type $I\leftarrow J$ form a complete lattice under this ordering. The smallest relation of type $I\leftarrow J$ is the empty relation, denoted here by $\perp\!\!\!\perp_{I\leftarrow J}$, and the largest relation of type $I\leftarrow J$ is the universal relation, which we denote by $\top\!\!\!\top_{I\leftarrow J}$. (We use this notation for the empty and universal relations because the conventional notation $\top$ for the universal relation is easily confused with T, a sans serif letter T, particularly in hand-written documents.) The union and intersection of two relations R and S of the same type are denoted by $R\cup S$ and $R\cap S$, respectively.

Because relations are sets, we have the shunting rule

$$R\cap S\subseteq T \;\equiv\; S\subseteq \neg R\cup T$$

where $\neg R$ denotes the complement of relation R. The only use we make of this rule here is the fact that relations of type $I\leftarrow I$ form a completely distributive lattice and thus a regular algebra. We use this fact when we exploit the unique extension property of regular algebras in the identification of fixed point equations that define a relation uniquely.

For each set I there is an identity relation on I which we denote by id_I. Thus $id_I\in I\leftarrow I$. Relations of type $I\leftarrow I$ contained in the identity relation of that type will be called *coreflexives*. (The terminology *partial identity relation* and *monotype* is also used.) By convention, we use R, S, T to denote arbitrary relations and A, B and C to denote coreflexives. A coreflexive A thus has the property that if $x[\![A]\!]y$ then $x=y$. Clearly, the coreflexives of type $I\leftarrow I$ are in one to one correspondence with the subsets of I; we shall exploit this correspondence by identifying subsets of a set I with the coreflexives of type $I\leftarrow I$. Specifically, by an abuse of notation, we write $x\in A$ for $x[\![A]\!]x$ (on condition that A is a coreflexive). We also identify coreflexives with predicates, particularly when discussing induction principles (which are traditionally formulated in terms of predicates rather than sets). So we shall say "x has property A" meaning formally that $x[\![A]\!]x$. Continuing this abuse of notation, we use $\sim A$ to denote the coreflexive having the same type as A and containing just those elements not in A. Thus, $x[\![\sim A]\!]y$ equivales the conjunction of $x\in I$ (where A has type $I\leftarrow I$) and $x=y$

and not $x[\![A]\!]x$. We also sometimes write I where id_I is meant. (This fits in with the convention in category theory of giving the same name to that part of a functor which maps objects to objects and that part which maps arrows to arrows.) A final, important remark about coreflexives is that their composition coincides with their intersection. That is, for coreflexives A and B, $A \circ B = A \cap B$.

We use an infix dot to denote function application. Thus $f.x$ denotes application of function f to argument x. Functions are particular sorts of relations; a relation R is *functional* if $y[\![R]\!]x$ and $z[\![R]\!]x$ together imply that $y = z$. If this is the case we write $R.x$ for the unique y such that $y[\![R]\!]x$. Note that functionality of relation R is equivalent to the property $R \circ R^{\cup} \subseteq id_I$ where I is the target of R. We normally use f, g and h to denote functional relations.

Dual to the notion of functionality of a relation is the notion of injectivity. A relation R with source J is *injective* if $R^{\cup} \circ R \subseteq id_J$. Which of the properties $R \circ R^{\cup} \subseteq id_I$ or $R^{\cup} \circ R \subseteq id_J$ one calls "functional" and which "injective" is a matter of interpretation. The choice here fits in with the convention that input is on the right and output on the left. More importantly, it fits with the convention of writing $f.x$ rather than say x^f (that is the function to the left of its argument). A sensible consequence is that type arrows point from right to left.

2.2 Imperative Programming

In this section we introduce the derivation of repetitive statements using invariant relations. The section contains just an outline of the methodology expressed in relation algebra. For extensive introductions see (for example) [Gri81, Bac86]. At the end of the section we identify a need to delve deeper into relation algebra, thus motivating the section which follows.

Given a (non-trivial) specification, X, the key to constructing a loop implementing X is the invention of an invariant, Inv. The invariant is chosen in such a way that it satisfies three properties. First, the invariant can be "established" by some initialisation $Init$. Second, the combination of the invariant and some termination $Term$ satisfies the specification X. Third, the invariant is "maintained by" some loop body $Body$ whilst making progress towards termination.

These informal requirements can be made precise in a very concise way. The three components Inv, $Init$ and $Term$ are all binary relations on the state space, just like the specification X. They are so-called input-output relations.

"Establishing" the invariant is the requirement that

$$Init \subseteq Inv \quad .$$

In words, any value w' related to input value w by Init is also related by the invariant relation to w.

That the combination of the termination and invariant satisfies the specification X is the requirement that

$$\mathit{Term} \circ \mathit{Inv} \subseteq X \quad .$$

This is the requirement that for all output values w' and input values w,

$$\forall \langle v\colon w'[\![\mathit{Term}]\!]v \wedge v[\![\mathit{Inv}]\!]w\colon w'[\![X]\!]w \rangle$$

(Here we see the convention of placing input values on the right and output values on the left.)

Finally, that the invariant is maintained by the loop body is expressed by

$$\mathit{Body} \circ \mathit{Inv} \subseteq \mathit{Inv}$$

Pointwise this is

$$\forall \langle w', v, w\colon\ w'[\![\mathit{Body}]\!]v \wedge v[\![\mathit{Inv}]\!]w\colon\ w'[\![\mathit{Inv}]\!]w \rangle \quad .$$

So Body maps values v related by the invariant Inv to w to values w' that are also related by Inv to w.

Together these three properties guarantee that

$$\mathit{Term} \circ \mathit{Body}^* \circ \mathit{Init} \subseteq X \quad .$$

That progress is made is the requirement that the relation Body be well-founded. (This we will return to shortly.)

As an example, consider the classic problem of finding the greatest common divisor (abbreviated gcd) of two positive numbers x and y. The state space of the program is $\mathit{Int} \times \mathit{Int}$. The specification, invariant, initialisation and termination are thus binary relations on this set. The specification, X, is simply

$$x' = y' = \mathrm{gcd}.(x,y) \quad .$$

Here priming x and y is a commonly used convention for abbreviating the definition of a relation between the pair of output values x' and y', and the pair of input values x and y. More formally, X is the relation

$$\{x, y, x', y'\colon x' = y' = \mathrm{gcd}.(x,y)\colon ((x',y'),(x,y))\} \quad .$$

The convention is that the definition

$$\{x, y, x', y'\colon p.(x,y,x',y')\colon ((x',y'),(x,y))\}$$

is abbreviated to

$$p.(x,y,x',y') \quad ,$$

the primes indicating the correspondence between input and output variables. Using this convention, the invariant is the relation

$$gcd.(x',y') = gcd.(x,y)$$

and the initialisation is the identity relation

$$x' = x \wedge y' = y \quad .$$

(The initialisation is thus implemented by skip , the do-nothing statement.) The termination is a subset of the identity relation on the state space. It is the relation

$$x' = x = y' = y \quad .$$

The composition of the termination relation and the invariant is thus the relation

$$x' = y' \wedge gcd.(x',y') = gcd.(x,y)$$

which, since $gcd.(x',x')$ equals x', is identical to the specification X. The loop body in Dijkstra's well-known guarded command solution to this problem is the union of two relations, the relation

$$x < y \wedge x' = x \wedge y' = y - x$$

and the relation

$$y < x \wedge y' = y \wedge x' = x - y \quad .$$

Exercise 1 Identify X, Inv, $Init$, $Body$ and $Term$ in the language recognition program discussed in the chapter on Galois Connections and Fixed Point Calculus.

□

2.3 Domains and Division

2.3.1 Domains

Our account of invariants is not yet complete. The relationship between the specification X and $Term \circ Body^* \circ Init$ is containment not equality, and may indeed be a proper superset relation. Not every subset of the specification will do, however. An additional requirement is that the input-output relation computed by the program is total on all input values. Formally this is a requirement on the so-called "right domain" of the

computed input-output relation. Right domains are also relevant if we are to relate our account of invariants to the implementation of loops by a **while** statement. Recall that $Body$ is the body of the loop, and $Term$ terminates the computation. The implementation of $Term \circ Body^*$ by a **while** statement demands that both of these relations are partial and, more specifically, that their right domains are complementary.

The *right domain* of a relation R is, informally, the set of input values that are related by R to at least one output value. Formally, the right domain $R^>$ of a relation R of type $I \leftarrow J$ is a coreflexive of type $J \leftarrow J$ satisfying the property that

$$\forall \langle A{:}\ A \subseteq id_J{:}\ R \circ A = R \equiv R^> \subseteq A \rangle \quad .$$

Given a coreflexive A, $A \subseteq id_J$, the relation $R \circ A$ can be viewed as the relation R restricted to inputs in the set A. Thus, in words, the right domain of R is the least coreflexive A that maintains R when R is restricted to inputs in the set A.

Note that the right domain should not be confused with the source of the relation. The source expresses the set of input values of interest in the context of the application being considered whereas the right domain is the set of input values over which the relation is defined. In other words, we admit the possibility of *partial* relations. Formally, a relation R of type $I \leftarrow J$ is *total* if $R^>$ is id_J, otherwise it is partial. Similarly the target should not be confused with the left domain of a relation. A relation R of type $I \leftarrow J$ is *surjective* if $R^<$ is id_I.

Returning to loops, the requirement is that the right domain of $Term$ is the complement of the right domain of $Body$. Letting b denote the right domain of $Body$ and $\sim b$ its complement (thus $b \cup \sim b = id$ and $b \cap \sim b = \perp\!\!\!\perp$) we thus have

$$Term = Term \circ \sim b \quad \text{and} \quad Body = Body \circ b \quad .$$

As a consequence,

$$Term \circ Body^* \circ Init = Term \circ \sim b \circ (Body \circ b)^* \circ Init \quad .$$

The statement **while** b **do** $Body$ is the implementation of $\sim b \circ (Body \circ b)^*$ in that the latter is the least solution of the equation

$$X{::}\ X = \sim b \cup X \circ Body \circ b$$

and executing this equation is equivalent to executing the program

$$X = \textbf{if } b \textbf{ then } Body; X \quad .$$

We continue this discussion in section 2.5.

2.3.2 Division

The body of a loop should maintain the loop invariant. Formally, the requirement is that $Body \circ Inv \subseteq Inv$. In general, for relations R of type $I \leftarrow J$ and T of type $I \leftarrow K$ there is a relation $R \backslash T$ of type $J \leftarrow K$ satisfying the Galois connection, for all relations S,

$$R \circ S \subseteq T \equiv S \subseteq R \backslash T \quad .$$

The operator $\backslash$ is called a *division* operator (because of the similarity of the above rule to the rule of division in ordinary arithmetic). The relation $R \backslash T$ is called a *residual* or a *factor* of the relation T. Relation $R \backslash T$ holds between output value w' and input value w if and only if

$$\forall \langle v:\ v [\![R]\!] w' :\ v [\![T]\!] w \rangle \quad .$$

Applying this Galois connection, the requirement on $Body$ is thus equivalent to

$$Inv \subseteq Body \backslash Inv \quad ,$$

the pointwise formulation of which is

$$\forall \langle w', w:\ w' [\![Inv]\!] w:\ \forall \langle w'' : w'' [\![Body]\!] w' : w'' [\![Inv]\!] w \rangle \rangle \quad .$$

The relation $Body \backslash Inv$ corresponds to what is called the *weakest prespecification* of Inv with respect to $Body$ in the more usual predicate calculus formulations of the methodology [HH86]. The *weakest liberal precondition* operator will be denoted here by the symbol "$\backslash\!\!\cdot$". Formally, if R is a relation of type $I \leftarrow J$ and A is a coreflexive of type $I \leftarrow I$ then $R \backslash\!\!\cdot A$ is a coreflexive of type $J \leftarrow J$ characterised by the property that, for all coreflexives B of type $J \leftarrow J$,

$$(2) \qquad (R \circ B)^{<} \subseteq A \equiv B \subseteq R \backslash\!\!\cdot A \quad .$$

(If we interpret the coreflexive A as a predicate p on the type I, then $R \backslash\!\!\cdot A$ is the predicate q such that

$$q.w \equiv \forall \langle w' : w' [\![R]\!] w : p.w \rangle \quad .$$

It is the weakest condition q on input values w that guarantees that all output values w' that are R-related to w satisfy the predicate p.)

The operator $\backslash\!\!\cdot$ plays a very significant role in what is to follow. For this reason it is useful to have a full and intimate understanding of its algebraic properties. This, however, is not the place to develop that understanding and we make do with a summary of the most frequently used properties.

First note that the function $(R \backslash\!\!\cdot)$, being an upper adjoint, distributes over arbitrary meets of coreflexives. Because meet on coreflexives coincides with composition it follows

that $R\backslash$ distributes over composition: $R\backslash(A\circ B) = (R\backslash A)\circ(R\backslash B)$. This corresponds to the fact that weakest liberal precondition operator associated with a statement R is universally conjunctive. From (2) we obtain the *cancellation* property:

(3) $\quad (R\circ R\backslash B)^{<} \subseteq B \quad .$

Often this property is used in a slightly different form, namely:

(4) $\quad R\circ R\backslash B \subseteq B\circ R \quad .$

Both (3) and (4) express that program R produces a result from set B when started in a state satisfying $R\backslash B$. If R is a function then $R\backslash A$ can be expressed without recourse to the left-domain operator. Specifically, we have for function f:

(5) $\quad f\backslash A = f^{\cup}\circ A\circ f \quad .$

A full discussion, including all the properties used here, can be found in [BW93].

2.4 Well-Foundedness Defined

Expressed in terms of points, a relation R is said to be *well-founded* if there are no infinite chains x_0, x_1, ... such that $x_{i+1}[\![R]\!]x_i$ for all i, $i \geq 0$. A relation R is thus *not* well-founded if there is a set A such that

$$A \neq \phi \wedge \forall\langle x: x\in A: \exists\langle y: y\in A: y[\![R]\!]x\rangle\rangle \quad .$$

Noting that $\exists\langle y: y\in A: y[\![R]\!]x\rangle \equiv x \in (A\circ R)^{>}$ this definition converts directly into the following point-free form.

Definition 6 (Well-founded) Relation R is said to be *well-founded* if and only if it satisfies

$$\forall\langle A:\ A\subseteq I:\ A\subseteq \perp\!\!\!\perp \Leftarrow A\subseteq (A\circ R)^{>}\rangle \quad .$$

□

The connection between well-foundedness and fixed points is the following.

Theorem 7 Relation R is well-founded equivales

$$\nu\langle A: A\subseteq I: (A\circ R)^{>}\rangle = \perp\!\!\!\perp \quad .$$

Proof For arbitrary monotonic function f we have:

$$
\begin{array}{ll}
 & \nu f = \bot\!\!\bot \\
\Leftarrow & \quad \{ \quad \text{reflexivity of } \subseteq,\ \bot\!\!\bot \text{ is the least element} \quad \} \\
 & \forall \langle X ::\ X \subseteq \bot\!\!\bot \Leftarrow X \subseteq \nu f \rangle \\
\Leftarrow & \quad \{ \quad \text{fixed point induction} \quad \} \\
 & \forall \langle X ::\ X \subseteq \bot\!\!\bot \Leftarrow X \subseteq f.X \rangle \quad .
\end{array}
$$

The corollary follows by instantiating f to $\langle A : A \subseteq I : (A \circ R)^{>} \rangle$.
□

Characteristic of definition 6 is that it is a rule for establishing when a set represented by a coreflexive A, $A \subseteq I$, is empty. In the following theorem we replace sets by arbitrary relations. This has the advantage that we can then immediately exploit the unique extension property of a regular algebra.

Theorem 8 For arbitrary relation $R \in I \leftarrow I$,

$$(\nu \langle X :: X \circ R \rangle)^{>} \ = \ \nu \langle A : A \subseteq I : (A \circ R)^{>} \rangle \quad .$$

Hence R is well-founded if and only if it satisfies

$$\nu \langle X :: X \circ R \rangle = \bot\!\!\bot \quad .$$

(Here the dummy X ranges over all relations of type $I \leftarrow I$.)

Proof We shall only sketch the proof since we have not discussed the algebraic properties of the right domain operator in sufficient detail to give a completely formal proof. (For such a proof see [DBvdW97].)

At first sight it would seem that a simple application of the fusion theorem would suffice. This is not the case, however, because the right domain operator is a lower adjoint in a Galois connection, not an upper adjoint as is required to apply fusion.

The key to the proof is to observe that $\nu \langle X :: X \circ R \rangle$ is a so-called *right condition*. That is,

$$\nu \langle X :: X \circ R \rangle \ = \ \top\!\!\top \circ \nu \langle X :: X \circ R \rangle \quad .$$

(This is easily proved by a mutual inclusion argument.) This suggests the use of the fusion theorem to prove that

$$\iota \,.\, \nu \langle p :\ p = \top\!\!\top \circ p :\ p \circ R \rangle = \nu \langle X :: X \circ R \rangle$$

where ι denotes the function that embeds the set of right conditions of type $I \leftarrow I$ into the set of relations of type $I \leftarrow I$. (Embedding functions between complete lattices are both upper and lower adjoints so there is no difficulty in applying the fusion theorem.) The

proof is now completed by using the fact that the two functions $\langle p{:}\ p = \top\!\!\top \circ p{:}\ p^{>}\rangle$ and $\langle A{:}\ A \subseteq I{:}\ \top\!\!\top \circ A\rangle$ are inverse lattice isomorphisms between the lattice of right conditions and the lattice of coreflexives of type $I \leftarrow I$. (This is an application of the unity-of-opposites theorem.) Thus, using the fusion theorem once again,

$$\nu\langle A::\ (A \circ R)^{>}\rangle = (\nu\langle p::\ p \circ R\rangle)^{>}$$

and

$$\top\!\!\top \circ \nu\langle A::\ (A \circ R)^{>}\rangle = \nu\langle p::\ p \circ R\rangle \quad .$$

From this it follows that

$$(\nu\langle X::\ X \circ R\rangle)^{>} \ = \ \nu\langle A{:}\ A \subseteq I{:}\ (A \circ R)^{>}\rangle \quad .$$

Now,

$$\begin{array}{ll} & R \text{ is well-founded} \\ \equiv & \quad \{ \quad \text{definition} \quad \} \\ & \nu\langle A{:}\ A \subseteq I{:}\ (A \circ R)^{>}\rangle = \bot\!\!\bot \\ \equiv & \quad \{ \quad \text{above} \quad \} \\ & (\nu\langle X::\ X \circ R\rangle)^{>} = \bot\!\!\bot \\ \equiv & \quad \{ \quad \text{domains} \quad \} \\ & \nu\langle X::\ X \circ R\rangle = \bot\!\!\bot \quad . \end{array}$$

□

Corollary 9 Relation R is well-founded equivales

$$\forall\langle S, T::\ T = S \cup T \circ R \ \equiv \ T = S \circ R^*\rangle$$

Proof A relation algebra is a regular algebra. So the unique extension property holds with the product operator instantiated to relational composition and the addition operator instantiated to set union.
□

Having expressed well-foundedness in terms of fixed points it is now possible to apply fixed-point calculus to deduce some of its properties. The following is elementary (even without the use of fixed points!) but needs to be stated because of its frequent use.

Lemma 10 If relation R is well-founded and $S \subseteq R$ then S is well-founded.
□

Fixed point calculus gives an easy proof of the following more interesting theorem.

Theorem 11 For all R, that R is well-founded equivales that R^+ is well-founded.

Proof We prove the stronger theorem that

$$\nu\langle X :: X \circ R\rangle = \nu\langle X :: X \circ R^+\rangle \quad .$$

The inclusion $\nu\langle X :: X \circ R\rangle \subseteq \nu\langle X :: X \circ R^+\rangle$ is immediate from monotonicity of ν and the fact that $R \subseteq R^+$. For the other inclusion, we calculate:

$$\begin{array}{ll}
& \nu\langle X :: X \circ R^+\rangle \subseteq \nu\langle X :: X \circ R\rangle \\
\Leftarrow & \quad \{ \quad \text{fixed point induction} \quad \} \\
& \nu\langle X :: X \circ R^+\rangle \subseteq \nu\langle X :: X \circ R^+\rangle \circ R \\
\equiv & \quad \{ \quad R^+ = R \circ R^* \quad \} \\
& \nu\langle X :: X \circ R^+\rangle \subseteq \nu\langle X :: X \circ R \circ R^*\rangle \circ R \\
\equiv & \quad \{ \quad \text{rolling rule} \quad \} \\
& \nu\langle X :: X \circ R^+\rangle \subseteq \nu\langle X :: X \circ R^* \circ R\rangle \\
\equiv & \quad \{ \quad R^+ = R^* \circ R \quad \} \\
& \text{true} \quad .
\end{array}$$

□

This concludes this section. With dummies A and p ranging over coreflexives and right conditions, respectively, and X, S and T over relations, we have established the equivalence of the properties:

- R is well-founded.
- $\nu\langle A :: (A \circ R)^{>}\rangle = \bot\!\!\!\bot$.
- $\nu\langle p :: p \circ R\rangle = \bot\!\!\!\bot$.
- $\nu\langle X :: X \circ R\rangle = \bot\!\!\!\bot$.
- R^+ is well-founded.
- $\forall\langle S, T ::\ T = S \cup T \circ R \ \equiv\ T = S \circ R^*\rangle$.

Exercise 12 We saw above that $\nu\langle X :: X \circ R\rangle$ is a right condition. *What is the interpretation of this right condition as a set?*

□

Exercise 13 The standard technique for proving termination of a loop statement or a recursive definition in a program is to use a *bound function*. That is, one defines a function from the state space to a set on which a well-founded relation is defined. Most commonly the set is the set of natural numbers, and one proves that the body of the loop statement or recursive definition strictly reduces the value of the bound function.

Suppose the body of the loop statement is given by relation S, the bound function is f and the well-founded relation is R. Then the technique amounts to proving that if $x[\![S]\!]y$ then $f.x\,[\![R]\!]\,f.y$. That is, $S \subseteq f^{\cup} \circ R \circ f$. In view of lemma 10, the validity of the use of bound functions is justified by the following theorem: If R is a relation and f a functional relation such that R is well-founded, then relation $f^{\cup} \circ R \circ f$ is well-founded as well.

Prove this theorem.

Hint: as in the proof of theorem 11 one can make a more general statement relating $\nu\langle X :: X \circ R\rangle$ and $\nu\langle X :: X \circ f^{\cup} \circ R \circ f\rangle$.

□

Exercise 14 *Show that if* R *is well-founded then* $R^{+} \cap I \subseteq \perp\!\!\!\perp$. (So no non-empty subset of a well-founded relation is reflexive.)

Under what conditions is the well-foundedness of R *equivalent to* $R^{+} \cap I \subseteq \perp\!\!\!\perp$ *? Provide examples where possible.*

□

2.5 Totality of while statements

We are now in a position to complete our discussion of the construction of **while** statements using loop invariants.

Recall that $Body$ is the body of the loop, and $Term$ terminates the computation. The well-foundedness of $Body$ guarantees that the execution of the **while** statement will always terminate. It also guarantees that the implementation is total, provided that $Term$ and $Body$ have complementary right domains, and the initialisation $Init$ is total. Specifically, we have:

$$
\begin{array}{ll}
 & (Term \circ Body^{*} \circ Init)^{>} = I \\
\equiv & \quad \{ \quad \text{domain calculus} \quad \} \\
 & ((Term \circ Body^{*})^{>} \circ Init)^{>} = I \\
\Leftarrow & \quad \{ \quad \text{by assumption, } Init \text{ is total, i.e. } Init^{>} = I \quad \} \\
 & (Term \circ Body^{*})^{>} = I
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \quad (\mathrm{Term} \circ \mathrm{Body}^*)^> \text{ is the unique solution of the equation} \\
& \quad\; A{::}\ A = \mathrm{Term}^> \cup (A \circ \mathrm{Body})^> \quad \} \\
& I = \mathrm{Term}^> \cup (I \circ \mathrm{Body})^> \\
\equiv & \{ \quad \text{by assumption,} \\
& \quad\; \mathrm{Term} \text{ and } \mathrm{Body} \text{ have complementary right domains.} \\
& \quad\; \text{In particular, } I = \mathrm{Term}^> \cup \mathrm{Body}^> \quad \} \\
& \text{true} \quad .
\end{array}
$$

The penultimate step needs further justification. The claim is that the equation

$$A{::}\ A = \mathrm{Term}^> \cup (A \circ \mathrm{Body})^>$$

has a unique solution provided that Body is well-founded. This is easily derived from (9). Indeed, for all coreflexives A,

$$
\begin{array}{ll}
& A = \mathrm{Term}^> \cup (A \circ \mathrm{Body})^> \\
\equiv & \{ \quad \text{domain calculus.} \\
& \quad\; \text{Specifically, } (\top\!\!\top \circ A)^> = A \text{ and } \top\!\!\top \circ R = \top\!\!\top \circ R^> \quad \} \\
& \top\!\!\top \circ A \ = \ \top\!\!\top \circ \mathrm{Term} \cup \top\!\!\top \circ A \circ \mathrm{Body} \\
\equiv & \{ \quad \mathrm{Body} \text{ is well-founded, (9)} \quad \} \\
& \top\!\!\top \circ A = \top\!\!\top \circ \mathrm{Term} \circ \mathrm{Body}^* \\
\equiv & \{ \quad \text{domain calculus (as above)} \quad \} \\
& A = (\mathrm{Term} \circ \mathrm{Body}^*)^> \quad .
\end{array}
$$

That is, $(\mathrm{Term} \circ \mathrm{Body}^*)^>$ is the unique solution of the above equation in A.

2.6 Induction Principles

Dual to the notion of well-foundedness is the notion of admitting induction. This section formulates the latter notion in terms of fixed points and then shows that well-foundedness and admitting induction are equivalent.

A relation R is said to *admit induction* if the following schema can be used to establish that property P holds everywhere: prove, for all y, that the induction hypothesis $\forall\langle x: x[\![R]\!]y: P.x\rangle$ implies $P.y$. That is, expressed in terms of points, R *admits induction* iff

$$\forall\langle y:: P.y\rangle \Leftarrow \forall\langle y:: \forall\langle x: x[\![R]\!]y: P.x\rangle \Rightarrow P.y\rangle \quad .$$

The subterm

$$\forall\langle x: x[\![R]\!]y: P.x\rangle$$

in this formula is called the *induction hypothesis* while the proof of the subterm

$$\forall\langle y:: \forall\langle x: x[\![R]\!]y: P.x\rangle \Rightarrow P.y\rangle$$

the *induction step*. For instance, replacing R by the less-than relation $(<)$ on natural numbers and the dummies x and y by m and n, the less-than relation admits induction iff

$$\forall\langle n:: P.n\rangle \Leftarrow \forall\langle n:: \forall\langle m: m<n: P.m\rangle \Rightarrow P.n\rangle \quad .$$

This is indeed the case since the above is the statement of the principle of strong mathematical induction. The induction hypothesis is $\forall\langle m: m<n: P.m\rangle$; the induction step is the proof that assuming the truth of the induction hypothesis one can prove $P.n$. That the less-than relation admits induction means that from the proof of the induction step one can infer that $P.n$ holds for all n.

Less directly but nevertheless straightforwardly, replacing R by the predecessor relation on natural numbers, i.e. $x[\![R]\!]y \equiv y = x+1$, and simplifying using the fact that no number is a predecessor of 0, one obtains the principle of simple mathematical induction: with n ranging over the natural numbers,

$$\forall\langle n:: P.n\rangle \;\Leftarrow\; P.0 \wedge \forall\langle n:: P.n \Rightarrow P.(n+1)\rangle \quad .$$

Thus the predecessor relation on natural numbers admits induction. Note that in this case the proof of $P.0$ is called the *basis* of the proof by induction and the proof of $P.n \Rightarrow P.(n+1)$, for all n, the induction step.

The pointwise definition of "admits induction" given above is in terms of predicates. Because we want to arrive at a definition in terms of relations we first reformulate it in terms of sets. So we define: relation R *admits induction* if and only if:

$$(15) \qquad \forall\langle y:: y\in A\rangle \Leftarrow \forall\langle y:: \forall\langle x: x[\![R]\!]y: x\in A\rangle \Rightarrow y\in A\rangle \quad .$$

To arrive at a definition without dummies we first notice that $\forall\langle y:: y\in A\rangle$, the (understood) domain of y being I, can be rewritten as $I\subseteq A$. Furthermore, we see that the expression in the domain of the antecedent, $\forall\langle x: x[\![R]\!]y: x\in A\rangle$, is just $y \in R\backslash A$. So (15) can be drastically simplified to

$$(16) \qquad I\subseteq A \Leftarrow R\backslash A\subseteq A \quad ,$$

for all coreflexives A of type $I\leftarrow I$.

According to the terminology introduced above, $R\backslash A$ is the induction hypothesis, whilst a proof of $R\backslash A\subseteq A$ is the induction step.

This then is the definition of "admits induction".

Definition 17 The relation R is said to *admit induction* if and only if it satisfies

$$\forall\langle A{:}\ A \subseteq I{:}\ I \subseteq A \Leftarrow R\backslash A \subseteq A\rangle \quad .$$

Equivalently, R is said to admit induction if and only if

$$\mu\langle A{:}\, A \subseteq I{:}\, R\backslash A\rangle = \top\top \quad .$$

□

Just as for well-foundedness, we propose a definition in which the type difference between the variables is removed.

Theorem 18 That relation R admits induction equivales $\mu\langle X{::}\, R\backslash X\rangle = \top\top$.

□

We omit the proof as it is essentially dual to the proof of theorem 8.

Exercise 19 Definition 17 draws attention to the coreflexive $\mu(R\backslash)$: if relation R admits induction then the set corresponding to $\mu(R\backslash)$ is the universe over which relation R is defined. By restricting the domain of any relation R it is always possible to obtain a relation that admits induction. Specifically, for any relation R, the relation $R \circ \mu(R\backslash)$ admits induction. *Prove this theorem.*

□

2.7 Admits-induction Implies Well-Founded

Now that we have seen several equivalent definitions of well-founded it is time to explore its relationship to admitting induction. The following lemma is the key insight.

Lemma 20 $\nu\langle T{::}\, T \circ R\rangle \circ \mu\langle T{::}\, R\backslash T\rangle = \bot\bot$.

Proof The form of the theorem suggests that we try to apply μ-fusion. Of course we then have to find a suitable function f such that $\mu f = \bot\bot$. The identity function is one possibility and, as it turns out, is a good choice. However, since we want to demonstrate how good use of the calculational technique can avoid the need to make guesses of this nature, we construct f. We have, for all X,

$$\begin{array}{ll} & X \circ \mu\langle T{::}\, R\backslash T\rangle = \bot\bot \\ \equiv & \quad \{ \quad \text{introduce } f \text{ such that } \bot\bot = \mu f \quad \} \\ & X \circ \mu\langle T{::}\, R\backslash T\rangle \subseteq \mu f \end{array}$$

$\Leftarrow$ { basic fusion theorem }

$\forall\langle T::\ X\circ R\backslash T \subseteq X\circ f.T\rangle$

$\equiv$ { choose for f, $f.T = T$, noting that indeed $\bot\!\!\bot = \mu\langle T::T\rangle$.
factor cancellation: specifically, $R\circ R\backslash T \subseteq T$ }

$X \subseteq X\circ R$

$\Leftarrow$ { definition of $\nu\langle T::T\circ R\rangle$ }

$X = \nu\langle T::T\circ R\rangle$.

□

Theorem 21 If R admits induction then R is well-founded.

Proof If R admits induction then, by definition, $\mu\langle T::R\backslash T\rangle = \top\!\!\top$. So, by lemma 20, $\nu\langle T::T\circ R\rangle \circ \top\!\!\top = \bot\!\!\bot$. But then, since $I \subseteq \top\!\!\top$, $\nu\langle T::T\circ R\rangle \subseteq \bot\!\!\bot$. By theorem 8 we have thus established that R is well-founded.
□

Exercise 22 One might suppose that an argument dual to the above leads to a proof that well-foundedness implies admits-induction. Unfortunately this is not the case: a true inverse, viz. complementation, is needed to do that. To prove the theorem using the techniques developed here it suffices to know that $R\backslash S = \neg(R^{\cup}\circ\neg S)$. (We haven't given enough information about relation algebra for you to verify this fact within the algebra. A pointwise verification can, of course, be given instead.) This fact can then be used to construct a function f such that $\nu\langle T::T\circ R\rangle = f.\mu\langle T::R\backslash T\rangle$. μ-fusion should be used bearing in mind the Galois connection $\neg R \subseteq S \ \equiv\ R \supseteq \neg S$ and being particularly careful about the reversal of the ordering relation. Having constructed f it is then straightforward to establish the equivalence between the two notions.

Prove that well-foundedness admits induction along the lines outlined above.

In general, the right condition $\nu\langle T::T\circ R\rangle$ can be interpreted as the set of all points from which an infinite R-chain begins.

What is the interpretation of $\mu\langle T::R\backslash T\rangle$?

□

3 Hylo Equations

In this section we introduce a methodology for the design of recursive programs. The methodology is based on constraining the recursion to a particular form of fixed point

equation, called a "hylo" equation, rather than allowing arbitrary recursion (which has been called the **goto** of functional programming). The methodology generalises the methodology for designing **while** statements by introducing a datatype as an additional parameter in the design process. (In the case of **while** statements the datatype is just the set of natural numbers, in the case of a divide-and-conquer algorithm the datatype is a tree structure.)

A hylo equation comprises three elements, a "relator" F (which is a function from relations to relations), and two relations, one of which is an "F-algebra" and the other is an "F-coalgebra". The complete definition is given in section 3.1. Section 3.2 gives a number of examples of programs that take the form of a hylo equation. It is shown that programs defined by structural or primitive recursion are instances of hylo programs as well as several standard sorting algorithms and other programs based on a divide-and-conquer strategy. The goal in this section is, of course, to demonstrate that restricting the design methodology to hylo programs still allows sufficient room for creativity. Sections 3.3 and 3.4 introduce an important fixed-point theorem which formally relates hylo equations with the use of an intermediate or "virtual" data structure. Understanding this theorem is crucial to understanding the methodology of designing hylo programs. The final section, section 3.5 is about generalising notions of well-foundedness and inductivity to take into account the intermediate data structure implicit in any hylo program.

3.1 Relators and Hylos

A hylo equation comprises three elements, a so-called "relator" and two relations. The notion of relator plays the same role in relation algebra as the notion of "functor" in the category of functions and sets.

Functors are relevant to functional programming because they correspond to type constructors. The canonical example is $List$, which is an endofunctor on the category Fun. The object part of the functor $List$ is the mapping from types (sets) to types. (For example $List.\mathbb{N}$, lists of natural numbers, is the result of applying $List$ to $\mathbb{N}$.) The arrow part of the functor $List$ is the function known as `map` to functional programmers. If $f \in I \leftarrow J$ then $map.f \in List.I \leftarrow List.J$ is the function that applies function f to each element in a list of Js to create a list of Is of the same length. It is a general fact that parameterised datatypes (of which $List$ is an example) define functors. The object part of the functor is the mapping from types to types and the arrow part is the "map" operation that applies a given function to every value stored in an instance of the datatype.

Rather than constrain ourselves to the design of functional programs, we consider programs involving relations as well. (The reasons are obvious: doing so means that we may allow non-determinism in our programs and do not have to make an arbitrary distinction between specifications —which typically involve an element of non-determinism— and

implementations. Also, as the theory below shows, there is no good reason for not extending the discussion to include relations.) But the categorical notion of functor is too weak to describe type constructors in the context of a relational theory of datatypes. The notion of an "allegory" [Fv90] extends the notion of a category in order to better capture the essential properties of relations, and the notion of a "relator" [BBM+91, BW93] extends the notion of a functor in order to better capture the relational properties of datatype constructors.

Formally an *allegory* is a category such that, for each pair of objects A and B, the class of arrows of type $A \leftarrow B$ forms an ordered set. In addition there is a converse operation on arrows and a meet (intersection) operation on pairs of arrows of the same type. These are the minimum requirements in order to be able to state the algebraic properties of the converse operation. For practical purposes more is needed. A *locally-complete, tabulated, unitary, division allegory* is an allegory such that, for each pair of objects A and B, the partial ordering on the set of arrows of type $A \leftarrow B$ is complete ("locally-complete"), the division operators introduced in section 2.3.2 are well-defined ("division allegory"), the allegory has a unit (which is a relational extension of the categorical notion of a unit — "unitary") and, finally, the allegory is "tabulated". We won't go into the details of what it means to be "tabulated" but, basically, it means that every arrow in the allegory can be represented by a pair of arrows in the underlying map category (i.e. by a pair of functions) and captures the fact that relations are subsets of the cartesian product of a pair of sets. (Tabularity is vital because it provides the link between categorical properties and their extensions to relations.)

A suitable extension to the notion of functor is the notion of a "relator". A *relator* is a functor whose source and target are both allegories —remember that an allegory is a category— that is monotonic with respect to the subset ordering on relations of the same type and commutes with converse. Thus, a *relator* F is a function to the objects of an allegory $\mathcal{C}$ from the objects of an allegory $\mathcal{D}$ together with a mapping to the arrows (relations) of $\mathcal{C}$ from the arrows of $\mathcal{D}$ satisfying the following properties:

(23) $\quad F.R \in F.I \xleftarrow{\mathcal{C}} F.J$ whenever $R \in I \xleftarrow{\mathcal{D}} J$.

(24) $\quad F.R \circ F.S = F.(R \circ S)$ for each R and S of composable type,

(25) $\quad F.id_A = id_{F.A}$ for each object A,

(26) $\quad F.R \subseteq F.S \;\Leftarrow\; R \subseteq S$ for each R and S of the same type,

(27) $\quad (F.R)^{\cup} = F.(R^{\cup})$ for each R.

Two examples of relators are List and product. List is a unary relator, and product is a binary relator. If R is a relation of type $I \leftarrow J$ then List.R relates a list of Is to a

list of J s whenever the two lists have the same length and corresponding elements are related by R . The relation $R \times S$ relates two pairs if the first components are related by R and the second components are related by S . List is an example of an inductively-defined datatype; in [BBH+92] it was observed that all inductively-defined datatypes are relators.

Now that we have the definition of a relator we may also give the definition of a hylo equation.

Definition 28 (Hylos) Let R and S be relations and F a relator. An equation of the form

$$X :: \quad X = R \circ F.X \circ S \tag{29}$$

is said to be a *hylo equation* or *hylo program*.
□

The identification of the importance of hylo equations is due to Meijer. (See e.g. [MFP91].)

Note that, on typing grounds, if the unknown X in equation (29) is to have type $A \leftarrow B$ then R must have type $A \leftarrow F.A$. We say that R is an F -*algebra with carrier* A . Also S must have type $F.B \leftarrow B$ (equivalently $S^{\cup}$ must be an F -algebra with carrier B). It is convenient to use the term *coalgebra* for a relation of type $F.B \leftarrow B$ for some B . So a *coalgebra with carrier* B is the converse of an algebra with carrier B .

3.2 Hylo Programs

In this section we show how frequently recursive programs can be rewritten in the form of hylo equations. We consider a variety of classes of recursion: structural recursion, primitive recursion, divide-and-conquer, and so on. In order to show that each of these classes is subsumed by the class of hylo equations some additional notation is introduced as and when necessary. It is not necessay to understand the notation in detail in order to be able to appreciate the examples, and the notation will not be used elsewhere.

Structural recursion The heart of functional programming is the declaration and use of datatypes. This is facilitated by the special purpose syntax that is used. A definition like that of the natural numbers in Haskell:

```
datatype Nat = Zero | Succ Nat
```

introduces two datatype constructors `Zero` and `Succ` of types `Nat` and `Nat -> Nat`, respectively. It also facilitates the definition of functions on natural numbers by pattern matching as in the definition of the function even:

```
even Zero = True

even (Succ n) = not (even n)
```

Category theory enables one to gain a proper understanding of such definitions and to lift the level of discussion from particular instances of datatypes to the general case, thus improving the effectiveness of program construction.

Category theory encourages us to focus on function composition rather than function application and to combine the two equations above into one equation, namely:

(30) $\quad \mathsf{even} \circ (\mathsf{zero} \triangledown \mathsf{succ}) \;=\; (\mathsf{true} \triangledown \mathsf{not}) \circ (\mathbb{1} + \mathsf{even}) \quad .$

In this form various important elements are more readily recognised. First, the two datatype constructors `Zero` and `Succ` have been combined into one *algebra* $\mathsf{zero} \triangledown \mathsf{succ}$. Similarly, `True` and `not` have been combined into the algebra $\mathsf{true} \triangledown \mathsf{not}$. The general mechanism being used here is the disjoint sum type constructor $(+)$ and the case operator $(\triangledown)$. Specifically, given types A and B, their disjoint sum $A+B$ comprises elements of A together with elements of B but tagged to say in which component of the disjoint sum they belong. Application of the function $f \triangledown g$ to a value of type $A+B$ involves inspecting the tag to see whether the value is in the left component of the sum or in the right. In the former case the function f is applied (after stripping off the tag); in the latter case the function g is applied. Thus for $f \triangledown g$ to be correctly typed, f and g must have the same target type. Then, if f has type $A \leftarrow B$ and g has type $A \leftarrow C$, the type of $f \triangledown g$ is $A \leftarrow B+C$.

Another important element of (30) is the *unit type* $\mathbb{1}$ and the term $\mathbb{1}+\mathsf{even}$. The unit type is a type with exactly one element. The term $\mathbb{1}+\mathsf{even}$ is read as the *functor* $\mathbb{1}+$ applied to the function even. As explained earlier, if f has type $A \leftarrow B$ the function $\mathbb{1}+f$ has type $\mathbb{1}+A \leftarrow \mathbb{1}+B$. It is the function that inspects the tag on a value of type $\mathbb{1}+B$ to see if it belongs to the left component, $\mathbb{1}$, or the right component, B. In the former case the value is left unaltered (complete with tag), and in the latter case the function f is applied to the untagged value, and then the tag is replaced. The functor $\mathbb{1}+$ is called the *pattern functor* of the datatype $\mathbb{N}$ (`Nat` in Haskell-speak) [BJJM99].

The final aspect of (30) that is crucial is that it uniquely defines the function even. (To be precise, the equation

$$X :: \quad X \circ (\mathsf{zero} \triangledown \mathsf{succ}) = (\mathsf{true} \triangledown \mathsf{not}) \circ (\mathbb{1} + X)$$

has a unique solution.) This is the concept of *initiality* in category theory. Specifically, $\mathsf{zero} \triangledown \mathsf{succ}$ is an *initial* $(\mathbb{1}+)$*-algebra* which means that for all $(\mathbb{1}+)$-algebras f the equation

$$X :: \quad X \circ (\mathsf{zero} \triangledown \mathsf{succ}) = f \circ (\mathbb{1} + X)$$

has exactly one solution.

In summary, category theory identifies three vital ingredients in the definition (30) of the function even, namely, the functor $\mathbb{1}+$, the initial $(\mathbb{1}+)$-algebra zero▽succ and the $(\mathbb{1}+)$-algebra true▽not.

The general form exemplified by (30) is

$$X \circ \mathsf{in} = f \circ \mathsf{F}.X \tag{31}$$

where F is a functor, in is an initial F-algebra and f is an F-algebra. This general form embodies the use of structural recursion in modern functional programming languages like Haskell. The left side embodies pattern matching since, typically, in embodies a case analysis as exemplified by zero▽succ. The right side exhibits recursion over the structure of the datatype, which is represented by the "pattern" functor F.

Here is the formal definition of an initial algebra. The definition is standard —an initial object in the category of F-algebras— but we give it nonetheless in order to introduce some terminology.

Definition 32 Suppose F is an endofunctor on some category $\mathcal{C}$. An arrow f in $\mathcal{C}$ is an F-*algebra* if $f \in A \leftarrow \mathsf{F}.A$ for some A, the so-called *carrier* of the algebra. If f and g are both F-algebras with carriers A and B then arrow $\varphi \in A \leftarrow B$ is said to be an F-algebra *homomorphism* to f from g if $\varphi \circ f = g \circ \mathsf{F}.\varphi$. The category FAlg has objects all F-algebras and arrows all F-algebra homomorphisms. Composition and identity arrows are inherited from the base category $\mathcal{C}$. The arrow $\mathsf{in} \in I \leftarrow \mathsf{F}.I$ is an *initial* F-algebra if for each $f \in A \leftarrow \mathsf{F}.A$ there exists an arrow $(\!|f|\!) \in A \leftarrow I$ such that for all $h \in A \leftarrow I$,

$$h = (\!|f|\!) \quad \equiv \quad h \in f \xleftarrow{\mathsf{FAlg}} \mathsf{in} \quad . \tag{33}$$

So, $(\!|f|\!)$ is the unique homomorphism to algebra f from algebra in. We call $(\!|f|\!)$ the *catamorphism* of f.

□

The "banana bracket" notation for catamorphisms (as it is affectionately known) was introduced by Malcolm [Mal90a, Mal90b]. Malcolm was also the first to express the unicity property using an equivalence in this way. It is a mathematically trivial device but it helps enormously in reasoning about catamorphisms. Note that the functor F is also a parameter of $(\!|f|\!)$ but the notation does not make this explicit. This is because the functor F is usually fixed in the context of the discussion. Where disambiguation is necessary, the notation $(\!|F; f|\!)$ is sometimes used. The initial algebra is also a parameter that is not made explicit; this is less of a problem because initial F-algebras are isomorphic and thus catamorphisms are defined "up to isomorphism".

An important property of initial algebras, commonly referred to as Lambek's lemma [Lam68], is that an initial algebra is both injective and surjective. Thus, for example, zero$\triangledown$succ is an isomorphism between $\mathbb{N}$ and $\mathbb{1}+\mathbb{N}$. Lambek's lemma has the consequence that, if in is an initial F-algebra,

$$h \in f \xleftarrow{FAlg} in \;\equiv\; h = f \circ F.h \circ in^{\cup}$$

where $in^{\cup}$ is the inverse of in. Thus, the characterising property (33) of catamorphisms is equivalent to, for all h and all F-algebras f,

$$(34) \qquad h = (\!|f|\!) \;\equiv\; h = f \circ F.h \circ in^{\cup} \quad .$$

That is, $(\!|f|\!)$ is the unique fixed point of the function mapping h to $f \circ F.h \circ in^{\cup}$. Equivalently, $(\!|f|\!)$ is the unique solution of the hylo equation:

$$h:: \qquad h = f \circ F.h \circ in^{\cup} \quad .$$

In the context of functions on lists the catamorphism $(\!|f|\!)$ is known to functional programmers as a *fold* operation. Specifically, for lists of type I the relevant pattern functor F is the functor mapping X to $\mathbb{1}+(I\times X)$ (where $\times$ denotes the cartesian product functor) and an F-algebra is a function of type $A \leftarrow \mathbb{1}+(I\times A)$ for some A. Thus an F-algebra takes the form $c \triangledown (\oplus)$ for some function c of type $A \leftarrow \mathbb{1}$ and some function $\oplus$ of type $A \leftarrow I\times A$. The characterising property of the catamorphisms is thus

$$h = (\!|c \triangledown (\oplus)|\!) \;\equiv\; h = (c \triangledown (\oplus)) \circ (\mathbb{1} + (I\times h)) \circ (nil^{\cup} \blacktriangledown cons^{\cup}) \quad .$$

Here $nil^{\cup} \blacktriangledown cons^{\cup}$ is the inverse of nil$\triangledown$cons. (In general, $R\blacktriangledown S$ is the converse conjugate of $R\triangledown S$. That is, $(R\blacktriangledown S)^{\cup} = R^{\cup} \triangledown S^{\cup}$.) It can be read as the pattern matching operator: look to see whether the argument is an empty list or a non-empty list. In the former case $nil^{\cup}$ returns an element of the unit type, tagging it so that the result of the test is passed on to later stages; in the latter case $cons^{\cup}$ splits the list into a head and a tail, the resulting pair also being tagged for later identification. Using the algebraic properties of case analysis, the characterising property is equivalent to

$$h = (\!|c \triangledown (\oplus)|\!) \;\equiv\; h \circ nil = c \;\wedge\; h \circ cons = (\oplus) \circ (I\times h)$$

the right side of which is a point-free free formulation of the definition of a fold with seed the constant c and binary operator $(\oplus)$. As a concrete example, the function sum that sums the elements of a list is

$$(\!|zero \triangledown add|\!)$$

where add is the addition function. In Haskell this function would be written

```
fold 0 add .
```

Although catamorphisms (folds) are best known in the context of functional programming many relations are also catamorphisms. For example, the prefix relation on lists is uniquely characterised by the two equations

$$\mathsf{nil}[\![\mathsf{prefix}]\!]\mathsf{nil}$$

and $$xs[\![\mathsf{prefix}]\!](y:ys) \;\equiv\; xs = \mathsf{nil} \vee \exists(zs ::\; xs = y:zs \wedge zs[\![\mathsf{prefix}]\!]ys) \quad .$$

Expressed as one, point-free equation this is

$$\mathsf{prefix} \circ (\mathsf{nil} \triangledown \mathsf{cons}) \;=\; (\mathsf{nil} \triangledown ((\mathsf{nil} \circ \top\!\top) \cup \mathsf{cons})) \circ (\mathbb{1} + (\mathrm{I} \times \mathsf{prefix})) \tag{35}$$

where I denotes the type of the list elements. Here we recognise a *relator* and two algebras: in this case the relator is $(\mathbb{1}+(\mathrm{I}\times))$ and the two $(\mathbb{1}+(\mathrm{I}\times))$-algebras are $\mathsf{nil}\triangledown\mathsf{cons}$ and $\mathsf{nil} \triangledown ((\mathsf{nil} \circ \top\!\top) \cup \mathsf{cons})$. (Note that the second algebra is not a function.) Equivalently, prefix is the unique solution of a hylo equation:

$$\mathsf{prefix} \;=\; (\mathsf{nil} \triangledown ((\mathsf{nil} \circ \top\!\top) \cup \mathsf{cons})) \circ (\mathbb{1} + (\mathrm{I} \times \mathsf{prefix})) \circ (\mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup}) \quad . \tag{36}$$

Primitive recursion Structural recursion is useful since many programs that arise in practice have this kind of recursion. However, just as structural induction is not enough to prove all facts that can be proved by induction, structural recursion is not enough to define all programs that can be defined by recursion. As an example of a program that is not structurally recursive, consider the factorial function, the function defined by the two equations

$$\mathsf{fact} \circ \mathsf{zero} = \mathsf{one} \quad \text{and} \quad \mathsf{fact} \circ \mathsf{succ} = \mathsf{times} \circ (\mathsf{fact} \vartriangle \mathsf{succ}) \quad ,$$

where one is the constant function returning the number 1 and times is the multiplication function. These equations can be combined into the single equation

$$\mathsf{fact} \;=\; (\mathsf{one} \triangledown (\mathsf{times} \circ (\mathsf{succ} \times \mathbb{N}))) \circ (\mathbb{1} + (\mathbb{N} \vartriangle \mathsf{fact})) \circ (\mathsf{zero}^{\cup} \blacktriangledown \mathsf{succ}^{\cup}) \quad . \tag{37}$$

Reading from the right, the factorial function first examines its argument to determine whether it is zero or the successor of another number; in the former case a tagged element of the unit type is returned, and in the latter case the predecessor of the input value is returned, suitably tagged. Subsequently, if the input value is $n+1$, the function $\mathbb{N}\vartriangle\mathsf{fact}$ constructs a pair consisting of the number n and the result of the factorial function applied to n. (As forewarned, $\mathbb{N}$ is used here to denote the identity function on natural numbers.) The calculation of $(n+1) \times n!$ is the result of applying the function $\mathsf{times} \circ (\mathsf{succ} \times \mathbb{N})$ to the (untagged) pair. On the other hand, if the input value is zero then one is returned as result.

To give an example of a relation defined by primitive recursion we need look no further than the suffix relation on lists. It satisfies

$$\mathsf{nil}[\![\mathsf{suffix}]\!]\mathsf{nil}$$

and $$xs[\![\mathsf{suffix}]\!](y : ys) \quad \equiv \quad xs = y : ys \ \vee \ xs[\![\mathsf{suffix}]\!]ys \quad .$$

Expressed as a fixed point equation this is:

$$\mathsf{suffix} \ = \ (\mathsf{nil} \triangledown ((\mathsf{cons} \circ \mathsf{exl}) \cup (\mathsf{exr} \circ \mathsf{exr}))) \circ (\mathbb{1} + (I \times (\mathsf{List}.I \vartriangle \mathsf{suffix}))) \circ (\mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup})$$

where I is the type of the list elements and exl and exr project a pair onto its left and right components, respectively. This is a definition by primitive recursion.

When we abstract from the particular functor and initial algebra in factorial program (37) a general recursion scheme is obtained.

$$(38) \qquad X \ :: \ X \ = \ R \circ \mathsf{F}.(I \times X) \circ \mathsf{F}.(I \vartriangle I) \circ \mathsf{in}^{\cup} \quad .$$

In the case of the factorial function R is $\mathsf{one} \triangledown (\mathsf{times} \circ \mathsf{succ} \times \mathsf{id})$, F is $(\mathbb{1} +)$, I is the (identity on) natural numbers and in is $\mathsf{zero} \triangledown \mathsf{succ}$. (Note that $(W \times X) \circ (I \vartriangle I) = W \vartriangle X$ for all W and X with source I. Hence $\mathsf{F}.(I \times X) \circ \mathsf{F}.(I \vartriangle I) = \mathsf{F}.(I \vartriangle X)$. We have applied this so-called $\times$-$\vartriangle$-fusion law in order to make the term $\mathsf{F}.(I \vartriangle I)$ explicit.) A definition of this form is called *primitive recursive*.

This generic formulation of primitive recursion was introduced (for functions) by Meertens [Mee92]. He called such an equation a *para equation* and a solution to the equation a *paramorphism*.

Divide and Conquer As the name suggests, "primitive" recursion is also unsuitable as the basis for a practical methodology of recursive program construction. Divide-and-conquer is a well-known technique that is not easily expressed using primitive recursion.

An example of a divide-and-conquer program is the sorting algorithm known as "quicksort". Quicksort, here abbreviated to qs, is uniquely defined by the hylo equation:

$$(39) \qquad \mathsf{qs} \ = \ (\mathsf{nil} \triangledown (\mathsf{join} \circ (I \times \mathsf{cons}))) \circ (\mathbb{1} + (\mathsf{qs} \times (I \times \mathsf{qs}))) \circ (\mathsf{nil}^{\cup} \blacktriangledown \mathsf{dnf})$$

To see that this is the quicksort program one has to interpret dnf as the well-known "Dutch national flag" relation: the relation that splits a non-empty list into a tuple $(xs, (x, ys))$ formed by a list, an element and a list such that all elements in the list xs are at most x and all elements in ys are greater than x. The results of the recursive calls are assembled to the output list by the operation $\mathsf{join} \circ (I \times \mathsf{cons})$, where join produces the concatenation of two lists.

A typical divide and conquer program is of the form

$$(40) \qquad X = (R \triangledown \mathsf{conquer}) \circ (I + (X \times X)) \circ (I + \mathsf{divide}) \circ (A \blacktriangledown B) \quad .$$

Interpreting this program should not be difficult. A test is made to determine whether the input is a base case (if the input satisfies A), the output then being computed by R. If the input is not a base case (if the input satisfies B) the input is split into two smaller "subproblems" by divide. Then the smaller problems are solved recursively and finally the two solutions of the subproblems are assembled into an output by conquer.

Of course there are more divide and conquer schemes. For example, the original problem can be split into more than two subproblems. It is also possible that the divide step produces, besides a number of subproblems, a value that is not "passed into the recursion"; then the middle relation of (40) has a form like $I \times (X \times X)$. Quicksort is an example of such a divide and conquer algorithm.

Repetition is an elementary and familiar example of divide and conquer in which the original problem is reduced to a single subproblem. A repetition is a solution of the equation in x:

$$(41) \qquad x \;=\; \mathsf{if}\ \neg b \to \mathsf{skip}\ [\!]\ b \to s;x\ \mathsf{fi} \quad .$$

Using the fact that skip (do nothing) corresponds to the identity function, I, on the state space and writing B for the coreflexive corresponding to predicate b and S for the relation corresponding to the statement s, we may express (41) using disjoint sum as:

$$(42) \qquad X \;=\; (I \triangledown I) \circ (I + X) \circ (\sim B \blacktriangleright (S \circ B)) \quad .$$

Here we see how **while** statements are expressed in terms of hylo equations, the relator being $(\ I+\ )$.

Parameterised recursion Often recursive programs conform to one of the schemes discussed above but this is obscured by the presence of an additional parameter. Elementary examples are the definitions of addition, multiplication and exponentiation on natural numbers, which are essentially, but not quite, definitions by structural recursion:

$$0+n = n \quad \text{and} \quad (m+1)+n = (m+n)+1 \ ,$$

$$0 \times n = 0 \quad \text{and} \quad (m+1) \times n = m \times n + n \ ,$$

$$n^0 = 1 \quad \text{and} \quad n^{m+1} = n^m \times n \ .$$

All these definitions have the form

$$X.(0, n) = f.n \quad \text{and} \quad X.(m+1, n) = g.(m, h.n)$$

where X is the function being defined and f, g and h are known functions. (We leave the reader to supply the instantiations for f, g and h.) In point-free form, we have yet again a hylo equation:

$$X \;=\; k \circ ((\mathbb{1} + X) \times \mathbb{N}) \circ (\mathsf{pass} \vartriangle \mathsf{exr}) \circ ((\mathsf{zero}^\cup \blacktriangledown \mathsf{succ}^\cup) \times \mathbb{N})$$

where $k \;=\; ((f \circ \mathsf{exr}) \triangledown (g \circ (\mathsf{id} \times h))) \circ \mathsf{distr}$.

Here distr is a function of type $(H \times K) + (J \times K) \leftarrow (H + J) \times K$ that is polymorphic in H, J and K, and pass is a function of type $\mathbb{1} + (I \times K) \leftarrow (\mathbb{1} + I) \times K$ that is polymorphic in I and K.

Despite the seeming complexity of the underlying algebra and coalgebra, the basic structure is thus a hylo equation.

Another example, with the same structure but defined on a datatype other than the natural numbers, is the program that appends two lists. The standard definition comprises the two equations

$$\mathsf{nil} +\!\!+ ys \;=\; ys \quad \text{and} \quad (x : xs) +\!\!+ ys \;=\; x : (xs +\!\!+ ys) \;\; .$$

As a single equation (where we write join instead of $+\!\!+$):

$$\mathsf{join} \;=\; \mathsf{post} \circ ((\mathbb{1} + (I \times \mathsf{join})) \times \mathsf{List}.I) \circ (\mathsf{pass} \vartriangle \mathsf{exr}) \circ ((\mathsf{nil}^\cup \blacktriangledown \mathsf{cons}^\cup) \times \mathsf{List}.I) \;\; .$$

where $\mathsf{post} \;=\; (\mathsf{exr} \triangledown \mathsf{cons}) \circ \mathsf{distr}$. Here distr is as before whereas in this case pass is a function of type $\mathbb{1} + (I \times (J \times K)) \leftarrow (\mathbb{1} + (I \times J)) \times K$ that is polymorphic in I, J and K. Once again we recognise a hylo equation.

3.3 Intermediate data structures

At the beginning of section 3.2 we discussed the use of recursion on the structure of a datatype; if R is an F-algebra with carrier A then the catamorphism $(\!|R|\!)$ can be seen as a program that *destructs* an element of an initial F-algebra in order to compute a value of type A. The converse $(\!|R|\!)^\cup$ is thus a program that *constructs* an element of the initial algebra from a value of type A.

Now suppose R and $S^\cup$ are both F-algebras with carriers A and B, respectively. Then the composition $(\!|R|\!) \circ (\!|S^\cup|\!)^\cup$ has type $A \leftarrow B$. It computes a value of type A from a value of type B by first building up an intermediate value which is an element of an initial F-algebra and then breaking the element down. The remarkable theorem is that

$(\!|R|\!) \circ (\!|S^\cup|\!)^\cup$ is the least solution of the hylo equation (29).

This theorem (which we formulate precisely below) gives much insight into the design of hylo programs. It says that executing a hylo program is equivalent to constructing an intermediate data structure, the form of which is specified by the relator F, and then breaking this structure down. The two phases are called the *anamorphism* phase and the *catamorphism* phase. Executing a hylo equation for a specific input value by

unfolding the recursion hides this process; it is as if the intermediate data structure is broken down as it is being built up. (A good comparison is with a Unix pipe in which the values in the pipe are consumed as soon as they are produced.) Execution of $(\![R]\!) \circ (\![S^\cup]\!)^\cup$ does make the process explicit. For this reason, the relator F is said to specify a *virtual* data structure [SdM93].

Two simple examples of virtual data structures are provided by do-statements and the factorial function. In the case of do-statements (see (42)) the virtual datatype is the carrier set of an initial $(I+)$-algebra, a type which is isomorphic to $I \times \mathbb{N}$ —thus an element of the virtual datatype can be seen as a pair consisting of an element of the state space and a natural number, the latter being a "virtual" count of the number of times the loop body is executed. In the case of the factorial function, definition (37) can be rewritten so as to make the relator F explicit:

$$\mathsf{fact} \;=\; (\mathsf{one} \triangledown (\mathsf{times} \circ (\mathsf{succ} \times \mathbb{N}))) \circ (\mathbb{1} + (\mathbb{N} \times \mathsf{fact})) \circ (\mathsf{zero}^\cup \blacktriangledown ((\mathbb{N} \vartriangle \mathbb{N}) \circ \mathsf{succ}^\cup)) \quad .$$

The "virtual" datatype is thus the type of lists of natural numbers, the carrier set of an initial $\mathbb{1}+(\mathbb{N}\times)$-algebra. The list that is constructed for a given input n is the list of natural numbers from $n-1$ down to 0 and the hylo theorem states that the factorial of n can be calculated by constructing this list (the anamorphism phase) and then multiplying the numbers together after adding 1 (the catamorphism phase).

Language recognition also illustrates the process well. Let us explain the process first with a concrete example following which we will sketch the generic process. Consider the following grammar:

$$S \;::=\; aSb \mid c$$

where, for our purposes, a, b and c denote some arbitrary sets of words over some fixed alphabet. Associated with this grammar is a data structure: the class of parse trees for strings in the language generated by the grammar. This data structure, Stree, satisfies the equation:

$$\mathsf{Stree} \;=\; (a \times \mathsf{Stree} \times b) + c \quad .$$

It is an initial F-algebra where F maps X to $(a \times X \times b) + c$. Now the process of *unparsing* a parse tree is very easy to describe since it is defined by induction on the structure of parse trees. Indeed the unparse function is the F-catamorphism $(\![(\mathsf{concat3} \circ (a \times \mathsf{id} \times b)) \triangledown c]\!)$ where $\mathsf{concat3}$ concatenates three strings together, a, b and c are the identity functions on the sets a, b and c, and id is the identity function on all words. Moreover, its left domain is equal to the language generated by the grammar. Since in general the left domain of function f is $f \circ f^\cup$ the language generated satisfies

$$S \;=\; (\![(\mathsf{concat3} \circ (a \times \mathsf{id} \times b)) \triangledown c]\!) \circ (\![(\mathsf{concat3} \circ (a \times \mathsf{id} \times b)) \triangledown c]\!)^\cup \quad .$$

This equation defines a (nondeterministic) program to recognise strings in the language. The program is a partial identity on words. Words are recognised by first building a parse tree and then unparsing the tree. By the hylo theorem (or directly from the definition of S) we also have the hylo program

$$S \;=\; ((\mathsf{concat3}\circ(a\times\mathsf{id}\times b))\triangledown c)\circ((a\times S\times b)+c)\circ(((a\times\mathsf{id}\times b)\circ\mathsf{concat3}^{\cup})\blacktriangledown c) \quad .$$

This is a program that works by (nondeterministically) choosing to split the input word into three segments (using $\mathsf{concat3}^{\cup}$) or to check whether the word is in the language c. In the former case the first segment is checked for membership in a, the third segment is checked for membership in b and the program is called recursively to check the middle segment. Subsequently the three segments are recombined into one. In the latter case the word is left unchanged.

The derivation of a language recogniser in this way can be generalised to an arbitrary context-free grammar. (This is only possible because we base our methodology on relation algebra. The non-determinism present in a typical context-free grammar prohibits the generalisation we are about to make in the context of functional programming.) A context-free grammar defines a type of parse trees in a fairly obvious way. Also an $\mathsf{unparse}$ function can always be defined mapping parse trees to strings. This function is a catamorphism. The language generated by the grammar is the left domain of the unparse function, which is $\mathsf{unparse}\circ\mathsf{unparse}^{\cup}$. This in turn is the composition of a catamorphism and the converse of a catamorphism, which can be expressed as a hylo program using the hylo theorem.

In practice the process is complicated by the fact that all practical context-free grammars have more than one nonterminal, and nonterminals are linked together via mutual recursion. But the theory we have developed covers this case too. Mutual recursion is modelled by endorelators on a product category.

3.4 The Hylo Theorem

We summarise the previous section with a formal statement of the hylo theorem. The theorem is rather deeper than just the statement that the least solution of a hylo equation is the composition of a catamorphism and an anamorphism. The proof of the theorem has been given in detail elsewhere [BH99][1].

[1] Actually [BH99] contains a proof of the dual theorem concerning final coalgebras and is more general than the theorem stated here. Unlike in a category, dualising between initiality and finality is not always straightforward in an allegory because of the lack of duality between intersection and union. However, dualising from a finality property to an initiality property is usually straightforward and it is the other direction that is difficult. That is one reason why [BH99] chose to present the theorem in terms of coalgebras rather than algebras. The extra generality offered by the theorem in [BH99] encompasses the relational

Recall that we defined the notion of an initial algebra in the context of a category. (See (32).) To all intents and purposes this amounts to defining the notion of an initial algebra in the context of functions between sets. What we need however is the notion of an initial algebra in the context of binary relations on sets, that is, in the context of an allegory. Definition 43 is such a definition. The hylo theorem states that the categorical notion of an initial algebra coincides with the allegorical notion if the allegory is locally complete and tabular.

Definition 43 Assume that F is an endorelator. Then (I, in) is a *relational initial* F-algebra iff $in \in I \leftarrow F.I$ is an F-algebra and there is a mapping $(\!|\_|\!)$ defined on all F-algebras such that

(44) $(\!|R|\!) \in A \leftarrow I \quad \text{if } R \in A \leftarrow F.A$,

(45) $(\!|in|\!) = id_I$, and

(46) $(\!|R|\!) \circ (\!|S|\!)^{\cup} = \mu\langle X :: R \circ F.X \circ S^{\cup} \rangle$.

That is, $(\!|R|\!) \circ (\!|S|\!)^{\cup}$ is the smallest solution of the equation $X :: R \circ F.X \circ S^{\cup} \subseteq X$.
□

In order to state the hylo theorem we let $Map(\mathcal{A})$ denote the sub-category of functions in the allegory $\mathcal{A}$. For clarity we distinguish between the endorelator F and the corresponding endofunctor defined on $Map(\mathcal{A})$.

Theorem 47 (Hylo Theorem) Suppose F is an endorelator on a locally-complete, tabular allegory $\mathcal{A}$. Let F' denote the endofunctor obtained by restricting F to the objects and arrows of $Map(\mathcal{A})$. Then in is an initial F'-algebra if and only it is a relational initial F-algebra.

□

3.5 Reducing problem size

There are two elements in the design of the body of a **while** statements: it should maintain an invariant relation established by the initialisation procedure, and it should make progress to the termination condition. The latter is guaranteed if the loop body is a well-founded relation on the state space. There are also two elements in the design of hylo equations. The intermediate data structure plays the role of the invariant relation, whilst making progress is achieved by ensuring that each recursive call is "smaller" than

properties of disjoint sum and cartesian product but at the expense of requiring a more sophisticated understanding of allegory theory which we wanted to avoid in the current presentation.

the original argument. In this section we formalise this requirement. The notion we introduce, "F-reductivity" due to Henk Doornbos [Doo96], generalises the notion of admitting induction essentially by making the intermediate data structure a parameter. As we shall indicate in section 3.6 this has important ramifications for developing a calculus of program termination.

Informally, for hylo program $X = S \circ F.X \circ R$ we require that all values stored in an output F-structure of R have to be smaller than the corresponding input to R. More formally, with $x[\![\mathsf{mem}]\!]y$ standing for "x is a member of F-structure y" (or, x is a value stored in F-structure y"), we demand that for all x and z

$$\forall\langle y:: x[\![\mathsf{mem}]\!]y \wedge y[\![R]\!]z \Rightarrow x \prec z\rangle \quad ,$$

for some well-founded ordering $\prec$. If this is the case we say that R is F*-reductive*.

To make the definition of reductivity completely precise we actually want to avoid the concept of "values stored in an F-structure". (This is because its incorporation into the definition of F-reductivity limits the practicality of the resulting theory.) Fortunately, Hoogendijk and De Moor [HdM00, Hoo97] have shown how to characterise membership of a so-called "container" type in such a way that it can be extended to other types where the intuitive notion of "membership" is not so readily apparent.

Hoogendijk and De Moor's characterisation of the membership relation of a relator is the following:

Definition 48 (Membership) Relation $\mathsf{mem} \in I \leftarrow F.I$ is a membership relation of relator F if and only if it satisfies, for all coreflexives A, $A \subseteq I$:

$$F.A = \mathsf{mem} \backslash A \quad .$$

□

When this definition is expressed pointwise it reads:

$$x \in F.A \equiv \forall\langle i: i[\![\mathsf{mem}]\!]x: i \in A\rangle \quad .$$

Informally: an F-structure satisfies the property $F.A$ iff all the values stored in the structure satisfy property A. For example, for the list relator mem holds between a point and a list precisely when the point is in the list. For product the relation holds between x and (x,y) and also between y and (x,y).

This definition of membership leads to a definition of F-reductivity independent of the notion of values stored in an F-structure. To see this we observe that, for coalgebra R with carrier I and for coreflexive A below I, we have:

$$
\begin{array}{ll}
 & (\mathsf{mem} \circ R)\backslash\!\!\cdot A \\
= & \quad \{ \quad \text{factors (2)} \quad \} \\
 & R\backslash\!\!\cdot(\mathsf{mem}\backslash\!\!\cdot A) \\
= & \quad \{ \quad \text{definition 48} \quad \} \\
 & R \backslash\!\!\cdot \mathsf{F}.A \quad .
\end{array}
$$

Now, that $S \in I \leftarrow I$ admits induction is the condition that the least prefix point of the function $\langle A :: S\backslash\!\!\cdot A\rangle$ is I, and our informal notion of the reductivity of $R \in \mathsf{F}.I \leftarrow I$ is that $\mathsf{mem} \circ R$ should be well-founded. Since being well-founded is equivalent to admitting induction, the latter is equivalent to the requirement that the least prefix point of the function $\langle A :: R \backslash\!\!\cdot \mathsf{F}.A\rangle$ is I, which does not involve any appeal to notions of membership of a "container" type. This gives us a precise, generic definition of the notion of F-reductivity:

Definition 49 (F-reductivity) Relation $R \in \mathsf{F}.I \leftarrow I$ is said to be F-*reductive* if and only if it enjoys the property:

$$\mu((R\backslash\!\!\cdot)\bullet\mathsf{F}) = I \quad . \tag{50}$$

□

Obviously F-reductivity generalises the notion of admitting induction. (A relation R admits induction if and only if it is Id-reductive, where Id denotes the identity relator.) An immediate question is whether there is a similar generalisation of the notion of well-foundedness and a corresponding theorem that F-well-foundedness is equivalent to F-reductivity. As it turns out, there is indeed a generic notion of well-foundedness but this is strictly weaker than F-reductivity. The definition is given below, the facts just stated are left as exercises in the use of fixed point calculus.

Well-foundedness of relation R is equivalent to the equation $X :: X = X \circ R$ having a unique solution (which is obviously $\perp\!\!\!\perp$, the empty relation). This is easily generalised to the property that, for all relations S, the equation $X :: X = S \circ X \circ R$ has a unique solution. The generic notion of well-foundedness focusses on this unicity of the solution of equations.

Definition 51 (F-well-founded) Relation $R \in \mathsf{F}.I \leftarrow I$ is F-*well-founded* iff, for all relations $S \in J \leftarrow \mathsf{F}.J$ and $X \in J \leftarrow I$,

$$X = S \circ \mathsf{F}.X \circ R \quad \equiv \quad X = \mu\langle Y :: S \circ \mathsf{F}.Y \circ R\rangle \quad .$$

□

Exercise 52 *Verify the claim made immediately before definition 51. That is, show that* R *is well-founded equivales*

$$\forall\langle X, S:: X = S\circ X\circ R \;\equiv\; X = \mu\langle Y:: S\circ Y\circ R\rangle\rangle \;\;.$$

In words, R is well-founded equivales R is Id-well-founded. (Hint: if R is well-founded then $\mu\langle Y:: S\circ Y\circ R\rangle = \bot\!\!\!\bot$.)

□

Exercise 53 *Prove that an* F*-reductive relation is* F*-well-founded.*

□

An example of a relation that is F-well-founded but not F-reductive can be constructed as follows. Define the relator F by $F.X = X\times X$. Suppose $R\in I\leftarrow I$ is a non-empty well-founded relation. Then the relation $R\triangle I$ of type $I\times I\leftarrow I$ (which relates a pair of values (x,y) each of type I to a single value z of type I iff x is related by R to z and $y = z$) is F-well-founded but not F-reductive. For a proof see [Doo96].

3.6 A calculus of F-reductivity

The introduction of a data structure —the relator F— as a parameter to the notion of reductivity is a significant advance because it admits the possibility of developing a calculus of reductivity and thus of program termination based on the structure of the parameter. A beginning has been made to the development of such a calculus [DB95, Doo96] sufficient to establish the termination of all the examples given in section 3.2 by a process akin to type checking.

Space only allows us to give a brief taste of the calculus here. The fundamental theorem is the following.

Theorem 54 The converse of an initial F-algebra is F-reductive.

Proof Let $\mathsf{in}\in I\leftarrow F.I$ be an initial F-algebra and A an arbitrary coreflexive of type $I\leftarrow I$. We must show that

$$I\subseteq A \;\;\Leftarrow\;\; \mathsf{in}^{\cup}\backslash F.A \subseteq A \;\;.$$

We start with the antecedent and derive the consequent:

$$\begin{array}{ll} & \mathsf{in}^{\cup}\backslash F.A \subseteq A \\ \equiv & \quad\{\quad \text{for function } f \text{ and coreflexive } B,\ f\backslash B = f^{\cup}\circ B\circ f, \\ & \qquad\ \mathsf{in}^{\cup} \text{ is a function and } F.A \text{ is a coreflexive} \quad\} \end{array}$$

$$
\begin{array}{ll}
 & \mathsf{in} \circ \mathsf{F}.A \circ \mathsf{in}^{\cup} \subseteq A \\
\Rightarrow & \quad \{ \quad \text{Hylo theorem: (47) and (43),} \\
 & \qquad \mathsf{in} \text{ is an initial } \mathsf{F}\text{-algebra} \quad \} \\
 & (\!|\mathsf{in}|\!) \subseteq A \\
\equiv & \quad \{ \quad \text{identity rule: (45), } \mathsf{in} \in \mathrm{I} \leftarrow \mathsf{F}.\mathrm{I} \text{ is an initial } \mathsf{F}\text{-algebra} \quad \} \\
 & \mathrm{I} \subseteq A \quad .
\end{array}
$$

□

Theorem 54 has central importance because, if we examine all the programs in section 3.2 we see that the converse of a initial F-algebra is at the heart of the coalgebra in all the hylo equations. In the case, for example, of primitive recursion the generic equation has the form

$$X \;::\; X = \mathrm{R} \circ \mathsf{F}.(\mathrm{I} \times X) \circ \mathsf{F}.(\mathrm{I} \triangle \mathrm{I}) \circ \mathsf{in}^{\cup}$$

and the coalgebra is $\mathsf{F}.(\mathrm{I} \triangle \mathrm{I}) \circ \mathsf{in}^{\cup}$ where $\mathsf{in} \in \mathrm{I} \leftarrow \mathsf{F}.\mathrm{I}$ is an initial F-algebra. For the equation to define a terminating program (and consequently have a unique solution) we must show that the coalgebra is $(\mathsf{F}\bullet(\mathrm{I}\times))$-reductive. This is done by showing that $\mathsf{F}.(\mathrm{I} \triangle \mathrm{I})$ transforms any F-reductive relation into an $(\mathsf{F}\bullet(\mathrm{I}\times))$-reductive relation — which is a consequence of the fact that $\mathsf{F}.(\mathrm{I} \triangle \mathrm{I})$ is an instance of a natural transformation of the relator F to the relator $\mathsf{F}\bullet(\mathrm{I}\times)$.

Acknowledgements The material in this paper was developed whilst the author was heading the Mathematics of Program Construction group at Eindhoven University of Technology, particularly during the period 1990–1995. It would not have been possible to write the paper without the wonderfully stimulating and highly productive team effort that went into all we did at that time. The sections on well-foundedness and admitting induction are extracted from [DBvdW97] written jointly with Henk Doornbos and Jaap van der Woude, the sections on hylomorphisms and reductivity are extracted from Doornbos's thesis [Doo96] (see also [DB95, DB96]), and the hylomorphism theorem in the form presented here is joint work with Paul Hoogendijk [BH99].

4 Solutions to Exercises

1 X is the relation on pairs (w,k)

$$w\in L \;\equiv\; w'=\varepsilon \wedge k'=0$$

where w is a word, k is a natural number and L is the language given in the statement of the problem. The invariant is the relation

$$w\in L \;\equiv\; w'\in L^{k'} \quad ,$$

the initialisation is the relation

$$w=w' \wedge k'=1 \quad ,$$

the loop body is the relation

$$k\neq 0 \;\wedge\; (\exists\langle v::\; w=av \wedge w'=v \wedge k'=k-1\rangle \vee \exists\langle v::\; w=bv \wedge w'=v \wedge k'=k+1\rangle) \quad ,$$

and the termination is the relation

$$w=w'=\varepsilon \;\wedge\; k'=k=0 \quad .$$

□

12 Let p denote $\nu\langle X::X\circ R\rangle$. Then p is characterised by the two properties

(a) $\quad p=p\circ R \quad$, and

(b) $\quad \forall\langle q:\; q=\top\top\circ q:\; q\subseteq p \Leftarrow q=q\circ R\rangle \quad .$

Interpreting p as a set, (a) is the property that $x\in p$ equivales $\exists\langle y:y\in p:y[\![R]\!]x\rangle$ and (b) is the property that p is the largest such set. In words, p is the largest set of elements such that each element begins an infinite chain of R-related elements.
□

13 By theorem 8 we have to show that the greatest fixed point of $\langle X::\; X\circ f\cup\circ R\circ f\rangle$ equals $\bot\!\!\!\bot$. We do this by first rewriting $\nu\langle X::\; X\circ f\cup\circ R\circ f\rangle$ in terms of $\nu\langle X::X\circ R\rangle$. We have:

$$\begin{array}{ll} & \nu\langle X::\; X\circ f\cup\circ R\circ f\rangle \\ = & \quad \{ \quad \text{rolling rule} \quad \} \\ & \nu\langle X::\; X\circ f\circ f\cup\circ R\rangle\circ f \\ \subseteq & \quad \{ \quad f\circ f\cup\subseteq I\text{, by definition of functional; } \nu \text{ is monotonic} \quad \} \\ & \nu\langle X::X\circ R\rangle\circ f \quad . \end{array}$$

The more general statement is thus

$$\nu\langle X::\ X\circ f\cup\circ R\circ f\rangle\ \subseteq\ \nu\langle X::\ X\circ R\rangle\circ f\quad .$$

Making use of it, we have:

$$\begin{array}{ll} & \nu\langle X::\ X\circ f\cup\circ R\circ f\rangle\subseteq \bot\!\!\!\bot \\ \Leftarrow & \quad \{\quad \text{transitivity}\quad \} \\ & \nu\langle X::\ X\circ R\rangle\circ f\ \subseteq\ \bot\!\!\!\bot \\ \Leftarrow & \quad \{\quad \bot\!\!\!\bot \text{ is zero of composition and composition is monotonic}\quad \} \\ & \nu\langle X::\ X\circ R\rangle\subseteq \bot\!\!\!\bot\quad . \end{array}$$

□

14 It suffices to show that if R is well-founded then $R\cap I\subseteq \bot\!\!\!\bot$ because R is well-founded equivales R^+ is well-founded.

$$\begin{array}{ll} & R\cap I\subseteq \bot\!\!\!\bot \\ \Leftarrow & \quad \{\quad \text{assumption, } R \text{ is well-founded}\quad \} \\ & R\cap I\subseteq (R\cap I)\circ R \\ \Leftarrow & \quad \{\quad R\cap I\subseteq R\text{, monotonicity and transitivity}\quad \} \\ & R\cap I\subseteq (R\cap I)\circ(R\cap I) \\ \equiv & \quad \{\quad \text{for all coreflexives } A\text{, } A=A\circ A\text{.} \\ & \qquad R\cap I \text{ is a coreflexive}\quad \} \\ & \mathsf{true}\quad . \end{array}$$

Well-foundedness of R is equivalent to $R^+\cap I\subseteq \bot\!\!\!\bot$ when R is a relation on a finite set. (This is proved by induction on the size of the set.) When R is a relation on an infinite set the two conditions are not equivalent. For example, the less-than ordering on integers is not well-founded but its intersection with the identity relation is empty.
□

19 We use 17 as definition of admitting induction: with A denoting $\mu((R\circ\mu(R\backslash_!))\backslash_!)$ we must show $I=A$, or equivalently (because A is a coreflexive) $I\subseteq A$. The proof has two phases. In the first phase we reduce the goal to the formally weaker $\mu(R\backslash_!)\subseteq A$.

$$\begin{array}{ll} & I\subseteq A \\ \equiv & \quad \{\quad \text{definition of } A\text{, } \mu F=F.\mu F\quad \} \\ & I\subseteq (R\circ\mu(R\backslash_!))\backslash_! A \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \quad \text{monotype factors} \quad \} \\
& (R \circ \mu(R \backslash_{\prec})) \prec \subseteq A \\
\equiv & \{ \quad \mu(R \backslash_{\prec}) \text{ is a monotype} \quad \} \\
& (R \circ \mu(R \backslash_{\prec}) \circ \mu(R \backslash_{\prec})) \prec \subseteq A \\
\equiv & \{ \quad \text{monotype factors; definition of } A \,;\ \mu F = F \,.\, \mu F \quad \} \\
& \mu(R \backslash_{\prec}) \subseteq A
\end{array}
$$

Now, in the second phase, we have the opportunity to apply Knaster-Tarski.

$$
\begin{array}{ll}
& \mu(R \backslash_{\prec}) \subseteq A \\
\Leftarrow & \{ \quad \text{Knaster-Tarski} \quad \} \\
& R \backslash_{\prec} A \subseteq A \\
\equiv & \{ \quad \text{definition of } A \,;\ \mu F = F \,.\, \mu F \quad \} \\
& R \backslash_{\prec} A \subseteq (R \circ \mu(R \backslash_{\prec})) \backslash_{\prec} A \\
\Leftarrow & \{ \quad \text{monotype factor is anti-monotone in left argument} \quad \} \\
& R \circ \mu(R \backslash_{\prec}) \subseteq R \\
\equiv & \{ \quad \mu(R \backslash_{\prec}) \subseteq I \,;\ \text{monotonicity of composition} \quad \} \\
& \mathsf{true}
\end{array}
$$

□

22 We first show that $R \backslash S = \neg(R^{\cup} \circ \neg S)$. We use indirect equality.

$$
\begin{array}{ll}
& X \subseteq \neg(R^{\cup} \circ \neg S) \\
\equiv & \{ \quad \text{complements} \quad \} \\
& \neg X \supseteq R^{\cup} \circ \neg S \\
\equiv & \{ \quad \text{middle exchange} \quad \} \\
& S \supseteq R \circ X \\
\equiv & \{ \quad \text{factors} \quad \} \\
& X \subseteq R \backslash S \quad .
\end{array}
$$

Now we construct function f as suggested:

$$
\begin{array}{ll}
& \nu(\circ R) = f.\mu(R \backslash) \\
\Leftarrow & \{ \quad \text{fusion, assuming } f \text{ is antimonotonic} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
 & \forall\langle X :: \ f.X \circ R = f.(R\backslash X)\rangle \\
\equiv & \quad \{ \quad R\backslash S = \neg(R^{\cup} \circ \neg S) \quad \} \\
 & \forall\langle X :: \ f.X \circ R \ = \ f \,.\, \neg(R^{\cup} \circ \neg X)\rangle \\
\Leftarrow & \quad \{ \quad \text{try } \ f.X = g \,.\, \neg X \quad \} \\
 & \forall\langle Y :: \ g.Y \circ R = g.(R^{\cup} \circ Y)\rangle \\
\Leftarrow & \quad \{ \quad \text{converse} \quad \} \\
 & \forall\langle Y :: \ g.Y = Y^{\cup}\rangle \quad .
\end{array}
$$

Thus, $\nu(\circ R) \ = \ \neg\mu(R\backslash)^{\cup}$. For the final step, we have

$$
\begin{array}{ll}
 & \nu(\circ R) \subseteq \bot\!\!\!\bot \\
\equiv & \quad \{ \quad \text{above} \quad \} \\
 & \neg\mu(R\backslash)^{\cup} \ \subseteq \ \bot\!\!\!\bot \\
\equiv & \quad \{ \quad \text{complement, converse} \quad \} \\
 & \top\!\!\top^{\cup} \subseteq \mu(R\backslash) \\
\equiv & \quad \{ \quad \text{converse} \quad \} \\
 & \top\!\!\top \subseteq \mu(R\backslash) \quad .
\end{array}
$$

The interpretation of $\mu(R\backslash)$ is the set of all points such that all chains starting in such a point are guaranteed to be of finite length.
□

52 By instantiating S to the identity relation, it is clear that the follows-from property is true. We only need to prove the implication. So assume that R is well-founded. Then, for all X and S, we have:

$$
\begin{array}{ll}
 & X = S \circ X \circ R \\
\Rightarrow & \quad \{ \quad \text{Leibniz} \quad \} \\
 & \top\!\!\top \circ X = \top\!\!\top \circ S \circ X \circ R \\
\Rightarrow & \quad \{ \quad \top\!\!\top \circ S \subseteq \top\!\!\top \quad \} \\
 & \top\!\!\top \circ X \subseteq \top\!\!\top \circ X \circ R \\
\Rightarrow & \quad \{ \quad \text{assumption: } R \text{ is well-founded. So } \nu\langle X :: X \circ R\rangle = \bot\!\!\!\bot \quad \} \\
 & \top\!\!\top \circ X \subseteq \bot\!\!\!\bot \\
\Rightarrow & \quad \{ \quad \bot\!\!\!\bot \subseteq X \subseteq \top\!\!\top \circ X \quad \}
\end{array}
$$

$$
\begin{array}{ll}
& X = \bot\!\!\bot \\
\equiv & \quad \{ \quad \bot\!\!\bot = S \circ \bot\!\!\bot \circ R \text{ and } \bot\!\!\bot \subseteq Y \text{ for all } Y \quad \} \\
& X = \mu\langle Y :: S \circ Y \circ R \rangle \\
\Rightarrow & \quad \{ \quad \text{computation rule} \quad \} \\
& X = S \circ X \circ R \quad .
\end{array}
$$

□

53 Suppose $R \in F.I \leftarrow I$ is F-reductive. Suppose $S \in J \leftarrow F.J$ and that $X \in J \leftarrow I$ satisfies

$$X = S \circ F.X \circ R \quad .$$

We have to show that $X = \mu\langle Y :: S \circ F.Y \circ R \rangle$.

$$
\begin{array}{ll}
& X = \mu\langle Y :: S \circ F.Y \circ R \rangle \\
\equiv & \quad \{ \quad X = S \circ F.X \circ R \text{, definition of } \mu \quad \} \\
& X \subseteq \mu\langle Y :: S \circ F.Y \circ R \rangle \\
\equiv & \quad \{ \quad R \in F.I \leftarrow I \text{ is } F\text{-reductive. So } \mu((R \backslash\!\!\cdot)\bullet F) = \mathrm{id}_I \, . \quad \} \\
& X \circ \mu((R \backslash\!\!\cdot)\bullet F) \subseteq \mu\langle Y :: S \circ F.Y \circ R \rangle \\
\Leftarrow & \quad \{ \quad \text{fusion} \quad \} \\
& \forall\langle A :\ A \subseteq I :\ X \circ R \backslash\!\!\cdot F.A \ \subseteq \ S \circ F.(X \circ A) \circ R \rangle \\
\equiv & \quad \{ \quad \text{assumption: } X = S \circ F.X \circ R \text{,} \\
& \qquad F \text{ distributes through composition} \quad \} \\
& \forall\langle A :\ A \subseteq I :\ S \circ F.X \circ R \circ R \backslash\!\!\cdot F.A \ \subseteq \ S \circ F.X \circ F.A \circ R \rangle \\
\Leftarrow & \quad \{ \quad \text{monotonicity} \quad \} \\
& \forall\langle A :\ A \subseteq I :\ R \circ R \backslash\!\!\cdot F.A \ \subseteq \ F.A \circ R \rangle \\
\equiv & \quad \{ \quad \text{factors: (4)} \quad \} \\
& \text{true} \quad .
\end{array}
$$

□

References

[Bac86] R.C. Backhouse. *Program Construction and Verification*. Prentice-Hall International, 1986.

[BBH+92] R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C.S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 303–326. Springer-Verlag, Workshops in Computing, 1992.

[BBM+91] R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, and J. van der Woude. Relational catamorphisms. In Möller B., editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers B.V., 1991.

[BH99] Roland Backhouse and Paul Hoogendijk. Final dialgebras: From categories to allegories. *Theoretical Informatics and Applications*, 33(4/5):401–426, 1999.

[BJJM99] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming. An introduction. In S.D. Swierstra, editor, *3rd International Summer School on Advanced Functional Programming, Braga, Portugal, 12th-19th September, 1998*, volume LNCS 1608, pages 28–115. Springer Verlag, 1999.

[BW93] R.C. Backhouse and J. van der Woude. Demonic operators and monotype factors. *Mathematical Structures in Computer Science*, 3(4):417–433, December 1993.

[DB95] Henk Doornbos and Roland Backhouse. Induction and recursion on datatypes. In B. Möller, editor, *Mathematics of Program Construction, 3rd International Conference*, volume 947 of *LNCS*, pages 242–256. Springer-Verlag, July 1995.

[DB96] Henk Doornbos and Roland Backhouse. Reductivity. *Science of Computer Programming*, 26(1–3):217–236, 1996.

[DBvdW97] H. Doornbos, R.C. Backhouse, and J. van der Woude. A calculation approach to mathematical induction. *Theoretical Computer Science*, (179):103–135, 1997.

[Doo96] H. Doornbos. *Reductivity arguments and program construction*. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, June 1996.

[Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, The Netherlands, 1992.

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

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[HdM00] Paul Hoogendijk and Oege de Moor. What is a container type? *Journal of Functional Programming*, 2000. to appear.

[HH86] C.A.R. Hoare and Jifeng He. The weakest prespecification. *Fundamenta Informaticae*, 9:51–84, 217–252, 1986.

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

[Lam68] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

[Mal90a] G. Malcolm. *Algebraic data types and program transformation*. PhD thesis, Groningen University, 1990.

[Mal90b] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, October 1990.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[MFP91] Eric Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA '91: Functional Programming Languages and Computer Architecture*, number 523 in LNCS, pages 124–144. Springer-Verlag, 1991.

[SdM93] Doaitse Swierstra and Oege de Moor. Virtual data structures. In Helmut Partsch, Bernhard Möller, and Steve Schuman, editors, *Formal Program Development*, volume 755 of *LNCS*, pages 355–371. Springer-Verlag, 1993.