Mathematics of Recursive Program Construction

Henk Doornbos* Roland Backhouse†‡

July 24, 2001

Abstract

A discipline for the design of recursive programs is presented. The core concept of the discipline is a new induction principle. The principle is based on a property of relations, called reductivity, that generalises the property of admitting induction to one relative to a given datatype. The principle is used to characterise a broad class of recursive equations that have a unique solution and is also related to standard techniques for proving termination of programs. Methods based on the structure of the given datatype for constructing reductive relations are discussed.

1 Introduction

The central issue of 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. For example, an insight that played an important role in the development of a methodology for sequential programs is that it is possible to restrict, without loss of generality, the attention to the class of **while** programs only. It is not necessary to consider and, indeed, it is not desirable to consider arbitrary goto programs.

In the same spirit, in the methodology for functional programs, arbitrary recursion (which has been called the goto of functional programming) has been replaced by structural recursion. However, the problem is that this is a real restriction: not all programs can be expressed using this type of recursion.

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

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

‡Most of the work reported in this paper was done when the authors were members of the Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands.

In this paper we present a methodology that is an integration and extension of the two methodologies. The class of programs that can be developed using the integrated methodology is sufficiently restricted that it is possible to develop a practical design discipline, but sufficiently unrestricted that it is still general enough to express all algorithms in an uncluttered and natural style. In particular, we do not restrict our attention to structural recursion. On the other hand, arbitrary recursion is too general because allowing it would make it hard to design a methodology, in the same way as it is hard to design a methodology for the construction of programs with goto statements.

The programs in the class on which our methodology is based are called *hylomorphisms*. The fact that many recursively defined functional programs are hylomorphisms was identified by Fokkinga, Meijer and Paterson [22]. Unlike [22], however, the current paper is not restricted to the functional programming paradigm. Indeed, a central contribution of the current paper is to combine ideas from sequential programming and functional programming using relation algebra as the vehicle. From sequential programming we borrow the use of weakest preconditions and well-founded relations, and from functional programming we borrow the use of datatypes and (initial) algebras. These are combined in the notion of an "F-reductive relation", which is basically the generalisation of the notion of well-founded to well-founded-in-the-context-of-datatype-F.

The methodology we propose has a very solid foundation in category theory and allegory theory. (Allegory theory extends category theory in the sense that if category theory is viewed as the theory of functions between sets then allegory theory is the theory of relations on sets.) To gain a full understanding the reader needs to have a good grasp of how the categorical notions of "functor", "natural transformation" and "initial algebra" relate to type constructors, polymorphic functions and inductive datatypes, respectively, in functional programming, and how the allegorical notion of "division" relates to weakest preconditions in sequential programming. These are not difficult concepts but we appreciate that our intended audience —the practising programmer rather than the theoretician— will rarely have knowledge of all these topics. To increase the accessibility of the paper we have therefore included short summaries of the relevant theory, but interspersed through the text. For more background in line with the presentation here see [6, 5].

We begin by explaining the basics of relation algebra. Relation algebra is a point-free formulation of the algebra of binary relations which is distinct from the more usual pointwise formulations in the predicate calculus. This is followed in section 3 by a discussion of how the paradigm of loop construction via invariant properties is expressed in relation algebra. The domain and division operators are the main theoretical tools introduced in these two sections. (The two types of "factor" of a relation are better known in computing science under the names weakest precondition and weakest prespecification.)

Subsequently we focus on the construction of recursive programs. Via several well-known examples, beginning with programs defined by structural recursion and continuing on through programs defined by primitive recursion to more complicated recursion schemes, we argue the case for developing a discipline of recursive programming restricted to the class of so-called “hylo” programs. In other words, we provide empirical evidence for the claim that the class of hylo programs is sufficiently general to be both practical and useful. Section 6.3, on the other hand, provides theoretical support for the usefulness of hylo programs. In this section we present the “hylo theorem”. This theorem states that a hylo program can be viewed as a two-step process. The first step creates a data structure (for example a tree or a list) which is then broken down in the second step. This intermediate structure is sometimes called a “virtual” data structure because the usual implementation of a hylo program does not explicitly build up the data structure. Just like the stream of values in a Unix pipe, the construction and destruction of the intermediate data structure occur simultaneously. This use of virtual data structures is the basis of the programming paradigm proposed in section 6.4.

Central to the design of loops in sequential programming is making progress to the termination condition. Making progress is achieved by ensuring that the loop body is a well-founded relation on its input and output values. Making progress is, of course, also central to the design of recursive programs. In section 7 we propose the notion of “F-reductivity” as the appropriate generalisation of well-foundedness in the context of a programming design methodology based on the construction of hylo programs. The study of F-reductivity is the primary innovative contribution of this paper. We provide both theoretical and empirical support for the importance of this notion and, in section 8, we present a calculus of F-reductivity which is applied to establish termination of a large collection of recursive programs.

Many of the results we present are “generic” in a type constuctor F. That is, the results are valid independently of the value chosen for F (be it List, Maybe or whatever). To provide further support for the importance of a generic, relational theory of datatypes, section 10 considers a quite different application of the reductivity calculus developed in section 8. The application concerns the properties of the “occurs-properly-in” relation in a generic unification algorithm. Here we show how the relation is defined generically and how it is then proved to be well-founded. The remarkable aspect of the proof, apart from its simplicity, is that it does not involve any appeal to the properties of natural numbers whatsoever. The core of the proof is the theorem that the converse of an initial F-algebra is F-reductive.

2 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 $m\langle\mathsf{succ}\rangle n$ 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. Our goal here is to combine the functional and imperative programming paradigms. For this reason, we adopt a typed algebra of relations (formally an "allegory" [13]). 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 $R :: I{\leftarrow}J$ (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 $x\langle R\rangle y$ if the pair (x,y) is an element of relation R. We use a raised infix dot to denote relational composition. Thus $R \cdot S$ denotes the composition of relations R and S (the relation defined by $x\langle R \cdot S\rangle z$ equivales $\exists(y{::}\ x\langle R\rangle y \wedge y\langle S\rangle z)$). The composition $R \cdot S$ is only defined when the source of R equals the target of S. Moreover, the target of $R \cdot S$ is the target of R, and the source of $R \cdot S$ is the source of S. Thus, $R \cdot S :: I{\leftarrow}K$ if $R :: I{\leftarrow}J$ and $S :: J{\leftarrow}K$. The converse of relation R is denoted by $R^{\cup}$. Thus, $x\langle R^{\cup}\rangle y$ equivales $y\langle R\rangle x$. The type rule is that $R^{\cup} :: I{\leftarrow}J$ equivales $R :: J{\leftarrow}I$.

Relations of the same type are ordered by set inclusion denoted in the conventional way by the infix $\subseteq$ operator. We assume that 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.

In previous publications on this topic [9, 10] we have assumed an untyped framework. The use of a typed framework makes some proofs easier. This is because the type restrictions sometimes strengthen the premises of a lemma or theorem. It can also be disadvantageous — mainly because the type restrictions are implicit and it is difficult to see when and where they are really necessary. One nuisance is that each use of the empty and/or universal relation should be properly typed whereas in most cases the type information is not relevant (although in a few cases it is!). In line with other authors we shall therefore often omit the type information, leaving the reader to infer which type is intended. This applies also to natural transformations — see section 4.1.

For each set I there is an identity relation on I which we denote by id_I. Thus $\mathsf{id}_I :: I \leftarrow I$. 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 $x\langle A\rangle y$ 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 $x\langle A\rangle x$ (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 $x\langle A\rangle x$. 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, $x\langle \sim A\rangle y$ equivales the conjunction of $x \in I$ (where A has type $I \leftarrow I$) and $x = y$ and not $x\langle A\rangle x$. 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. See section 4.1.) A final, important remark about coreflexives is that their composition coincides with their intersection. That is, for coreflexives A and B, $A \cdot B = A \cap B$.

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 $y\langle R\rangle x$ and $z\langle R\rangle x$ together imply that $y = z$. If this is the case we write $R.x$ for the unique y such that $y\langle R\rangle x$. Note that functionality of relation R is equivalent to the property $R \cdot R^{\cup} \subseteq \mathsf{id}_I$ 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 $R^{\cup} \cdot R \subseteq \mathsf{id}_J$. Which of the properties $R \cdot R^{\cup} \subseteq \mathsf{id}_I$ or $R^{\cup} \cdot R \subseteq \mathsf{id}_J$ 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.

3 Imperative Programming

The discipline of imperative programming is embodied in so-called "Hoare logic" which is a body of inference rules, one for each statement type in the language. In this section we formulate the methodology of sequential program construction in terms of relation algebra, focusing on the design of loop structures. Two sets of operators are introduced, the domain operators and the division operators. The domain operators subsume the role of assertions in Hoare logic, and the division operators subsume weakest preconditions and prespecifications.

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 in relation algebra. 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

$$Term \cdot Inv \subseteq X \quad .$$

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

$$\forall(v{:}\ w'\langle Term\rangle v \wedge v\langle Inv\rangle w{:}\ w'\langle X\rangle w)$$

(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

$$Body \cdot Inv \subseteq Inv$$

Pointwise this is

$$\forall(w', v, w{:}\ \ w'\langle Body\rangle v \wedge v\langle Inv\rangle w{:}\ \ w'\langle Inv\rangle w) \quad .$$

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

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

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

As an example, consider the problem of reversing a list. We assume that lists are built from the empty list, nil, by adding elements to the head of the list via the cons operation (which we denote by " : " as in Haskell when used in infix form). Conversely, the function head returns the head (i.e. first) element of a non-empty list, and tail returns the remainder of the list. Thus if x is an element and xs is a list, $x{:}xs$ is the list obtained by adding x at the head of xs. Moreover, $\mathsf{head}.(x{:}xs)$ is x and $\mathsf{tail}.(x{:}xs)$ is xs.

A standard iterative technique for reversing a list xs uses an additional list ys to accumulate the reversed list. The state space of the program is thus $\mathsf{List}.I\times\mathsf{List}.I$ for some element type I. The specification, invariant, initialisation and termination are thus binary relations on this set. The specification, X, is simply

$$ys' = \mathsf{reverse}.xs \quad .$$

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

$$\{xs', ys', xs, ys\colon\; ys' = \mathsf{reverse}.xs\colon\; ((xs',ys')\,,(xs,ys))\} \quad .$$

The convention is that the definition

$$\{xs', ys', xs, ys\colon\; p.(xs',ys',xs,ys)\colon\; ((xs',ys')\,,(xs,ys))\}$$

is abbreviated to

$$p.(xs',ys',xs,ys) \quad ,$$

the primes indicating the correspondence between input and output variables.

Initially ys is empty and the program terminates when xs is empty. Thus (using the convention explained above) the initialisation Init is the relation

$$xs' = xs \;\wedge\; ys' = \mathsf{nil} \quad .$$

The termination is a subset of the identity relation on the state space. It is the relation

$$xs' = xs = \mathsf{nil} \;\wedge\; ys' = ys \quad .$$

The invariant is the relation

$$\mathsf{reverse.xs}' + \!\!+\; ys' \;=\; \mathsf{reverse.xs} + \!\!+\; ys$$

where $+\!\!+$ is the operation that "joins" (or concatenates) two lists together. Since $\mathsf{reverse.nil}$ is nil and $\mathsf{nil} +\!\!+ ys'$ is ys', the composition of the termination relation and the invariant is thus the relation

$$ys' \;=\; \mathsf{reverse.xs} +\!\!+\; ys \quad .$$

Moreover, since $zs +\!\!+ \mathsf{nil}$ is zs (for all zs), the composition of this relation with the initialisation is thus the specification X.

Maintaining the invariant whilst making progress to the termination condition that xs is nil is based on the identity

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

This is achieved by the relation $Body$ characterised by

$$xs' = \mathsf{tail.xs} \;\wedge\; ys' = (\mathsf{head.xs}){:}ys$$

(i.e. by the assignment $xs,ys \;:=\; \mathsf{tail.xs}\,,(\mathsf{head.xs}){:}ys$). The program to compute the reverse of a list is thus as follows:

$ys \;:=\; \mathsf{nil};$

while $xs \neq \mathsf{nil}$ **do** $xs,ys \;:=\; \mathsf{tail.xs}\,,(\mathsf{head.xs}){:}ys$

3.1 The Domain Operators

Our account of invariants is not yet complete. The relationship between the specification X and $Term \cdot Body^* \cdot Init$ is containment not equality. The example of the reverse procedure shows that it 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 $Term \cdot Body^*$ 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

$$\forall (A\colon\; A \subseteq \mathrm{id}_J\colon\; R \cdot A = R \equiv R{>} \subseteq A) \quad .$$

Given a coreflexive A, $A \subseteq \mathsf{id}_J$, the relation $R \cdot A$ 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.

In the case of the program to reverse a list presented above, the right domain of $Term$ is $Term$ itself. (Recall that $Term$ was defined to be a subset of the identity relation, i.e. a coreflexive.) It is the coreflexive corresponding to the set of pairs of lists of which the first is empty. The right domain of $Body$ is the coreflexive corresponding to pairs of lists of which the first is non-empty. This is because the right domain of the head and tail functions is the coreflexive corresponding to the set of non-empty lists. The right domain of $Body$ and (the right domain of) $Term$ are thus indeed complementary.

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.

The right domain operator and its dual left domain operator play a prominent role in the effective use of relation algebra, but this is not the place for us to go into them in depth. For the most part, their calculational properties are reasonably self-evident from their pointwise interpretation. We refer to the properties in calculations by the hint "domain calculus" and refer the reader to other publications for more extensive discussion. One property that is not so self-evident but figures greatly in our calculations is that the right domain operator is a lower adjoint in a Galois connection. Specifically,

$$\forall (R, A\colon\ A \subseteq \mathsf{id}_J\colon\ R \subseteq \top\!\!\top \cdot A \equiv R{>} \subseteq A)$$

where $\top\!\!\top$ denotes the universal relation of type $I \leftarrow J$. The significance of this property is its use in combination with the fusion theorem on fixed points [24].

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 $b \cup {\sim b} = \mathsf{id}$ and $b \cap {\sim b} = \perp\!\!\perp$) we thus have

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

As a consequence,

$$Term \cdot Body^* \cdot Init \ = \ Term \cdot {\sim b} \cdot (Body \cdot b)^* \cdot Init \ .$$

The statement **while** b **do** $Body$ is the implementation of ${\sim b} \cdot (Body \cdot b)^*$. If $Body \cdot b$ is well-founded, ${\sim b} \cdot (Body \cdot b)^*$ is the unique solution [8] of the equation:

$$X\!::\ X = {\sim b} \cup X \cdot Body \cdot b \ .$$

Executing this equation is equivalent to executing the program

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

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. The proof illustrates the sort of calculations we make with the domain operators. We have:

$$\begin{array}{ll}
& (\mathit{Term} \cdot \mathit{Body}^* \cdot \mathit{Init})> \;=\; \mathsf{id}_I \\
\equiv & \quad \{ \quad \text{domain calculus} \quad \} \\
& ((\mathit{Term} \cdot \mathit{Body}^*)> \cdot \mathit{Init})> \;=\; \mathsf{id}_I \\
\Leftarrow & \quad \{ \quad \text{by assumption, } \mathit{Init} \text{ is total, i.e. } \mathit{Init}> = \mathsf{id}_I \quad \} \\
& (\mathit{Term} \cdot \mathit{Body}^*)> \;=\; \mathsf{id}_I \\
\equiv & \quad \{ \quad (\mathit{Term} \cdot \mathit{Body}^*)> \text{ is the unique solution of the equation} \\
& \qquad\quad A{::}\ A \;=\; \mathit{Term}> \cup (A \cdot \mathit{Body})> \quad \} \\
& \mathsf{id}_I \;=\; \mathit{Term}> \cup (\mathsf{id}_I \cdot \mathit{Body})> \\
\equiv & \quad \{ \quad \text{by assumption,} \\
& \qquad\quad \mathit{Term} \text{ and } \mathit{Body} \text{ have complementary right domains.} \\
& \qquad\quad \text{In particular, } \mathsf{id}_I \;=\; \mathit{Term}> \cup \mathit{Body}> \quad \} \\
& \mathsf{true} \quad .
\end{array}$$

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

$$A{::}\ \ A \;=\; \mathit{Term}> \cup (A \cdot \mathit{Body})>$$

has a unique solution provided that Body is well-founded. As mentioned earlier, it is proved in [8] that the equation

$$X{::}\ \ X \;=\; R \cup X \cdot S$$

has a unique solution if relation S is well-founded. Thus, for all coreflexives A,

$$\begin{array}{ll}
& A \;=\; \mathit{Term}> \cup (A \cdot \mathit{Body})> \\
\equiv & \quad \{ \quad \text{domain calculus.} \\
& \qquad\quad \text{Specifically, } (\top\!\!\top \cdot A)> = A \text{ and } \top\!\!\top \cdot R \;=\; \top\!\!\top \cdot R> \quad \} \\
& \top\!\!\top \cdot A \;=\; \top\!\!\top \cdot \mathit{Term} \cup \top\!\!\top \cdot A \cdot \mathit{Body}
\end{array}$$

$$\equiv \quad \{ \quad \mathsf{Body} \text{ is well-founded} \quad \}$$

$$\top\!\top \cdot A = \top\!\top \cdot \mathsf{Term} \cdot \mathsf{Body}^*$$

$$\equiv \quad \{ \quad \text{domain calculus (as above)} \quad \}$$

$$A = (\mathsf{Term} \cdot \mathsf{Body}^*)_> \quad .$$

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

3.2 Factors — alias Weakest Liberal Preconditions

The body of a loop should maintain the loop invariant. Formally, the requirement is that $\mathsf{Body} \cdot Inv \subseteq Inv$. 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,

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

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

$$\forall (v {:}\; v \langle R \rangle w' {:}\; v \langle T \rangle w) \quad .$$

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

$$Inv \subseteq \mathsf{Body} \backslash Inv \quad ,$$

the pointwise formulation of which is

$$\forall (w', w {:}\; w' \langle Inv \rangle w {:}\; \forall (w'' {:}\; w'' \langle \mathsf{Body} \rangle w' {:}\; w'' \langle Inv \rangle w)) \quad .$$

The relation $\mathsf{Body} \backslash Inv$ corresponds to what is called the *weakest prespecification* of Inv with respect to Body in the more usual predicate calculus formulations of the methodology [14]. The *weakest liberal precondition* operator will be denoted here by the symbol " $\searrow$ ". Formally, if R is a relation of type $I \leftarrow J$ and A is a coreflexive of type $I \leftarrow I$ then $R \searrow A$ is a coreflexive of type $J \leftarrow J$ characterised by the property that, for all coreflexives B of type $J \leftarrow J$,

(1) $\quad (R \cdot B)_< \subseteq A \equiv B \subseteq R \searrow A \quad .$

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

$$q.w \equiv \forall (w' {:} w' \langle R \rangle w {:} p.w) \quad .$$

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 $\backslash$ 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 $(R\backslash)$, 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: $R\backslash(A \cdot B) = (R\backslash A) \cdot (R\backslash B)$. This corresponds to the fact that the weakest liberal precondition operator associated with a statement R is universally conjunctive. From (1) we obtain the *cancellation* property:

(2) $\quad (R \cdot R\backslash B)< \subseteq B \quad .$

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

(3) $\quad R \cdot R\backslash B \subseteq B \cdot R \quad .$

Both (2) and (3) 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:

(4) $\quad f\backslash A = f^{\cup} \cdot A \cdot f \quad .$

A full discussion, including all the properties used here, can be found in [3]. Finally, we recall that relation R of type $I \leftarrow I$ is well-founded equivales that

$$\mu(A \mapsto R\backslash A) = \mathsf{id}_I \quad .$$

(Here and elsewhere, μ denotes the least fixed point operator. See [8] for a detailed discussion of how well-foundedness is expressed in terms of least fixed points.) Much of this paper is in fact about generalising this property to the situation where types play a prominent role.

4 Functional Programming

Functional programming is many things to many people. For us, functional programming is about user-defined (recursive) datatypes and polymorphic functions. The theory that embodies functional programming most adequately is category theory: type constructors (like List and Maybe) are functors, polymorphic functions (like the flatten function on lists of lists) are natural transformations and inductively defined datatypes are initial algebras. Allegory theory is a relatively minor extension to category theory that enriches functional programming with relation algebra.

In this section we briefly review these elements of category and allegory theory. At the same time we introduce some notation. The section is not meant to be an introduction to the categorical concepts mentioned above. For a gentle introduction in line with the presentation here see [5]. For a more extensive introduction see [6]

4.1 Functors, Relators and Natural Transformations

A *category* consists of a collection of *objects* and a collection of *arrows*. Each arrow has a *source* object and a *target* object. We write $f :: I \xleftarrow{\mathcal{C}} J$ if arrow f in category $\mathcal{C}$ has target I and source J. (Often $f :: I \leftarrow J$ is written if the category is fixed.) Arrows can be composed. If the source of arrow f is the target of arrow g then their composition $f \cdot g$ exists; its target is the target of f and its source is the source of g. That is, if $f :: I \leftarrow J$ and $g :: J \leftarrow K$ then $f \cdot g :: I \leftarrow K$. Finally, each object I in the category defines an arrow $\mathsf{id}_I :: I \leftarrow I$ which is a left and right identity of composition.

The canonical example of a category is $\mathcal{F}un$, the category with sets as objects and functions between sets as arrows. Another example of a category is $\mathcal{R}el$, the category with sets as objects and binary relations as arrows. The category $\mathcal{F}un$ is a subcategory of $\mathcal{R}el$.

Given two categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* with target $\mathcal{C}$ and source $\mathcal{D}$ comprises two mappings, a mapping from the objects of $\mathcal{D}$ to the objects of $\mathcal{C}$ and a mapping from the arrows of $\mathcal{D}$ to the arrows of $\mathcal{C}$. It is customary to give the mappings the same name. Using this convention, the requirements on a functor F are that if $f :: I \xleftarrow{\mathcal{D}} J$ then $F.f :: F.I \xleftarrow{\mathcal{C}} F.J$. Also, functors are required to preserve identities. That is, for each object I of category $\mathcal{D}$, $F.\mathsf{id}_I = \mathsf{id}_{F.I}$. Finally, functors preserve composition. That is, $F.(f \cdot g) = F.f \cdot F.g$ for all composable arrows f and g.

We shall mainly be dealing with endofunctors. An endofunctor is a functor whose source and target categories are the same. Occasionally we refer to n-ary functors. An n-ary functor is a functor with source category $\mathcal{C}^n$, the n-fold product of category $\mathcal{C}$ with itself, and target category $\mathcal{C}$. (Objects of $\mathcal{C}^n$ are n-tuples of objects of $\mathcal{C}$, and arrows of $\mathcal{C}^n$ are n-tuples of arrows of $\mathcal{C}$.)

Functors are relevant to functional programming because they correspond to type constructors. The canonical example is List, which is an endofunctor on the category $\mathcal{F}un$. The object part of the functor List is the mapping from types (sets) to types. (For example $\mathsf{List.Nat}$, lists of natural numbers, is the result of applying List to Nat.) The arrow part of the functor List is the function known as `map` to functional programmers. If $f :: I \leftarrow J$ then $\mathsf{map}.f :: \mathsf{List}.I \leftarrow \mathsf{List}.J$ 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.

Two binary functors that we will use extensively are cartesian product and disjoint sum. The cartesian product of types I and J is the type $I{\times}J$ consisting of pairs, the first component of which has type I and the second component of which has type J. This is the object part of the binary product functor on Fun. The arrow part is such that $f{\times}g$ applied to a pair (x,y) is the pair $(f.x\,,\,g.y)$. The disjoint sum of types I and J is the type $I{+}J$ consisting of values of type I and values of type J appropriately tagged to indicate whether they are in the left or right component of the sum. The arrow part is such that if f has type $I{\leftarrow}J$ and g has type $K{\leftarrow}L$ then applying $f{+}g$ to an element of type $J{+}L$ involves inspecting the tag, then applying f or g to the untagged value (as dictated by the tag) and finally reinstating the tag. In this way the value obtained is of type $I{+}K$.

Other functors that are important are the *identity functor* and the *constant functors*. The *identity functor* comprises the identity function on objects and the identity function on arrows. For each object I there is a *constant functor* that when applied to an object yields the object I and when applied to an arrow yields the identity arrow on I.

It is convenient not to distinguish notationally between the object I and its corresponding constant functor. This is particularly so when discussing functors formed by sectioning disjoint sum or cartesian product. For example, if $\mathbb{1}$ denotes the unit type (the type with exactly one element — both the type and its element are denoted by () in Haskell) then the *functor* $\mathbb{1}{+}$ maps the type A to the type $\mathbb{1}{+}A$. Also, 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. Formally, given endofunctors F and G their sum is the functor $F{+}G$ which yields $F.X + G.X$ when applied to an object X and yields $F.f + G.f$ when applied to an arrow f. In this way, the functor $\mathbb{1}{+}$ is the sum of the constant functor $\mathbb{1}$ and the identity functor.

As we have said, Rel is a category just like Fun. Also type constructors like List are endofunctors on Rel, just as they are on Fun. 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" [13] extends the notion of a category in order to better capture the essential properties of relations, and the notion of a "relator" [2, 3] 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 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:

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

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

(7) $\quad F.\mathsf{id}_A = \mathsf{id}_{F.A}$ for each object A,

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

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

Two examples of relators have already been given. List is a unary relator, and product is a binary relator. If R is a relation of type $I \leftarrow J$ then $\mathsf{List}.R$ relates a list of Is to a list of Js 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 [1] it was observed that all inductively-defined datatypes are relators.

A design requirement which lead to the above definition of a relator [1, 2] is that a relator should extend the notion of a functor but in such a way that it coincides with the latter notion when restricted to functions. Formally, relation $R :: I \leftarrow J$ is *total* iff

$$\mathsf{id}_J \supseteq R^\cup \cdot R \quad ,$$

and relation R is *single-valued* iff

$$R \cdot R^\cup \subseteq \mathsf{id}_I \quad .$$

A *function* is a relation that is both total and single-valued. It is easy to verify that total relations are closed under composition, as are single-valued relations. Hence, functions are closed under composition too. In other words, the functions form a sub-category. For an allegory $\mathcal{A}$, we denote the sub-category of functions by $\mathcal{M}ap(\mathcal{A})$. In particular, $\mathcal{M}ap(\mathcal{R}el)$ is the category $\mathcal{F}un$. Now the desired property of relators is that relator $F :: \mathcal{A} \leftarrow \mathcal{B}$ is a functor of type $\mathcal{M}ap(\mathcal{A}) \leftarrow \mathcal{M}ap(\mathcal{B})$. It is easily shown that our definition of relator guarantees this property.

(Bird and De Moor [6] observe that, under a number of conditions on the allegories concerned, a functor is monotonic if and only if it commutes with converse. A relator is thus a monotonic functor. The use of commutativity with converse is so ubiquitous that we prefer to stick to our original definition.)

Polymorphic functions play a major role in functional programming. An insight that has helped to increase the understanding of the relevance of category theory to functional programming is that polymorphic functions like the flatten function on lists are so-called "natural transformations".

Formally, a collection of arrows α_A indexed by objects (equivalently, a mapping α of objects to arrows) is *a natural transformation of type* $F \leftarrow G$, for functors F and G, iff for each object A, $\alpha_A :: F.A \leftarrow G.A$ and, for each pair of objects A and B and each arrow $f :: A \leftarrow B$,

$$F.f \cdot \alpha_B = \alpha_A \cdot G.f \quad .$$

For example, $\mathsf{flatten}$ is a natural transformation of type $\mathsf{List} \leftarrow \mathsf{List} \circ \mathsf{List}$ since, for each type A, the function $\mathsf{flatten}_A$ of type $\mathsf{List}.A \leftarrow \mathsf{List}.(\mathsf{List}.A)$ satisfies the naturality property

$$\mathsf{map}.f \cdot \mathsf{flatten}_B = \mathsf{flatten}_A \cdot \mathsf{map}.(\mathsf{map}.f) \quad \text{for each } f :: A \leftarrow B.$$

If we map function f to all values stored in a list of lists and then flatten the resulting list of lists to a list then this is the same as first flattening the list of lists and then applying f to each stored value. Another example of a natural transformation is the length function on lists. It has type $\mathsf{Nat} \leftarrow \mathsf{List}$ where Nat is the constant natural number functor and it is indeed the case that

$$\mathsf{length}_B = \mathsf{length}_A \cdot \mathsf{map}.f \quad \text{for each } f \ :: \ A \leftarrow B.$$

(Note that $\mathsf{Nat}.f$, the constant functor Nat applied to f, is the identity function on numbers and so disappears from the equation.) This says that mapping a function over a list does not alter its length.

A remarkable theorem due to Reynolds [26] and promulgated by Wadler [28] is that it is possible to deduce a naturality property from the type of a polymorphic function. Interestingly, the formulation of Reynolds' theorem involves extending endofunctors on $\mathcal{F}un$ to endorelators on the allegory $\mathcal{R}el$. However, many collections of relations are not natural with equality but with an inclusion. They are sometimes called *lax natural transformations*. The collection of lax natural transformations to relator F from G is denoted by $F \hookleftarrow G$ and defined by

$$(10) \qquad \alpha \ :: \ F \hookleftarrow G \ \equiv \ (F.R \cdot \alpha_J \supseteq \alpha_I \cdot G.R \quad \text{for each } R \ :: \ I \leftarrow J) \ .$$

A relationship between naturality in the allegorical sense and in the categorical sense is given by two lemmas [15]. Recall that relators respect functions, i.e. relators are functors on the sub-category $\mathcal{M}ap$. The first lemma states that an allegorical natural transformation is a categorical natural transformation:

$$(F.f \cdot \alpha_J = \alpha_I \cdot G.f \quad \text{for each function } f \ :: \ I \leftarrow J) \ \Leftarrow \ \alpha \ :: \ F \hookleftarrow G \ .$$

The second lemma states the converse; the lemma is valid under the assumption that the source allegory of the relators F and G is tabular:

$$\alpha \ :: \ F \hookleftarrow G \ \Leftarrow \ (F.f \cdot \alpha_J = \alpha_I \cdot G.f \quad \text{for each function } f \ :: \ I \leftarrow J) \ .$$

In the case that all elements of the collection α are *functions* we thus have:

$$\alpha \ :: \ F \hookleftarrow G \text{ in } \mathcal{A} \ \equiv \ \alpha \ :: \ F \leftarrow G \text{ in } \mathcal{M}ap(\mathcal{A})$$

where by "in X" we mean that all quantifications in the definition of the type of natural transformation range over the objects and arrows of X. This lemma means that the notion of "lax" natural transformation is the more appropriate allegorical extension of the categorical notion of natural transformation rather than being a natural transformation in the underlying category. Thus we shall not use the qualifier "lax". For us, a natural transformation is as defined by (10).

4.2 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:

(11) $\quad \mathsf{even} \cdot (\mathsf{zero} \triangledown \mathsf{succ}) = (\mathsf{true} \triangledown \mathsf{not}) \cdot (\mathbb{1} + \mathsf{even}) \quad .$

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 $A \leftarrow B{+}C$.

As explained earlier, the term $\mathbb{1}{+}\mathsf{even}$ is read as the *functor* $\mathbb{1}{+}$ applied to the function even. It is the function that inspects the tag on a value of type $\mathbb{1}{+}\mathsf{Nat}$ to see if it belongs to the left component, $\mathbb{1}$, or the right component, Nat. In the former case the value is left unaltered (complete with tag), and in the latter case the function even is applied to the untagged value, and then the tag is replaced. The functor $\mathbb{1}{+}$ is called the *pattern functor* of the datatype Nat [5].

The final aspect of (11) that is crucial is that it uniquely defines the function even. (To be precise, the equation

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

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

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

has exactly one solution.

In summary, category theory identifies three vital ingredients in the definition (11) of the function even, namely, the functor $\mathbb{1}+$, the initial $(\mathbb{1}+)$-algebra $\mathsf{zero} \triangledown \mathsf{succ}$ and the $(\mathbb{1}+)$-algebra $\mathsf{true} \triangledown \mathsf{not}$.

The general form exemplified by (11) is

$$(12) \qquad X \cdot \mathsf{in} = f \cdot F.X$$

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 $\mathsf{zero} \triangledown \mathsf{succ}$. 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 13 Suppose F is an endofunctor on some category $\mathcal{C}$. An arrow f in $\mathcal{C}$ is an F-*algebra* if $f :: A \leftarrow F.A$ 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 $\varphi :: A \leftarrow B$ is said to be an F-algebra *homomorphism* to f from g if $\varphi \cdot f = g \cdot F.\varphi$. 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 $\mathsf{in} :: I \leftarrow F.I$ is an *initial* F-algebra if for each $f :: A \leftarrow F.A$ there exists an arrow $(\![f]\!) :: A \leftarrow I$ such that for all $h :: A \leftarrow I$,

$$(14) \qquad h = (\![f]\!) \;\equiv\; h :: f \xleftarrow{\mathsf{FAlg}} \mathsf{in} \quad .$$

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 [19, 20]. 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 [18], is that an initial algebra is both injective and surjective. Thus, for example, $\mathsf{zero}\triangledown\mathsf{succ}$ is an isomorphism between Nat and $1\!\!1+\mathsf{Nat}$. Lambek's lemma has the consequence that, if in is an initial F-algebra,

$$h\cdot\mathsf{in} = f\cdot F.h \;\equiv\; h = f\cdot F.h\cdot\mathsf{in}^{\cup}$$

where $\mathsf{in}^{\cup}$ is the inverse of in. Thus, the characterising property (14) of catamorphisms is equivalent to, for all h and all F-algebras f,

$$(15)\qquad h = (\![f]\!) \;\equiv\; h = f\cdot F.h\cdot\mathsf{in}^{\cup} \quad .$$

That is, $(\![f]\!)$ is the unique fixed point of the function mapping h to $f\cdot F.h\cdot\mathsf{in}^{\cup}$.

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 $1\!\!1+(I\times X)$ (where $\times$ denotes the cartesian product functor) and an F-algebra is a function of type $A\leftarrow 1\!\!1+(I\times A)$ for some A. Thus an F-algebra takes the form $c\triangledown(\oplus)$ for some function c of type $A\leftarrow 1\!\!1$ and some function $\oplus$ of type $A\leftarrow I\times A$. The characterising property of the catamorphisms is thus

$$h = (\![c\triangledown(\oplus)]\!) \;\equiv\; h \;=\; c\triangledown(\oplus)\;\cdot\;1\!\!1+(I\times h)\;\cdot\;\mathsf{nil}^{\cup}\blacktriangledown\mathsf{cons}^{\cup} \quad .$$

Here $\mathsf{nil}^{\cup}\blacktriangledown\mathsf{cons}^{\cup}$ is the inverse of $\mathsf{nil}\triangledown\mathsf{cons}$. (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 $\mathsf{nil}^{\cup}$ 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 $\mathsf{cons}^{\cup}$ 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

$$h = (\![c\triangledown(\oplus)]\!) \;\equiv\; h\cdot\mathsf{nil}=c \;\wedge\; h\cdot\mathsf{cons} = (\oplus)\cdot I\times h$$

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

$$(\![\mathsf{zero} \triangledown \mathsf{add}]\!)$$

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

$$\mathsf{nil}\langle\mathsf{prefix}\rangle\mathsf{nil}$$

and

$$xs\langle\mathsf{prefix}\rangle(y{:}ys) \quad \equiv \quad xs = \mathsf{nil} \ \vee \ \exists(zs{::} \ \ xs = y{:}zs \wedge zs\langle\mathsf{prefix}\rangle ys) \quad .$$

Expressed as one, point-free equation this is

(16) $\quad \mathsf{prefix} \cdot \mathsf{nil} \triangledown \mathsf{cons} \ = \ \mathsf{nil} \triangledown ((\mathsf{nil} \cdot \top\!\!\top) \cup \mathsf{cons}) \cdot \mathbb{1} + (\mathrm{I} \times \mathsf{prefix})$

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 $\mathsf{nil} \triangledown ((\mathsf{nil} \cdot \top\!\!\top) \cup \mathsf{cons})$. (Note that the second algebra is not a function.) Equivalently, this equation is:

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

The prefix relation is also an example of a relational natural transformation. Indeed, $\mathsf{prefix} \ :: \ \mathsf{List} \leftarrow \mathsf{List}$ in the *category* $\mathtt{Rel}$. That is, prefix is a *proper* natural transformation.

4.3 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

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

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

$$(18) \qquad \mathsf{fact} \ = \ \mathsf{one} \triangledown (\mathsf{times} \cdot \mathsf{succ} \times \mathsf{id}_{\mathsf{Nat}}) \ \cdot \ \mathsf{id}_{\mathbb{1}} + (\mathsf{id}_{\mathsf{Nat}} \vartriangle \mathsf{fact}) \ \cdot \ \mathsf{zero}^{\cup} \blacktriangledown \mathsf{succ}^{\cup} \qquad .$$

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 $\mathsf{id}_{\mathsf{Nat}} \vartriangle \mathsf{fact}$ constructs a pair consisting of the number n and the result of the factorial function applied to n. The calculation of $(n+1) * n!$ is the result of applying the function $\mathsf{times} \cdot \mathsf{succ} \times \mathsf{id}_{\mathsf{Nat}}$ to the (untagged) pair. On the other hand, if the input value is zero then one is returned as result.

The presence of the split operator ($\vartriangle$) in this definition is because of the fact that the result of the factorial function depends directly on the input in addition to the result of the recursive call: $(n+1)! = (n+1) * n!$, the input n appears twice in the right side. This means that the input n is copied. If one gives a pointwise definition this copying is done by writing n twice. However, in a point-free formulation the copying has to be coded by a function, in this case by the doubling function $\mathsf{id}_{\mathsf{Nat}} \vartriangle \mathsf{id}_{\mathsf{Nat}}$.

Instances of the doubling and similar functions occur when a point-wise formula is rewritten in a point-free form. The location and number of occurrences of the points in the point-wise formula have to be coded by a function in the point-free formula. Such relations are called (somewhat derogatively) *plumbing*. A much more impressive name is "natural transformation". Natural transformations map structures to structures, moving the stored data around without doing any computation on it.

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

$$\mathsf{nil}\langle \mathsf{suffix} \rangle \mathsf{nil}$$

and

$$xs \langle \mathsf{suffix} \rangle (y{:}ys) \quad \equiv \quad xs = y{:}ys \ \vee \ xs \langle \mathsf{suffix} \rangle ys \quad .$$

Expressed as a fixed point equation this is:

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

where I is the type of the list elements. This is a definition by primitive recursion. (Note that, as forewarned, we write I and $\mathsf{List}.\mathrm{I}$ here instead of the formally correct id_{I} and $\mathsf{id}_{\mathsf{List}.\mathrm{I}}$.)

Because of the presence of a plumbing, the form of equation (18) differs from the form of the cata equation (15) and the program is therefore not structurally recursive. The question is whether equation (18) is a definition, i.e. is there a unique solution? If not, we are obliged to define fact by a phrase like: "the factorial function is the least solution of the equation (18)". Of course it is well-known that this is unnecessary: equation (18) has a unique solution and characterises the factorial function. Although this fact is a consequence of the definition of initial algebra it is not an immediate consequence and therefore requires a proof. (We present a proof later.)

When we abstract from the particular functor and initial algebra in factorial program (18) a general recursion scheme is obtained.

$$(19) \qquad \mathsf{X} \;::\; \mathsf{X} = \mathsf{R} \cdot \mathsf{F}.(\mathrm{I}{\times}\mathsf{X}) \cdot \mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I}) \cdot \mathsf{in}^{\cup} \quad .$$

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

This generic formulation of primitive recursion was introduced (for functions) by Meertens [21]. He called such an equation a *para equation* and a solution to the equation a *paramorphism*. Among the results he established three are worthy of note. The first is that paramorphisms are universal in the sense that every function with source the carrier of an initial algebra can be expressed as a paramorphism. The second is that every catamorphism can be expressed as a paramorphism, and the third is that a para equation has a unique solution.

The first two of these results suggest that a useful discipline of recursive program construction could be built around paramorphisms. Indeed in intuitionistic type theory (for example Martin-Löf's theory of types [25]) this is what is done. There paramorphisms correspond to the eliminators for inductive types, and their computation rules are para equations. Unfortunately, as the name suggests, primitive recursion is still not general enough to capture all *useful* recursion schemes. No truly practical theory of datatypes has been built around primitive recursion, as all experienced functional programmers would testify.

5 Advanced recursion

The observation that not all programs are primitive recursive is of course not new. For instance in the well-known definition of the computable functions on the natural numbers the primitive recursive functions are a proper subclass of the class of all computable functions. Therefore, a number of researchers have proposed generalisations of the para equation. Malcolm [19] defined the *zygo* equation, which type of equation was generalised by Fokkinga [11] to the *mutu* equation

$$(20) \qquad X \ :: \ X = R \cdot F.X \vartriangle F.X \cdot \mathsf{in}^{\cup} \quad .$$

It can be shown that such an equation has a unique solution. (Fokkinga's proof requires that R is a function. The proof we give later allows R to be an arbitrary relation.)

The problem with this kind of generalisation of primitive recursion is that it is rather limited. The generalised programs are of the form

$$(21) \qquad X \ :: \ X = R \cdot G.X \cdot \mathsf{plumbing} \cdot \mathsf{in}^{\cup} \quad ,$$

where G is a functor, in an initial F-algebra for some functor F, and $\mathsf{plumbing}$ a natural transformation. Recall that a natural transformation is only a coding of "where the arguments have to go", so it does not represent a proper computation. Yet, many programs perform a substantial amount of preprocessing before the recursive call and are therefore not of the form (21).

Consider, for instance, the program known as "quicksort" , here abbreviated to qs.

$$(22) \qquad \mathsf{qs} \ = \ \mathsf{nil} \triangledown (\mathsf{join} \cdot I \times \mathsf{cons}) \ \cdot \ \mathbb{1} + (\mathsf{qs} \times (I \times \mathsf{qs})) \ \cdot \ \mathsf{nil}^{\cup} \blacktriangledown \mathsf{dnf}$$

To see that this is the quicksort program one has to interpret dnf as the well-known "Dutch national flag" relation. This relation 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 $\mathsf{join} \cdot I \times \mathsf{cons}$, where join produces the concatenation of two lists.

The recursion of the quicksort program differs from the recursion of equation (21) because relation dnf can not be considered as a simple plumbing relation: it is not merely a copying and rearranging of the input values in the sense that it does not disappear if formula (22) is expressed point-wise. In short, dnf represents a considerable amount of preprocessing.

Therefore, we might want to consider equations of the form

$$(23) \qquad X \ :: \ X = R \cdot G.X \cdot \mathsf{prepro} \cdot \mathsf{in}^{\cup}$$

as the standard recursion scheme. Fokkinga [11] has studied a limited class of equations of this form (the class was limited because there were a number of restrictions on the preprocessor relation prepro and functor G). He has shown that (in the case that all components are functions) such equations have a unique solution and called that unique solution a *prepromorphism*.

However, for our purposes (23) is still not general enough. We want for instance to capture divide and conquer algorithms and such programs do not, in general, have the form of equation (23) because no initial algebra is involved. A typical divide and conquer program is of the form

$$(24) \qquad X \ :: \ X = R\triangledown \mathsf{conquer} \cdot I{+}(X{\times}X) \cdot I{+}\mathsf{divide} \cdot A{\blacktriangledown}B \quad .$$

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 $\mathsf{conquer}$.

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 (24) 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:

$$(25) \qquad x \quad = \quad \mathsf{if}\ \neg \mathsf{b} \rightarrow \mathsf{skip}\ \ [\!]\ \ \mathsf{b} \rightarrow \mathsf{s;x}\ \mathsf{fi} \quad .$$

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 (25) using disjoint sum as:

$$(26) \qquad X \quad = \quad I\triangledown I \cdot I{+}X \cdot {\sim}B \blacktriangledown (S \cdot B) \quad .$$

The function to reverse a list given in section 3 is a specific case.

If we express the implementation of list reversion in a style more typical of functional programming rather than an imperative style then we see a further generalisation of the sort of equations we want to consider. Specifically, the functional programmer would write the definition of the reverse function as follows:

```
reverse xs = accumrev xs nil
where
% accumrev accumulates the reverse of the input list xs
% in the list ys
accumrev nil ys = ys
accumrev (x:xs) ys = accumrev xs (x:ys)
```

The form of this definition is structural recursion on the first argument, with an additional parameter as second argument, the additional parameter being used to accumulate the reverse of the list.

Rewriting this definition in a point-free style it takes the form

$$\mathsf{reverse} = \mathsf{exr} \cdot \mathsf{X} \quad \text{, where}$$

$$\mathsf{X} = \mathsf{id}\triangledown\mathsf{id} \cdot \mathsf{id}+\mathsf{X} \cdot \mathsf{plumbing} \cdot \mathsf{in}^{\cup} \times \mathsf{id} \quad .$$

Here the occurrences of id are all identity functions (the types of which are different but not relevant to the current discussion) and plumbing is a function of type

$$\mathsf{List.J}\times\mathsf{List.J} + \mathrm{I}\times\mathsf{List.J} \longleftarrow (\mathbb{1}+\mathrm{J}\times\mathrm{I}) \times \mathsf{List.J}$$

that is polymorphic in I and J. Reading from right to left, $\mathsf{in}^{\cup}$ maps the first argument into either an element of $\mathbb{1}$ (if it is an empty list) or its head element and its tail, tagging the result to indicate whether the input list was empty or not. Then, in the case of the empty list, plumbing reconstitutes the input lists. In the case of a non-empty list, plumbing maps the triple $((j,i),js)$ into the pair $(i,j{:}js)$. Note that in this particular application the type I will always be List.J. However, the fact that the type of plumbing is more general than this is important to the termination argument given in section 8.2.

6 A Programming Paradigm

6.1 Hylo programs

So far we have considered various restrictions on recursive program schemes and dismissed them all as too restrictive. Now it is time to define the class of programs we propose as the standard recursion scheme. The defining property of programs in the class is very simple and generalises all the defining equations of the "morphisms" we have encountered thus far.

Definition 27 (Hylos) Let R and S be relations and F a relator. An equation of the form

(28) $\quad \mathsf{X}{::} \quad \mathsf{X} = \mathsf{R} \cdot \mathsf{F.X} \cdot \mathsf{S}$

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

(The name "hylomorphism" for the least solution of a (functional) equation of this form was coined by Erik Meijer [11].)

Note that there are three components to a hylo program, the relator F and two relations R and S. On typing grounds, if X is to have type $A \leftarrow B$ then R must be 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.

Recall that one of the goals of this paper was to identify a recursion scheme that makes arbitrary recursion superfluous. The question is whether the class of hylo programs forms a good candidate for such a recursion scheme. The answer, we believe, is positive for the following reasons.

First, it is well known that everything that can be computed can be computed with a repetition. Now a repetition is a hylo program. (See (26).) Therefore, the class of hylo equations is large enough to write all programs we might want to write: every program can be transformed into a hylo program. In other words we can safely restrict the class of programs we want to consider to the class of hylo programs.

The class of programs can also be safely restricted to just repetitions. However, although that is *safe* it would not be *convenient* because there are many practical algorithms which can not be expressed as repetitions in a simple way, quicksort being the classic example. It would not be convenient because we are then forced to express such programs as repetitions, thereby making them unnecessarily complicated. On the other hand, the question of whether it is convenient to require that all algorithms are expressed in the form of hylo programs seems to have a positive answer. This is because the great majority of programs that one encounters in practice are in the form of a hylo equation. The few programs that are not are by no means standard, much used algorithms. On the contrary: they tend to be rather artificial. An example is, for instance, the program to compute the Ackerman function. Hylo programs thus combine convenience with practicality.

Finally, the hylo recursion scheme offers substantially greater freedom in designing programs because the solution strategy is a parameter of the scheme. The solution strategy is encapsulated in the relator, F. For instance relator $X \mapsto I+X$ encapsulates repetition, $X \mapsto I+X\times X$ encapsulates a divide and conquer strategy, and $X \mapsto F.(I\times X)$ encapsulates primitive recursion. A first step in the design of hylo programs is thus the choice of the relator.

Because of these considerations we conclude that a discipline of programming focused on the design of hylo programs is well worth pursuing, particularly if it extends the discipline of designing repetitions using invariants and bound functions and has the same spirit. Developing such a discipline is the topic of the remaining sections of this paper.

A discipline of programming should be underpinned by a mathematical theory that guides the engineer to good, clean programs and protects them from inadvertent error. In the next section we present an important theorem —the hylo theorem— that relates hylo programs to so-called "virtual" data structures. The theorem is important because it provides much insight into how to design the solution strategy. Later sections are concerned with how to design terminating hylo programs.

6.2 Intermediate data structures

In section 4.2 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 $(\!|R|\!) \cdot (\!|S^{\cup}|\!)^{\cup}$ 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

$(\!|R|\!) \cdot (\!|S^{\cup}|\!)^{\cup}$ is the least solution of the hylo equation (28).

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 $(\!|R|\!) \cdot (\!|S^{\cup}|\!)^{\cup}$ does make the process explicit. For this reason, the relator F is said to specify a *virtual* data structure [27].

Two simple examples of virtual data structures are provided by do-statements and the factorial function. In the case of do-statements (see (26)) the virtual datatype is the carrier set of an initial (I+)-algebra, a type which is isomorphic to $I \times \mathsf{Nat}$ —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 (18) can be rewritten so as to make the relator F explicit:

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

The "virtual" datatype is thus the type of lists of natural numbers, the carrier set of an initial $\mathbb{1}+(\mathsf{Nat}\times)$-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:

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

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:

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

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 $(\!|(\mathsf{concat3}\cdot a\times\mathsf{id}\times b)\triangledown c|\!)$ 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 $f\cdot f^{\cup}$ the language generated satisfies

$$S \quad = \quad (\!|(\mathsf{concat3}\cdot a\times\mathsf{id}\times b)\triangledown c|\!)\cdot(\!|(\mathsf{concat3}\cdot a\times\mathsf{id}\times b)\triangledown c|\!)^{\cup} \quad .$$

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

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

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. A context-free grammar defines a type of parse trees in a fairly obvious way. Also an 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 $\mathsf{unparse}\cdot\mathsf{unparse}^{\cup}$. 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.

Of course, 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.

To illustrate how this is done let us consider a slightly more complicated example than the one we have just treated. Consider the grammar:

$$\begin{aligned} S &::= ST \mid d \\ T &::= aSb \mid c \end{aligned}$$

where, again small letters denote some arbitrary sets of words over some fixed alphabet. Since the definition of the grammar involves mutual recursion, the class of parse trees for strings in the language generated by the grammar is defined by mutual recursion. Specifically, the two data structures, Stree and Ttree satisfy:

$$\begin{aligned} \mathsf{Stree} &= (\mathsf{Stree}\times\mathsf{Ttree})+d \\ \mathsf{Ttree} &= (a\times\mathsf{Stree}\times b)+c \quad . \end{aligned}$$

The *pair* of data structures forms an initial F-algebra where F maps the pair (X,Y) to the pair $((X\times Y)+d\,,\,(a\times X\times b)+c)$. The relator F is thus an endorelator on the product allegory Rel^2. Unparsing functions unparseS and unparseT on Stree s and Ttree s are defined in the obvious way by mutual recursion; formally, the *pair* $(\mathsf{unparseS},\mathsf{unparseT})$ is a catamorphism on the initial F-algebra. Finally, the *pair* of languages (S,T) is the left domain of this catamorphism. Specifically, the catamorphism takes the form

$$(\!\![(\mathsf{concat}\triangledown d\,,\,(\mathsf{concat3}\cdot a\times\mathsf{id}\times b)\triangledown c)]\!\!)$$

where concat concatenates a pair of strings and concat3 concatenates a triple of strings. So the *pair* (S,T) is the composition of a catamorphism after an anamorphism. Applying the hylo theorem we get the (single) hylo equation

$$(S,T) \;=\; (f,g)\;\cdot\;F.(S,T)\;\cdot\;(f,g)^{\cup}$$

in Rel^2 where, for brevity, we have used f to stand for $\mathsf{concat}\triangledown d$ and g for $\mathsf{concat3}\cdot a\times\mathsf{id}\times b$. Unravelling the definition of composition and converse in the product allegory one gets the *pair* of equations:

$$S \;=\; \mathsf{concat}\triangledown d\;\cdot\;(S\times T)+d\;\cdot\;\mathsf{concat}^{\cup}\blacktriangledown d$$

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

This *pair* of equations is thus a *single* hylo equation in the allegory Rel^2.

We can now sketch how the hylo theorem relates to context-free language recognition in general. A context-free grammar with n nonterminal symbols defines an endorelator F on Rel^n. The definition is straightforward: choice between righthand sides is mapped to disjoint sum and concatenation to cartesian product. Also terminal symbols are mapped to constant endorelators (on Rel). An initial F-algebra is a vector of types indexed by non-terminals; the element indexed by nonterminal A is the type of parse trees of words generated by A. There are now two ways we can interpret the hylo theorem as applied to this situation.

The first interpretation involves observing that the grammar also defines an $\mathsf{unparse}$ operation. This too is a vector indexed by nonterminals; each element is a function that unparses the corresponding parse trees. The $\mathsf{unparse}$ vector is an F-catamorphism and its left domain is the vector of languages generated by the nonterminals of the grammar. Since the left domain of a function f is $f\cdot f^{\cup}$ the left domain of $\mathsf{unparse}$ is the composition of a catamorphism and the converse of a catamorphism. The hylo theorem states that this is the least solution of a hylo equation. That is, the hylo theorem confirms that the vector of languages generated by the nonterminals of a context-free grammar is the least solution of a fixed point equation.

The second interpretation allows us to derive the unparsing operation. We observe that the vector of languages generated by the nonterminals of a context-free grammar is the least solution of a fixed point equation. Using the properties of disjoint sum and cartesian product, this fixed point equation can be rewritten as a hylo equation. Now, applying the hylo theorem, the vector of languages is equal to the composition of a catamorphism after the converse of a catamorphism. The former is the unparse vector for the grammar and the latter is the parse vector (albeit not in a form that can be directly implemented).

6.3 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 [4][1].

[1]Actually [4] 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

Recall that we defined the notion of an initial algebra in the context of a category. (See (13).) 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 29 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 29 Assume that F is an endorelator. Then $(\mathrm{I}, \mathsf{in})$ is a *relational initial* F-algebra iff $\mathsf{in} :: \mathrm{I} \leftarrow \mathsf{F}.\mathrm{I}$ is an F-algebra and there is a mapping $(\![\_]\!)$ defined on all F-algebras such that

(30) $\quad (\![R]\!) :: A \leftarrow \mathrm{I} \quad \text{if } R :: A \leftarrow \mathsf{F}.A \quad ,$

(31) $\quad (\![\mathsf{in}]\!) = \mathsf{id}_{\mathrm{I}} \quad$, and

(32) $\quad (\![R]\!) \cdot (\![S]\!)^{\cup} = \mu(X \mapsto R \cdot \mathsf{F}.X \cdot S^{\cup}) \quad .$

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

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

Theorem 33 (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 $\mathcal{M}ap(\mathcal{A})$. Then in is an initial F'-algebra if and only it is a relational initial F-algebra.

□

Note that the hylo theorem states an equivalence between two definitions. Considering first the implication (loosely speaking, an initial F-algebra is a relational initial F-algebra), property (32) is the property that we have been referring to as the "hylo theorem" above. Property (30) is a necessary prerequisite; essentially it states that catamorphisms are well-defined on relations given that they are well-defined on functions. Property (31) is the key to proving Lambek's lemma that an initial F-algebra is an isomorphism between its source and its target. A consequence of the opposite implication (a relational initial F-algebra is an initial F-algebra) is that catamorphisms on functions are the unique solutions of their defining equations. We shall see a generalisation of this result in section 9.1.

direction that is difficult. That is one reason why [4] chose to present the theorem in terms of coalgebras rather than algebras. The extra generality offered by the theorem in [4] encompasses the relational 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.

6.4 A programming paradigm

Hylo programs, combined with relation algebra form the basis for a paradigm for program design that combines the paradigms of sequential and functional programming.

In general, the design of a program starts with a specification (relation) S, often expressed using predicates. Relational, i.e. point-free, formulae can be far more compact than predicates, so S is rewritten in relational form. Such a rewrite has the additional advantage that it helps to structure the specification. In this process, however, one should not be holier than the pope by insisting that all points be removed. One can, for instance, avoid making the plumbing of rearrangements of arguments and other natural transformations explicit by introducing a name for the plumbing relation and just defining it pointwise.

Once specification S has been moulded into relational form, program design starts. We have to determine a relator F and relations R and T such that:

$$(34) \qquad R \cdot F.S \cdot T \subseteq S \quad .$$

The choice of F corresponds to the choice of a solution strategy like structural recursion, repetition, divide-and-conquer, etc. Alternatively, it may be easier to design a suitable intermediate data structure. In the case of more complex problems it may also be necessary to generalise S to a collection of specifications, the implementations of which are designed together resulting in a mutually recursive program.

The design of R and T reflects two phases in the implementation, the catamorphism and the anamorphism phases, respectively. The design of the anamorphism part involves the construction of a "type invariant" whilst the construction of the catamorphism is a recursive problem decomposition, making use of the type invariant.

A good example is the design of merge sort. In the first phase a tree structure (with the values in the leaves) is formed with the invariant property that, first, the subtrees are balanced and, second, the elements of the subtrees are the elements of the input list. In the second phase the tree is broken down into a sorted list whereby two lists are joined by the standard "merging" process. Of course, in the usual implementations of merge sort the construction of the tree is "virtual", i.e. not made explicit. (However, the tree is implicitly present in the stack used to implement the recursion.)

Quicksort is also an example: the first phase constructs a binary search tree. That is, a binary tree is constructed with the invariant property that the value in the root is at least as big as the maximum value in the left subtree and smaller than the values in the right subtree. Another invariant is that the values in the tree are the values in the input list. It is then easy to construct the program for the second phase that flattens this tree into a sorted list. For another example in which the intermediate data structure is a heap we refer the reader to [27].

There is one vital element that is missing in this account: the notion of *making progress*, that is, constructing a terminating program. Making progress towards the termination condition is a vital element in the design of sequential programs, equal in importance to the invariant property. The body of a loop is designed so as decrease the value of a bound function whilst maintaining the invariant. Making progress is also vital in designing recursive functions in a functional programming language. In the remainder of this paper we develop a theory of program termination relative to a design strategy. Formally, we develop a calculus of "F-reductivity", where relator F represents the design strategy, and "reductivity" relates to reducing the problem size.

7 Reductivity and termination

In this section we introduce the primary novel contribution of this paper. The section is devoted to justifying a formal, calculational definition of termination of hylo programs. In order to give such a justification without resorting to a long, possibly complicated, formal operational semantics of hylo programs we approach the issue from two different angles. To begin with we develop the formal definition of "reductivity" from an informal operational understanding of the execution of hylo programs. This is done in subsection 7.1. Then, in subsection 7.2 we show how the definition corresponds to an intuitive notion of reducing problem size. Section 9, which makes use of results in section 8, provides further justification for the relevance of the proposed "reductivity" definition by relating it to well-foundedness and structural induction.

7.1 Safe sets

In order to derive a definition of termination, we must first have an idea of how a hylo program,

$$X = S \cdot F.X \cdot R$$

is to be executed. As long as no non-determinism is involved there seems to be only one reasonable choice: the program is executed by first unfolding the equation and then computing the argument for the recursive call by executing R. This procedure is repeated until a base case is reached and no further unfoldings are necessary. Then the output is computed by executing S as often as the equation was unfolded. If no base case is reached then the execution does not terminate and no output is produced. Somewhat more precisely: for each input which gives rise to a terminating computation there exists a natural number n such that the output value is related to the input value by relation

(35) $$S \cdot F.S \cdot \cdots \cdot F^n.S \cdot F^{n+1}.\perp\!\!\perp \cdot F^n.R \cdot \cdots \cdot F.R \cdot R$$

(where the relation $F.\perp\!\!\!\perp$ is the input-output relation for the base case). This is the familiar execution scheme applied by the implementations of imperative languages like Pascal and functional languages such as Haskell. (Because of this execution scheme, the computed input-output relation is the least solution of the hylo program. Relators that arise in practice are continuous, so the union over all n of the relations (35) is the least solution of the program.)

If R is deterministic there is only one possible argument for each recursive call (and therefore only one value for n). However if R is non-deterministic then each unfolding can give rise to many possibilities for continuing the computation. The question is now which of those possible continuations is chosen. The standard semantics of the guarded command language is defined in such a way that one of the possibilities can be chosen non-deterministically. However, there is another scenario imaginable: all possibilities could be pursued, maybe in parallel, until a base case is reached.

An ideal implementation of Prolog should do the latter. There is then the problem that R can be unboundedly nondeterministic: there can be an infinite number of possible continuations. In that case (assuming that no machines exist with unbounded parallellism) there can be two causes of non-termination: either there is never a base case reached or at some point infinitely many possible continuations have to be followed. However, if R is boundedly non-deterministic this execution scheme will more often lead to a terminating computation than the execution scheme of the guarded command language.

For a given input, exploring all possible continuations in parallel will result in a terminating computation if one of the possible continuations leads to a base case. On the other hand, in the case that one continuation is chosen non-deterministically, termination can only be guaranteed if all continuations lead to a base case.

Although the execution scheme of the guarded command language is perhaps less attractive because it terminates less often we choose that as the way the hylo program is run. The reason is that choosing a continuation non-deterministically, rather than pursuing all possible continations, is compatible with the standard implementations of functional and imperative languages.

Given program

$$(36) \qquad X = S \cdot F.X \cdot R \qquad ,$$

a subset of the carrier set I of coalgebra R is called “safe” if the program is guaranteed to terminate when started in a state from the subset. The union of all safe sets is called the maximal safe set and is modelled by coreflexive B. So, maximal safe set B is to be interpreted as the set of all states in which the program can be started without the risk of non-termination. Note that, with this interpretation of safe, a set is safe if and only if it is contained in B.

The assumption is now that the maximal safe set exists and our goal is to characterise this set.

A first requirement on a safe set A (i.e. a set with property $A \subseteq B$) is that, whenever program X is started in it, all possible recursive calls terminate: we can not be sure which of them is chosen. So, if the program is started in set A, the recursive call should be started in the maximal safe set B. This can be expressed as $(R \cdot A)_< \subseteq F.B$. In words, the left domain of $R \cdot A$ is an F-structure and each element in such an F-structure should satisfy property B. The requirement on A can thus be formalised as:

$$(37) \quad A \subseteq B \quad \Rightarrow \quad (R \cdot A)_< \subseteq F.B \quad .$$

Conversely, if property $(R \cdot A)_< \subseteq F.B$ holds (i.e. the recursive calls are started in the safe set B whenever the program is started in set A) then set A can be considered safe and should therefore be contained in the maximal safe set:

$$(38) \quad A \subseteq B \quad \Leftarrow \quad (R \cdot A)_< \subseteq F.B \quad .$$

Now we can make use of the defining Galois connection of the coreflexive factor: expression $(R \cdot A)_< \subseteq F.B$ can be rewritten as $A \subseteq R \searrow F.B$. Therefore (37) and (38) can be combined giving:

$$(39) \quad \forall(A :: \; A \subseteq B \;\equiv\; A \subseteq R \searrow F.B)$$

The rule of indirect equality gives that (39) is equivalent to $B = R \searrow F.B$. This leaves us with the choice which solution of this equation in B to take. Basically there are only two obvious choices: the least and the greatest solution. In general, however, the greatest solution is equal to the identity relation. So if we take that one as a safe set we adopt the view that the computation can be started safely in any state, regardless of what R is. This would obviously be too optimistic. Therefore we take the least solution as the maximal safe set. (The pessimistic view is a “demonic” view as opposed to an “angelic” view of program termination.)

Definition 40 (Safe set) The *safe set* of program $X = S \cdot F.X \cdot R$ is the coreflexive $\mu(A \mapsto R \searrow F.A)$.
□

Now the execution of program $X = S \cdot F.X \cdot R$ terminates if the program is started in its safe set. Admittedly, this definition is a bit informal: we did not define what it means for a program (an equation) to be started in a coreflexive. This can be made precise, but we will not do so here because here we are interested only in programs which terminate everywhere. Such a program will be called a terminating program and the notion “terminating program” can be defined (precisely) in terms of reductivity.

If $\mu(A \mapsto R\searrow F.A)$ equals the carrier of R, the program can be started safely in any state. Conversely, if $\mu(A \mapsto R\searrow F.A)$ does not equal the carrier of R, there is a non-safe state because $\mu(A \mapsto R\searrow F.A)$ is the maximal safe set. This gives the connection between reductivity and termination.

Definition 41 (F-reductivity) Relation $R :: F.I \leftarrow I$ is said to be F-*reductive* if and only if it enjoys the property:

(42) $\quad \mu(R\searrow\circ F) = \mathsf{id}_I \quad .$

□

Definition 43 (Terminating program) Program $X = S \cdot F.X \cdot R$ is a terminating program if and only if the relation R is F-reductive.
□

Note that reductivity states that the equation $B = R\searrow F.B$ has only one solution. So if we restrict our attention to terminating programs in the sense of definition 43 the choice which solution of this equation should be taken as the maximal safe set becomes irrelevant.

7.2 Reducing problem size

Having defined termination, we have to check that the definition given here is compatible with the programmer's definition.

A programmer proves termination by using well-founded relations: he has to prove that the argument of every recursive call is "smaller" than the original argument. For hylo program $X = S \cdot F.X \cdot R$ this means that all values stored in an output F-structure of R have to be smaller than the corresponding input of R. More formally, with $x\langle\mathsf{mem}\rangle y$ standing for "x is a member of F-structure y" (or, x is a value stored in F-structure y"), we need for all x and z

$$\forall(y :: \ x\langle\mathsf{mem}\rangle y \wedge y\langle R\rangle z \ \Rightarrow \ x \prec z) \qquad ,$$

for some well-founded ordering $\prec$. That is, a relation R is F-reductive if and only if there is a well-founded relation $\prec$ such that whenever an F-structure is related by R to some y, it is the case that every value stored in the F-structure is related to y by $\prec$.

To make this statement precise we need to formalise the concept of "values stored in an F-structure". Hoogendijk and De Moor [16, 15] have shown that this is possible for so-called "container types". For the relators from this class one can define a membership relation, say mem. For example, for the list relator this relation holds between a point of the universe 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).

A precise characterisation of the membership relation of a relator is the following (see [16, 15]):

Definition 44 (Membership) Relation $\mathsf{mem} :: I \leftarrow F.I$ is a membership relation of relator F if and only if it satisfies, for all coreflexives A, $A \subseteq I$:

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

□

When this definition is expressed pointwise it reads:

$$x \in F.A \;\equiv\; \forall(i{:}\; i\langle\mathsf{mem}\rangle x{:}\; i \in A) \quad .$$

Informally: an F-structure satisfies the property $F.A$ iff all the values stored in the structure satisfy property A.

Using this definition of membership we get a precise relationship between reductivity and well-foundedness. Indeed, for coalgebra R with carrier I and coreflexive A below I, we have:

$$\begin{array}{ll}
 & R \searrow F.A \\
= & \{ \quad \text{definition 44} \quad \} \\
 & R \searrow (\mathsf{mem} \searrow A) \\
= & \{ \quad \text{factors (1)} \quad \} \\
 & (\mathsf{mem} \cdot R) \searrow A \quad .
\end{array}$$

Now, well-foundedness of $S :: I \leftarrow I$ is the condition that the least prefix point of the function $A \mapsto S \searrow A$ is I [8] whereas reductivity of $R :: F.I \leftarrow I$ is the condition the least prefix point of the function $A \mapsto R \searrow F.A$ is I. So, for coalgebra $R :: F.I \leftarrow I$, the statement that R is F-reductive is equivalent to the statement that $\mathsf{mem} \cdot R$ is well-founded. Formally,

$$R \text{ is F-reductive} \;\equiv\; \mathsf{mem} \cdot R \text{ is well-founded} \quad .$$

Conversely,

$$S \text{ is well-founded} \;\equiv\; \mathsf{mem} \backslash S \text{ is F-reductive} \quad .$$

This follows because, by the argument above with R instantiated to $\mathsf{mem} \backslash S$,

$$(\mathsf{mem} \backslash S) \searrow F.A \;=\; (\mathsf{mem} \cdot \mathsf{mem} \backslash S) \searrow A \quad .$$

But $\mathsf{mem} \cdot \mathsf{mem}\backslash S \subseteq S$. So, by the anti-monotonicity of $(\searrow A)$,

$$(\mathsf{mem}\backslash S) \searrow F.A \;\supseteq\; S \searrow A \quad .$$

Hence, the least prefix point of $(\,((\mathsf{mem}\backslash S)\searrow) \circ F\,)$ is at least the least prefix point of $(\,S\searrow\,)$. Summarising, we have:

Theorem 45 Suppose mem is the membership relation for relator F. Then the functions $R \mapsto \mathsf{mem} \cdot R$ and $S \mapsto \mathsf{mem}\backslash S$ form a Galois connection between the F-reductive relations, R, and the well-founded relations, S.

□

To illustrate this theorem we recall that the relator corresponding to iteration over state space I is $(\,I+\,)$. Termination of a loop with body S should therefore be expressible in terms of $(\,I+\,)$-reductivity. Indeed:

Theorem 46 For all R and S,

$$R{\blacktriangledown}S \text{ is } (I+)\text{-reductive} \;\equiv\; S \text{ is well-founded} \quad .$$

Proof The membership relation for $(\,I+\,)$ is $\mathsf{inr}^{\cup}$ and $\mathsf{inr}^{\cup} \cdot R{\blacktriangledown}S = S$ and $\mathsf{inr}^{\cup} \backslash S = \top\!\!\top{\blacktriangledown}S$. So, by theorem 45, if $R{\blacktriangledown}S$ is $(\,I+\,)$-reductive then S is well-founded and if S is well-founded $\top\!\!\top{\blacktriangledown}S$ is $(\,I+\,)$-reductive. But $\top\!\!\top{\blacktriangledown}S \supseteq R{\blacktriangledown}S$ for all R. Thus $R{\blacktriangledown}S$ is $(\,I+\,)$-reductive for all R. (The theorem can be proved without recourse to theorem 46 if desired.)
□

Bird and De Moor [6, chapter 6] avoid the introduction of the notion of reductivity by always requiring that $\mathsf{mem} \cdot R$ is well-founded whenever F-reductivity of R is required. This only works if F has a membership relation. The class of such relators is large. It contains for instance all the polynomial relators, the relators we use in our everyday work. So, although there exist relators that do not have a membership relation it is likely that these are interesting for theoretical reasons only. The main advantage of defining termination in terms of reductivity instead of well-foundedness and membership is that it is possible to formulate theorems relating reductivity of one type to reductivity of another type. Several of the rules presented in section 8 are of this nature.

For a profound discussion of membership the reader is referred to Hoogendijk's thesis [15].

8 A calculus of reductive relations

In the previous section we argued that the notion of F-reductivity captures precisely the termination of hylo programs. In this section we give a number of rules that allow us to prove that a relation is reductive. These rules form the basis of a calculus of reductive relations. In each case we motivate the rule by showing how it is used to verify the termination of a known program or class of programs. However, the major design criterion for the calculus is not program *verification* but that it is useful for the *construction* of terminating programs.

8.1 Basic F-reductive relations

In this section it is shown that for any relator F there exist F-reductive relations. We begin with perhaps the most commonly used theorem.

Theorem 47 The converse of an initial F-algebra is F-reductive.

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

$$\mathrm{I} \subseteq A \quad \Leftarrow \quad \mathsf{in}^{\cup} \searrow \mathsf{F}.A \subseteq A \quad .$$

We start with the antecedent and derive the consequent:

$$
\begin{array}{ll}
 & \mathsf{in}^{\cup} \searrow \mathsf{F}.A \subseteq A \\
\equiv & \quad \{ \quad \text{for function } \mathsf{f} \text{ and coreflexive } \mathsf{B},\ \mathsf{f} \searrow \mathsf{B} = \mathsf{f}^{\cup} \cdot \mathsf{B} \cdot \mathsf{f}, \\
 & \qquad\quad \mathsf{in}^{\cup} \text{ is a function and } \mathsf{F}.A \text{ is a coreflexive} \quad \} \\
 & \mathsf{in} \cdot \mathsf{F}.A \cdot \mathsf{in}^{\cup} \subseteq A \\
\Rightarrow & \quad \{ \quad \text{Hylo theorem: (33) and (29)}, \\
 & \qquad\quad \mathsf{in} \text{ is an initial } \mathsf{F}\text{-algebra} \quad \} \\
 & (\!|\mathsf{in}|\!) \subseteq A \\
\equiv & \quad \{ \quad \text{identity rule: (31), } \mathsf{in} :: \mathrm{I} \leftarrow \mathsf{F.I} \text{ is an initial } \mathsf{F}\text{-algebra} \quad \} \\
 & \mathrm{I} \subseteq A \quad .
\end{array}
$$

□

An immediate corollary of theorem 47 is that the cata program

$$\mathsf{X} \ :: \ \mathsf{X} = \mathsf{R} \cdot \mathsf{F.X} \cdot \mathsf{in}^{\cup}$$

is terminating according to definition 43. Also, by theorem 63 which we prove later, the solution of the equation is unique for all R and not just the maps in the allegory.

Our next theorem is motivated by a desire to show that selection sort is a terminating program. The program is:

$$(48) \qquad \mathsf{slsrt} \ = \ \mathsf{nil} \triangledown \mathsf{cons} \cdot \mathbb{1} + \mathrm{I} \times \mathsf{slsrt} \cdot \mathsf{nil}^{\cup} \blacktriangledown (\mathsf{cons}^{\cup} \cdot \mathsf{select}) \quad .$$

Relation select holds between two lists if the output list has the property that it can be obtained from the input list by swapping the first element and the minimum of the list. Because there is no plumbing relation involved the program is easy to interpret: it relates the empty list to the empty list. A non-empty list is sorted by swapping the first element and the minimum of the input list (select), then the list is taken apart into the head and the tail ($\mathsf{cons}^{\cup}$), the tail is sorted recursively ($I{\times}\mathsf{slsrt}$), finally the head, i.e. the minimum of the input, is added to the result of the recursive call (cons).

The termination proof of selection sort depends on the observation that select is a relation between lists of equal length. The largest relation between lists of equal length is $\mathsf{List}.\top\!\!\top$: this relation holds between lists of equal length such that the elements of the input and output list are related by the total relation, which means that the only thing we can say about the input and output is that they are of equal length. In fact, the relation $\mathsf{List}.\top\!\!\top$ can be used to formalise the notion "equal length": relation R is a relation between lists of equal length iff R is contained in $\mathsf{List}.\top\!\!\top$.

The desired theorem is generic in inductively defined types like List . The basis for the theorem is that if $\oplus$ is a binary relator and, for each I , $\mathsf{in}_I \;::\; T.I \leftarrow I \oplus T.I$ is an initial ($I\oplus$)-algebra, then the function mapping $R :: A{\leftarrow}B$ to the catamorphism $(\![(\mathsf{id}_B\oplus)\,;\; \mathsf{in}_I \cdot R \oplus \mathsf{id}_{T.A}]\!)$ extends the mapping T on objects to a relator, often called a *tree relator* [5, 6].

Theorem 49 Let $\oplus$ be a binary relator, in_I an initial ($I\oplus$)-algebra, and T the tree relator corresponding to $\oplus$ and in_I . Then $\mathsf{in}_I{}^{\cup} \cdot T.\top\!\!\top_{I\leftarrow I}$ is ($I\oplus$)-reductive.

Proof For brevity we omit the subscripts on in and $\top\!\!\top$ (except where the information is relevant), and we let B denote $\mu(A \mapsto (\mathsf{in}^{\cup} \cdot T.\top\!\!\top) \searrow (I\oplus A))$. Then

$$
\begin{array}{ll}
 & \mathsf{in}^{\cup} \cdot T.\top\!\!\top \quad \text{is } (I\oplus)\text{-reductive} \\
\equiv & \qquad \{ \qquad \mathsf{in}^{\cup} \cdot T.\top\!\!\top \;::\; I \oplus T.I \leftarrow T.I\text{, definition 41} \quad \} \\
 & \mathsf{id}_{T.I} \subseteq B \\
\Leftarrow & \qquad \{ \qquad \mathsf{in}^{\cup} \;::\; I\oplus T.I \leftarrow T.I \text{ is } (I\oplus)\text{-reductive.} \quad \} \\
 & \mathsf{in}^{\cup}\searrow(I\oplus B) \subseteq B \\
\Leftarrow & \qquad \{ \qquad \text{by the rolling rule of fixed point calculus [24]} \\
 & \qquad\qquad\quad \mathsf{in}^{\cup}\searrow(I\oplus B) = \mu(A \mapsto \mathsf{in}^{\cup}\searrow(I\oplus(T.\top\!\!\top\searrow A))) \quad \} \\
 & \mu(A \mapsto \mathsf{in}^{\cup}\searrow(I\oplus(T.\top\!\!\top\searrow A))) \;\subseteq\; \mu(A \mapsto (\mathsf{in}^{\cup} \cdot T.\top\!\!\top)\searrow(I\oplus A)) \\
\Leftarrow & \qquad \{ \qquad \text{for all } A\text{, } \mathsf{in}^{\cup}\searrow(I\oplus(T.\top\!\!\top \searrow A)) \subseteq (\mathsf{in}^{\cup} \cdot T.\top\!\!\top)\searrow(I\oplus A) \\
 & \qquad\qquad\quad \text{(for proof, see below) monotonicity of } \mu \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

The proof is completed by establishing the inclusion contained in the last hint. This we do as follows.

$$
\begin{array}{ll}
 & (\mathsf{in}^\cup \cdot \mathsf{T}.\top\!\top) \backslash\!\!\!\_\, (\mathrm{I} \oplus A) \\
= & \{ \quad \mathsf{T}.\top\!\top \;=\; (\![(\mathsf{id}_\mathrm{I} \oplus)\,;\, \mathsf{in} \cdot \top\!\top \oplus \mathsf{id}_{\mathsf{T}.\mathrm{I}}]\!) \text{ and } \top\!\top_{\mathrm{I}\leftarrow\mathrm{I}} = (\top\!\top_{\mathrm{I}\leftarrow\mathrm{I}})^\cup \;. \\
 & \quad\;\; \text{Thus } \quad \mathsf{in}^\cup \cdot \mathsf{T}.\top\!\top \;=\; \top\!\top \oplus \mathsf{T}.\top\!\top \cdot \mathsf{in}^\cup \quad \} \\
 & (\top\!\top \oplus \mathsf{T}.\top\!\top \cdot \mathsf{in}^\cup) \backslash\!\!\!\_\, (\mathrm{I} \oplus A) \\
= & \{ \quad \text{factors: (1)} \quad \} \\
 & \mathsf{in}^\cup \backslash\!\!\!\_\, ((\top\!\top \oplus \mathsf{T}.\top\!\top) \backslash\!\!\!\_\, (\mathrm{I} \oplus A)) \\
\supseteq & \{ \quad \text{relators distribute over coreflexive factors;} \\
 & \quad\;\; \text{this also holds for binary relators} \quad \} \\
 & \mathsf{in}^\cup \backslash\!\!\!\_\, ((\top\!\top \backslash\!\!\!\_\, \mathrm{I}) \oplus (\mathsf{T}.\top\!\top \backslash\!\!\!\_\, A)) \\
= & \{ \quad \mathsf{R} \backslash\!\!\!\_\, \mathrm{I} = \mathrm{I} \quad \} \\
 & \mathsf{in}^\cup \backslash\!\!\!\_\, (\mathrm{I} \oplus (\mathsf{T}.\top\!\top \backslash\!\!\!\_\, A)) \;\;.
\end{array}
$$

□

The following theorem is not deep, nevertheless it is extremely useful. Recall that the refinement order of programs is the same as inclusion of relations. The content of theorem 50 is therefore that reductivity is preserved under refinement.

Theorem 50 If R is F-reductive and $\mathsf{S} \subseteq \mathsf{R}$ then S is F-reductive.

Proof Immediate from the definition of F-reductivity and the monotonicity properties of the coreflexive factor, relators and the operator μ.
□

Now we can return to the proof of termination of selection sort. We have:

$$
\begin{array}{ll}
 & \mathsf{nil}^\cup \blacktriangledown (\mathsf{cons}^\cup \cdot \mathsf{select}) \\
\subseteq & \{ \quad \mathsf{select} \text{ is a relation between lists of equal length} \quad \} \\
 & \mathsf{nil}^\cup \blacktriangledown (\mathsf{cons}^\cup \cdot \mathsf{List}.\top\!\top) \\
= & \{ \quad \mathsf{List.R} \cdot \mathsf{nil} = \mathsf{nil} \text{, i.e. } \mathsf{List.R} \text{ maps the empty list to the empty} \\
 & \quad\;\; \text{list; this is used in the “converse” form} \\
 & \quad\;\; \mathsf{nil}^\cup \cdot (\mathsf{List}.\top\!\top)^\cup = \mathsf{nil}^\cup ; \\
 & \quad\;\; \text{relators commute with converse; } \top\!\top^\cup = \top\!\top \quad \}
\end{array}
$$

$$(\mathsf{nil}^\cup \cdot \mathsf{List}.\top\!\!\top)\blacktriangledown(\mathsf{cons}^\cup \cdot \mathsf{List}.\top\!\!\top)$$

$= \quad \{ \quad \blacktriangledown\text{-}\cdot\text{-fusion} \quad \}$

$$\mathsf{nil}^\cup \blacktriangledown \mathsf{cons}^\cup \cdot \mathsf{List}.\top\!\!\top$$

By theorem 50, relation $\mathsf{nil}^\cup \blacktriangledown (\mathsf{cons}^\cup \cdot \mathsf{select})$ is $(\mathbb{1}{+}I\times)$-reductive if $\mathsf{nil}^\cup \blacktriangledown \mathsf{cons}^\cup \cdot \mathsf{List}.\top\!\!\top$ is, which is a consequence of theorem 49 obtained by taking List for map, $\mathbb{1}{+}(R{\times}S) = R{\oplus}S$, and $\mathsf{nil} \triangledown \mathsf{cons} = \mathsf{in}$.

8.2 New F-reductive relations from old

This section is intended to show how, given an F-reductive relation, other reductive relations can be constructed.

An important lemma in fixed point calculus is the so-called *square rule*. The rule says that if in is an initial F-algebra then $\mathsf{in} \cdot F.\mathsf{in}$ is an initial F^2-algebra.

A concrete instance of this theorem in action is the definition of integer division by two: 0 and 1 divided by two are both 0, and $n{+}2$ divided by two is equal to n divided by two plus one. This defines division by two on a $(\mathbb{1}{+}\mathbb{1}{+})$-algebra, rather than on a $(\mathbb{1}{+})$-algebra which is the usual case when defining functions by primitive recursion on the natural numbers.

The theoretical importance of the square rule is as a lemma in the proof that the cartesian product of two algebraically complete categories is also algebraically complete [12]. The square rule can clearly be extended to an nth power rule. The corresponding reductivity lemma is the following:

Lemma 51 (Power Rule) Suppose $R :: F.I \leftarrow I$ is F-reductive. Define the function f on positive numbers by $f.1 = R$, $f.(n{+}1) = F.(f.n) \cdot R$. Then $f.n$ is F^n-reductive.

Proof We first prove by induction on n, $n \geq 1$, that

$$(R\setminus\circ F)^n.A \subseteq f.n \setminus F^n.A \quad .$$

The basis, $n = 1$, is trivial. For the induction step we have:

$$(R\setminus\circ F)^{n+1}.A$$

$= \quad \{ \quad \text{definition of } g^{n+1} \quad \}$

$$R\setminus F.((R\setminus\circ F)^n.A)$$

$\subseteq \quad \{ \quad \text{induction hypothesis, monotonicity of } R\setminus \text{ and } F \quad \}$

$$R\setminus F.(f.n \setminus F^n.A)$$

$\subseteq \quad \{ \quad \text{factors: (1), } F \text{ distributes through composition} \quad \}$

$$
\begin{array}{ll}
 & R\!\searrow\!(F.(f.n)\searrow F.(F^n.A)) \\
= & \{ \quad \text{factors: (1)} \quad \} \\
 & (F.(f.n)\cdot R)\searrow F.(F^n.A) \\
= & \{ \quad \text{definition} \quad \} \\
 & f.(n+1)\searrow F^{n+1}.A \quad .
\end{array}
$$

The proof of the lemma is now straightforward. We have:

$$
\begin{array}{ll}
 & f.n \text{ is } F^n\text{-reductive} \\
\equiv & \{ \quad \text{definition} \quad \} \\
 & \mu(((f.n)\searrow)\circ F^n) \supseteq I \\
\Leftarrow & \{ \quad \text{above, monotonicity of the fixed point operator} \quad \} \\
 & \mu((R\searrow\circ F)^n) \supseteq I \\
\Leftarrow & \{ \quad \mu(g^n)\supseteq \mu g \quad \} \\
 & \mu(R\searrow\circ F) \supseteq I \\
\equiv & \{ \quad \text{definition} \quad \} \\
 & R \text{ is } F\text{-reductive} \quad .
\end{array}
$$

□

The next two theorems can be used to change the "kind of reductivity", i.e. to construct F-reductive relations from G-reductive relations. These theorems formalise the idea that composing a reductive relation with a relation which "has no real effect" on a data structure results in a reductive relation. Of course, the phrase "has no real effect" is a bit vague, but the intended meaning is that such a relation transforms G-structures into F-structures without affecting the contents of the structures. That is, the only thing that can happen is that elements are copied or discarded. In order to state the theorem precisely we need to formalise what we have been loosely describing as "plumbing".

Definition 52 Relation R is a *plumbing* to relator F from relator G, written $R :: F \dot{\leftsquigarrow} G$, iff $R :: F.I \leftarrow G.I$, for some I, and for all coreflexives A such that $A \subseteq I$:

$$G.A \;\subseteq\; R\searrow F.A \quad .$$

□

Natural transformations are families of plumbing relations:

Lemma 53 Suppose $\alpha :: F \hookleftarrow G$ is a natural transformation. Then, for each I, α_I is a plumbing to F from G.

Proof Suppose A is a coreflexive below I. Then

$$
\begin{array}{ll}
& \mathsf{G}.A \subseteq \alpha_{\mathrm{I}} \backslash \mathsf{F}.A \\
\equiv & \quad \{ \quad \text{factors: (1)} \quad \} \\
& (\alpha_{\mathrm{I}} \cdot \mathsf{G}.A)_{<} \subseteq \mathsf{F}.A \\
\equiv & \quad \{ \quad \text{domains} \quad \} \\
& \mathsf{F}.A \cdot \alpha_{\mathrm{I}} \cdot \mathsf{G}.A = \alpha_{\mathrm{I}} \cdot \mathsf{G}.A \\
\equiv & \quad \{ \quad \alpha :: \mathsf{F} \hookleftarrow \mathsf{G}\,.\ \text{Thus, } \mathsf{F}.A \cdot \alpha_{\mathrm{I}} \supseteq \alpha_{\mathrm{I}} \cdot \mathsf{G}.A\,. \\
& \qquad\quad \mathsf{G}.A \cdot \mathsf{G}.A = \mathsf{G}.A \quad \} \\
& \mathsf{F}.A \cdot \alpha_{\mathrm{I}} \cdot \mathsf{G}.A \subseteq \alpha_{\mathrm{I}} \cdot \mathsf{G}.A \\
\equiv & \quad \{ \quad \mathsf{F}.A \subseteq \mathsf{id}_{\mathsf{F}.\mathrm{I}} \quad \} \\
& \mathsf{true} \quad .
\end{array}
$$

□

We can now formulate our theorem.

Theorem 54 Let Q be G-reductive and $\mathsf{S} :: \mathsf{F} \overset{\cdot}{\leftsquigarrow} \mathsf{Id}$, where Id denotes the identity relator. Then $\mathsf{F}.\mathsf{Q} \cdot \mathsf{S}$ is ($\mathsf{F} \circ \mathsf{G}$)-reductive.

Proof We prove the stronger:

$$\mu(A \mapsto \mathsf{Q} \backslash \mathsf{G}.A) \subseteq \mu(A \mapsto (\mathsf{F}.\mathsf{Q} \cdot \mathsf{S}) \backslash \mathsf{F}.(\mathsf{G}.A)) \quad .$$

This follows, by monotonicity of the fixpoint operator μ, from the fact that, for all A,

$$
\begin{array}{ll}
& (\mathsf{F}.\mathsf{Q} \cdot \mathsf{S}) \backslash \mathsf{F}.(\mathsf{G}.A) \\
= & \quad \{ \quad \text{factors: (1)} \quad \} \\
& \mathsf{S} \backslash (\mathsf{F}.\mathsf{Q} \backslash \mathsf{F}.(\mathsf{G}.A)) \\
\supseteq & \quad \{ \quad \text{factors: (1)} \quad \} \\
& \mathsf{S} \backslash \mathsf{F}.(\mathsf{Q} \backslash \mathsf{G}.A) \\
\supseteq & \quad \{ \quad \mathsf{S} :: \mathsf{F} \overset{\cdot}{\leftsquigarrow} \mathsf{Id} \quad \} \\
& \mathsf{Q} \backslash \mathsf{G}.A \quad .
\end{array}
$$

□

A typical use of theorems (50) and (54) is: R is F-reductive follows from the fact that there is a well-founded relation Q and a relation $S :: F \overset{\cdot}{\leftsquigarrow} \mathsf{Id}$ such that $R \subseteq F.Q \cdot S$.

As an example of this theorem, consider the largest relation R with the property that $m\langle R\rangle x$ implies that x is a natural number and m is a list of natural numbers, all smaller than x. Now consider the relation fan [2] which relates a number x to a list of arbitrary length containing only copies of x. This relation certainly has the property $\mathsf{fan} \cdot A \subseteq \mathsf{List}.A \cdot \mathsf{fan}$: if fan is applied to an argument enjoying property A, the result is a list and all of the elements in that list have property A. If fan is now composed with the relation $\mathsf{List}.<$, where $<$ is the (well-founded) less-than relation on the natural numbers, it follows that the resulting relation $\mathsf{List}.< \cdot\, \mathsf{fan}$ has precisely the properties of relation R. By instantiating Q to $<$ and G to the identity relator in theorem 54, it follows that R is List-reductive.

(This argument is in fact an instance of the generic discussion of membership in section 7.2. Associated with each container type F there is a family of fan relations such that $\mathsf{fan}_I :: F.I \leftarrow I$. Given a seed value x of type I the fan relation fan_I constructs non-deterministically an F-structure in which the value stored at each storage location is x. Given relation $R :: I \leftarrow I$, the relation $F.R \cdot \mathsf{fan}_I$ is equal to $\mathsf{mem}\backslash R$ where mem is the membership for F (of the appropriate type). See [16, 15] for further details. Thus, by applying theorem 45, $T.R \cdot \mathsf{fan}_I$ is T-reductive if R is well-founded.)

A particularly important F-reductivity theorem is:

Theorem 55 Let $R :: F.I \leftarrow I$ be F-reductive, and $S :: H.(G.I) \leftarrow G.(F.I)$ such that $S :: H \circ G \overset{\cdot}{\leftsquigarrow} G \circ F$, and G be a relator that is a lower adjoint in a Galois connection. Then $S \cdot G.R$ is H-reductive.

Proof We have to prove that $G.I \subseteq \mu(A \mapsto (S \cdot G.R) \searrow H.A)$, assuming that R is F-reductive. We in fact prove the stronger: for all F-coalgebras R

$$(56) \qquad G\,.\,\mu(A \mapsto R \searrow F.A) \;\subseteq\; \mu(A \mapsto (S \cdot G.R) \searrow H.A)$$

The theorem then follows from the assumed F-reductivity of R. Because G is a lower adjoint in a Galois connection, property (56) follows by fixpoint fusion [24] from the fact that, for all A,

$$\begin{array}{ll}
 & (S \cdot G.R) \searrow H.(G.A) \\
= & \{ \quad \text{factors: (1)} \quad \} \\
 & G.R \searrow (S \searrow H.(G.A)) \\
\supseteq & \{ \quad S :: H \circ G \overset{\cdot}{\leftsquigarrow} G \circ F \quad \}
\end{array}$$

[2] See [16, 15] for a discussion of the generic concept of a fan relation.

$$\begin{array}{ll} & G.R \searrow G.(F.A) \\ \supseteq & \{ \quad \text{factors: (1), } G \text{ is a relator} \quad \} \\ & G.(R \searrow F.A) \quad . \end{array}$$

□

The restriction imposed on relator G in this theorem is rather severe. However, there is one important class of relators satisfying it: the sections $X \mapsto J \times X$ and $X \mapsto X \times J$ of the product relator. With this instantiation of G, the theorem allows one to prove termination of programs with several parameters that are defined by structural recursion on one of the parameters.

There are, of course, many examples of such programs. Elementary examples are the inductive definitions of addition, multiplication and exponentiation on natural numbers:

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

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

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

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,

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

where

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

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.

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

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

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

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

where $\mathsf{post} = \mathsf{exr} \triangledown \mathsf{cons} \cdot \mathsf{distr}$. Here distr is as before whereas in this case pass is a function of type $1\!\!1 + (I{\times}(J{\times}K)) \leftarrow (1\!\!1 + (I{\times}J)) \times K$ that is polymorphic in I, J and K. Yet another example (which we will not spell out in detail) is the program that inserts an element in a tree. The recursion is according to the structure of its tree argument. The other argument, i.e. the element to be inserted, serves as a parameter that is only used in the "base case" of the recursion.

All these examples conform to the general form:

$$(57) \qquad X \;=\; R \cdot F.X \times P \cdot F.(I{\times}S) \times P \cdot \mathsf{pass} \vartriangle \mathsf{exr} \cdot \mathsf{in}^{\cup} \times P \quad .$$

Here P is (the identity function on) the type of the parameter. The carrier of the initial algebra in is I, and the type of X is $J \leftarrow I{\times}P$ for some J. The types of the relations R and S are $J \leftarrow F.J \times P$ and $P{\leftarrow}P$, respectively.

The generic component pass has type $F.(I{\times}P) \leftarrow F.I \times P$. Its function is to pass the parameter to all values stored in an F-structure. (Other names for it are "broadcast" [15] and "strength" [23].) In order to prove termination of program (57) we require that, for all coreflexives A under I,

$$(58) \qquad \mathsf{pass} \cdot F.A \times B \;\subseteq\; F.(A{\times}B) \cdot \mathsf{pass} \quad .$$

It can be shown that for any so-called regular relator (a relator built, possibly inductively, from constant, product, sum and map relators) such a relation pass can be constructed.

Making use of the requirement (58) on pass we can show that program (57) fulfills the conditions for theorem 55 to be applied as follows. First we note that, for all coreflexives A and B where $A \subseteq P$ and $B \subseteq P$,

$$\begin{array}{ll}
 & F.(P{\times}S) \times P \cdot \mathsf{pass} \vartriangle \mathsf{exr} \cdot F.A \times B \\
\subseteq & \{ \quad \text{distribution of composition over split} \quad \} \\
 & F.(P{\times}S) \times P \cdot (\mathsf{pass} \cdot F.A \times B) \vartriangle (\mathsf{exr} \cdot F.A \times B) \\
\subseteq & \{ \quad \text{assumption (58); computation rules} \quad \} \\
 & F.(P{\times}S) \times P \cdot (F.(A{\times}B) \cdot \mathsf{pass}) \vartriangle (B \cdot \mathsf{exr}) \\
= & \{ \quad \times\text{-}\vartriangle\text{-fusion} \quad \} \\
 & F.(P{\times}S) \times P \cdot F.(A{\times}B) \times B \cdot \mathsf{pass} \vartriangle \mathsf{exr} \\
= & \{ \quad \text{functors distribute over composition} \quad \} \\
 & F.(A{\times}(S \cdot B)) \times B \cdot \mathsf{pass} \vartriangle \mathsf{exr} \\
\subseteq & \{ \quad A \subseteq P \text{ and domains} \quad \} \\
 & F.(A \times (S \cdot B)_{<}) \times B \cdot F.(P{\times}S) \times P \cdot \mathsf{pass} \vartriangle \mathsf{exr} \quad .
\end{array}$$

Thus it follows that

$$\mathsf{F}.(\mathsf{P}{\times}\mathsf{S})\times\mathsf{P}\cdot\mathsf{pass}\vartriangle\mathsf{exr}\cdot\mathsf{F}.A\times\mathsf{P}\subseteq\mathsf{F}.(A{\times}\mathsf{P})\times\mathsf{P}\cdot\mathsf{F}.(\mathsf{P}{\times}\mathsf{S})\times\mathsf{P}\cdot\mathsf{pass}\vartriangle\mathsf{exr}\quad .$$

In other words: relation

$$\mathsf{F}.(\mathsf{P}{\times}\mathsf{S})\times\mathsf{P}\cdot\mathsf{pass}\vartriangle\mathsf{exr}$$

is a plumbing relation with type

$$({\times}\mathsf{P})\circ\mathsf{F}\circ({\times}\mathsf{P})\ \dot{\leftsquigarrow}\ ({\times}\mathsf{P})\circ\mathsf{F}\quad .$$

Furthermore, $({\times}\mathsf{P})$ is a relator which distributes over all unions of coreflexives. By theorem 55 it now follows that

$$\mathsf{F}.(\mathsf{P}{\times}\mathsf{S})\times\mathsf{P}\cdot\mathsf{pass}\vartriangle\mathsf{exr}\cdot\mathsf{in}^{\cup}\times\mathsf{P}$$

is a $({\times}\mathsf{P})\circ\mathsf{F}$-reductive relation. Hence, by definition, program (57) is a terminating program.

In this way, with one theorem we have also proved that all the examples mentioned above (addition, multiplication, exponentiation and join) are terminating programs. (Understanding how to apply the theorem is, of course, much harder than proving the termination of the individual programs on an ad hoc basis but, in our view, the extra effort is well worth while.)

From theorem 55 the next result follows as a simple corollary.

Corollary 59 If R is F-reductive and $\mathsf{S} :: \mathsf{H}\ \dot{\leftsquigarrow}\ \mathsf{F}$ then $\mathsf{S}\cdot\mathsf{R}$ is H-reductive.

Proof Instantiate theorem 55 with the identity relator (which distributes, of course, over any union).
□

Termination of the para program (see section 5)

$$X\ ::\ X=\mathsf{R}\cdot\mathsf{F}.(\mathrm{I}{\times}X)\cdot\mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I})\cdot\mathsf{in}^{\cup}$$

is now straightforward to show. Relation $\mathsf{in}^{\cup}$ is F-reductive. Furthermore, we have

$$\mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I})\cdot\mathsf{F}.A\subseteq\mathsf{F}.(\mathrm{I}{\times}A)\cdot\mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I})$$

To show this one needs that relators distribute over composition, that composition distributes over split (i.e. $\mathrm{I}{\vartriangle}\mathrm{I}\cdot\mathrm{Y}\subseteq\mathrm{Y}{\vartriangle}\mathrm{Y}$), the $\times$-$\vartriangle$-fusion rule, that A is a coreflexive and also that relators are monotonic. This means that relation $\mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I})$ is a plumbing relation of type $\mathsf{F}\circ(\mathrm{I}{\times})\ \dot{\leftsquigarrow}\ \mathsf{F}$. It now follows by corollary 59 that $\mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I})\cdot\mathsf{in}^{\cup}$ is an $\mathsf{F}\circ(\mathrm{I}{\times})$-reductive relation. Hence, by definition 43, the para program is terminating (and has, by theorem 63 proved later, a unique solution).

The proof that a mutu program (see section 5)

$$X \ :: \ X = R \cdot F.X \triangle F.X \cdot in^{\cup}$$

is terminating is similar to the proof for the para program. One needs to check that $I \triangle I$ is of type $G \stackrel{\cdot}{\leftarrow\!\sim} F$, where relator G is defined by $G.Y = F.Y \times F.Y$, i.e. one has to show that

$$I \triangle I \cdot F.A \subseteq I \triangle I \cdot F.A \times F.A$$

which follows immediately by distribution of composition over split and $\times$-$\triangle$-fusion.

8.3 Bound functions

The mathematical construction of while loops typically makes use of a so-called *bound* function, often with range the natural numbers. The idea is that termination of the loop is guaranteed if the loop body decreases the bound function at each iteration of the loop. The formal basis for the use of bound functions is the theorem that if R is a well-founded relation on the set I, and f is a function to I from some set J, then any relation S on J such that $S \subseteq f^{\cup} \cdot R \cdot f$ is well-founded. That is, S is well-founded if, for all x and y, $x\langle S\rangle y$ implies that $f.x\langle R\rangle f.y$. In particular, taking J to be the state space of the program, S to be the loop body, and R to be the less-than ordering on natural numbers, it thus follows that S is well-founded if $x\langle S\rangle y$ implies that $f.x < f.y$.

Generalising this theorem to F-reductivity, we have to take account of the fact that the outputs of an F-coalgebra are F-structures. We get:

Theorem 60 Let $R \ :: \ F.I \leftarrow I$ be an F-reductive relation and $f \ :: \ I \leftarrow J$ a functional relation. Then $F.f^{\cup} \cdot R \cdot f$ is F-reductive.

Proof We have, for all A such that $A \subseteq J$,

$$\begin{array}{ll}
 & \mu(A \mapsto (F.f^{\cup} \cdot R \cdot f) \searrow F.A) \\
= & \{ \quad \text{factors: } (1) \quad \} \\
 & \mu(A \mapsto f \searrow ((F.f^{\cup} \cdot R) \searrow F.A)) \\
= & \{ \quad \text{rolling rule} \quad \} \\
 & f \searrow \mu(A \mapsto (F.f^{\cup} \cdot R) \searrow F.(f \searrow A)) \\
= & \{ \quad \text{factors: } (1) \quad \} \\
 & f \searrow \mu(A \mapsto R \searrow F.f^{\cup} \searrow F.(f \searrow A)) \\
\supseteq & \{ \quad \text{factors: } (1),\ F \text{ is a relator, monotonicity} \quad \}
\end{array}$$

$$f\backslash\!\!\!\backslash\, \mu(A \mapsto R\backslash\!\!\!\backslash\, F.(f^{\cup}\backslash\!\!\!\backslash(f\backslash\!\!\!\backslash A)))$$

$= \quad \{ \quad$ factors: (1) $\quad \}$

$$f\backslash\!\!\!\backslash\, \mu(A \mapsto R\backslash\!\!\!\backslash\, F.((f \cdot f^{\cup})\backslash\!\!\!\backslash A))$$

$\supseteq \quad \{ \quad f \cdot f^{\cup} \subseteq I$, antimonotonicity of $\backslash\!\!\!\backslash$,

monotonicity of the other operators $\quad \}$

$$f\backslash\!\!\!\backslash\, \mu(A \mapsto R\backslash\!\!\!\backslash F.A) \quad .$$

So, if R is F-reductive, $\mu(A \mapsto (F.f^{\cup} \cdot R \cdot f)\backslash\!\!\!\backslash\, F.A) \supseteq f\backslash\!\!\!\backslash I$. The result follows from the fact that $S\backslash\!\!\!\backslash I$ equals J for all $S :: I \leftarrow J$.

□

It now follows by theorem 50 that, if R and f satisfy the conditions of theorem 60, and S satisfies the property

$$S \subseteq F.f^{\cup} \cdot R \cdot f \quad ,$$

then S is F-reductive. This condition is satisfied when f is a homomorphism to coalgebra R from coalgebra S. In particular we have:

Theorem 61 Let f be an isomorphism to F-coalgebra S from F-reductive relation R. Then S is F-reductive. In other words: reductivity is preserved under isomorphism of coalgebras.

□

9 Connections to other concepts

The notion of reductivity introduced in section 7 is novel and, as such, needs to be explored from several different angles before it can be claimed that it is the "right" notion. In this section we study the connection between reductivity and alternative notions that might have been proposed in its place.

In general, a relation on some state space is well-founded if and only if it admits induction. An alternative notion that we might wish to explore is therefore a generalisation of well-founded to "F-well-founded". This alternative is discussed in section 9.1 where it is shown that every F-reductive relation is F-well-founded. A consequence is that every hylo program with an F-reductive coalgebra is terminating. It is shown, however, that not every F-well-founded relation is F-reductive.

We also explore in section 9.2 a point-free formulation of the principle of structural induction, which we call "F-inductivity". Here we show that the converse of every total F-reductive relation is F-inductive but that it is not the case that the converse of every

F-inductive relation is F-reductive. We also show that the converse of every injective F-inductive relation is F-reductive.

9.1 Well-foundedness generalised

In general, a relation on some state space is well-founded if and only if it admits induction. Point-free formulations of these concepts have been given in [8]. Comparing these with the definition of F-reductivity it is clear that F-reductivity generalises the notion of admitting induction. Our concern in this section is with generalising the notion of well-foundedness and relating the generalised notion to F-reductivity.

Well-foundedness of relation R is equivalent to the equation $X{::}\ X = X\cdot R$ having a unique solution (which is obviously $\perp\!\!\!\perp$, the empty relation) [8]. This is easily generalised to the property that, for all relations S, the equation $X{::}\ X = S\cdot X\cdot R$ has a unique solution. The generic notion of well-foundedness we propose focusses on this unicity of the solution of equations.

Definition 62 (F-well-founded) Relation $R :: F.I \leftarrow I$ is F-*well-founded* iff, for all relations $S :: I \leftarrow F.I$ and $X :: I \leftarrow I$,

$$X = S\cdot F.X\cdot R \quad\equiv\quad X = \mu(Y \mapsto S\cdot F.Y\cdot R) \quad .$$

□

As mentioned above, a relation is Id-well-founded if and only if it is well-founded in the traditional sense [7]. So F-well-foundedness is a proper generalisation of well-foundedness.

Next we show that the property that reductivity implies well-foundedness goes through for the generalised notions. In other words: if $R :: F.I \leftarrow I$ is an F-reductive relation then, for any relation $S :: I \leftarrow F.I$, the function $Y \mapsto S\cdot F.Y\cdot R$ has a unique fixed point. This, in turn, is equivalent to: every fixed point is contained in the least fixed point. So we assume that X is an arbitrary fixed point and Z is the least fixed point of $Y \mapsto S\cdot F.Y\cdot R$. We have to show that $X \subseteq Z$ under the assumption that R is F-reductive.

$$
\begin{array}{ll}
 & X \subseteq Z \\
\equiv & \qquad \{ \qquad \text{assumption: } R \text{ is } F\text{-reductive, i.e } \mu(R\searrow\circ F) = I \quad \} \\
 & X\cdot\mu(R\searrow\circ F) \subseteq Z \\
\Leftarrow & \qquad \{ \qquad \mu\text{-fusion ;} \\
 & \qquad\qquad\quad \text{assumption: } Z \text{ is least fixed point} \quad \} \\
 & \forall(A{::}\ X\cdot R\searrow F.A \subseteq S\cdot F.(X\cdot A)\cdot R)
\end{array}
$$

$\equiv \qquad \{ \qquad$ assumption: X is a fixed point;

$\qquad\qquad\quad$ F distributes over composition $\quad \}$

$\quad \forall(A{::}\ S \cdot F.X \cdot R \cdot R\backslash F.A \subseteq S \cdot F.X \cdot F.A \cdot R)$

$\Leftarrow \qquad \{ \qquad$ monotonicity $\quad \}$

$\quad \forall(A{::}\ R \cdot R\backslash F.A \subseteq F.A \cdot R)$

$\equiv \qquad \{ \qquad$ factors: (3) $\quad \}$

$\quad \mathsf{true}\ .$

This completes the proof of the following theorem.

Theorem 63 An F-reductive relation is F-well-founded.

□

For the identity relator it is the case that "admitting induction" and "well-founded" are equivalent notions. This is not the case for the generalisations F-reductive and F-well-founded, as is demonstrated by the following counter example. Let $F.X = X{\times}X$ and consider the relation $R{\vartriangle}I$ where $R :: I{\leftarrow}I$ is a non-empty but (Id-)well-founded relation. First we demonstrate that $R{\vartriangle}I$ is F-well-founded by showing that any fixed point of the function $Y \mapsto S \cdot F.Y \cdot R{\vartriangle}I$ is equal to the least fixed point of that function.

$\quad X = S \cdot F.X \cdot R{\vartriangle}I$

$\Rightarrow \qquad \{ \qquad$ Leibniz; definition of F; $\times$-$\vartriangle$-fusion $\quad \}$

$\quad \top\!\!\top \cdot X = \top\!\!\top \cdot S \cdot (X \cdot R){\vartriangle}X$

$\Rightarrow \qquad \{ \qquad \top\!\!\top \cdot S \subseteq \top\!\!\top \quad \}$

$\quad \top\!\!\top \cdot X \subseteq \top\!\!\top \cdot (X \cdot R){\vartriangle}X$

$\equiv \qquad \{ \qquad$ domains: $\top\!\!\top \cdot Y = \top\!\!\top \cdot Y{>}$ $\quad \}$

$\quad \top\!\!\top \cdot X \subseteq \top\!\!\top \cdot ((X \cdot R){\vartriangle}X){>}$

$\Rightarrow \qquad \{ \qquad$ domains: $(Y{\vartriangle}Z){>} = Y{>} \cap Z{>}$ $\quad \}$

$\quad \top\!\!\top \cdot X \subseteq \top\!\!\top \cdot (X \cdot R){>}$

$\equiv \qquad \{ \qquad$ domains: $\top\!\!\top \cdot Y = \top\!\!\top \cdot Y{>}$ $\quad \}$

$\quad \top\!\!\top \cdot X \subseteq \top\!\!\top \cdot X \cdot R$

$\Rightarrow \qquad \{ \qquad$ assumption: R well-founded $\quad \}$

$\quad \top\!\!\top \cdot X = \perp\!\!\perp$

$\equiv \qquad \{ \qquad$ as before $\quad \}$

$X = \perp\!\!\perp$

$\equiv \quad \{ \quad F$ is $\perp\!\!\perp$-strict $\quad \}$

$X = \mu(Y \mapsto S \cdot F.Y \cdot R{\vartriangle}I)$

On the other hand $R{\vartriangle}I$ is certainly not F-reductive, something we show by contradiction.

$R{\vartriangle}I$ is F-reductive

$\Rightarrow \quad \{ \quad \mathsf{exr} \in \mathsf{Id} \overset{\cdot}{\leftsquigarrow} F$; Corollary 59;

Id-reductive implies well-founded $\quad \}$

$\mathsf{exr} \cdot R{\vartriangle}I$ is well-founded

$\equiv \quad \{ \quad \mathsf{exr} \cdot R{\vartriangle}I \;=\; I \cdot R_{>} \quad \}$

$R_{>}$ is well-founded

$\equiv \quad \{ \quad$ The only well-founded coreflexive is $\perp\!\!\perp \quad \}$

$R_{>} = \perp\!\!\perp$

However, by assumption R, and therefore $R_{>}$, differ from bottom. Hence $R{\vartriangle}I$ is not F-reductive

Because an F-reductive relation is also F-well-founded we have that a consequence of termination is that a terminating program has a unique solution (i.e. a unique input-output relation).

Theorem 64 If program $X = S \cdot F.X \cdot R$ is terminating it has a unique solution.

Proof Definition 43, theorem 63 and definition 62.
□

In order to illustrate the importance of unicity consider the following context-free grammar:

$$S \;::=\; \varepsilon \mid aSbS \mid bSaS \;\; .$$

Here ε denotes the empty word and the assumed alphabet is $\{a,b\}$.

The hylo program corresponding to this grammar (see section 6.2) is clearly terminating. Formally this is a consequence of theorem 60: the bound function is the length function on words, which is clearly reduced in every recursive call of the hylo program. It therefore follows that the language generated, $L.S$, is the unique fixed point of the hylo equation. Equivalently, $L.S$ is the unique fixed point of the equation

(65) $\quad X{::} \quad X \;=\; \{\varepsilon\} \cup \{a\}X\{b\}X \cup \{b\}X\{a\}X \;\; .$

The language generated by this grammar is in fact the set of all words with an equal number of a s and b s. Let M denote this set. The unicity property means that we can prove this fact by showing that, first,

$$M \;\supseteq\; \{\varepsilon\} \cup \{a\}M\{b\}M \cup \{b\}M\{a\}M$$

and, second,

$$M \;\subseteq\; \{\varepsilon\} \cup \{a\}M\{b\}M \cup \{b\}M\{a\}M \quad .$$

The former (which is easy to prove) shows that M is at least the least solution of (65), whilst the latter (which is the harder part to prove and, of course, depends on the alphabet being $\{a,b\}$) shows that M is at most the greatest solution of (65). Since (65) has unique solution $L.S$ it follows that M equals $L.S$.

Now consider the grammar

$$S \quad ::= \quad \varepsilon \mid aSb \mid bSa \mid SS \quad .$$

Straightforward fixed point calculus shows that the languages generated by the two grammars are equal. However, the hylo equation corresponding to this grammar is not terminating. Indeed it is easy to see that $\{a,b\}^*$ is also a solution of the equation

$$X\!:: \quad X \;=\; \{\varepsilon\} \cup \{a\}X\{b\} \cup \{b\}X\{a\} \cup XX \quad .$$

The task of proving that the language generated by this grammar is M cannot be achieved by using the same strategy. Thus either one has to show that the transformation to the original grammar is valid, or one has to use an inductive argument based on the length of words in M. The former strategy is, in our view, preferable in that it separates the proof into distinct lemmas, each of which is relatively straightforward and each of which adds additional insight.

9.2 Structural Induction

Structural induction is the standard induction scheme that is part of the definition of recursive datatypes. For instance, structural induction over the type of natural numbers is what is usually called *the* principle of induction, and its validity is one of the defining properties of the naturals. In this section we present a point-free relational definition of structural induction and relate it to reductivity.

The principle of induction on natural numbers can be expressed informally as: a property is true of all natural numbers if it is an *invariant* of $\mathsf{zero}\triangledown\mathsf{succ}$. By this we mean that the property is established by zero —a property is an "invariant" of a constant function if the result of the function satisfies the property— and the property is an invariant of succ if the function succ maps numbers satisfying the property to numbers also satisfying the property.

The question we have to tackle is how to formalise the notion of "invariance". We propose calling a coreflexive A an *invariant* of R iff

$$(R \cdot F.A)_< \subseteq A \quad .$$

Equivalently, in the predicate calculus, A is an *invariant* of F-algebra R iff

$$\forall (x : \exists (y : x\langle R\rangle y : y \in F.A) : x \in A) \quad .$$

We call this property an invariance property because it expresses the idea that an F-structure (y) all of whose elements satisfy property A ($y \in F.A$) is mapped by R into a value (x) also satisfying A ($x \in A$).

Our notion of a relation R, being "inductive" with respect to F, is that it is possible to deduce that all elements of the left domain of R satisfy some property A whenever A is an invariant of R.

Definition 66 (F-inductivity) A relation $R :: I \leftarrow F.I$ is said to be F-*inductive* if, for all coreflexives A under I,

(67) $\quad I \subseteq A \;\Leftarrow\; (R \cdot F.A)_< \subseteq A \quad .$

□

To make clear that definition 66 indeed captures structural induction, we consider the datatype of lists. Let F be the relator that maps X to $J+(\mathbb{1}+(X\times X))$ and R the function $\tau_J \triangledown (\mathsf{nill}_J \triangledown \mathsf{join}_J)$. Thus R is a coalgebra with carrier I equal to $\mathsf{List}.J$ consisting of the junc of the relations τ_J (the function that maps a point x of type J to the singleton list $[x]$), nill_J (the constant function mapping the element of the unit type to the empty list of type $\mathsf{List}.J$) and join_J (the function that concatenates two lists of type $\mathsf{List}.J$). We now show that R is F-inductive.

With the above substitutions for F and R, the antecedent of (67) can be rewritten:

$$
\begin{array}{ll}
 & (\tau_J \triangledown (\mathsf{nill}_J \triangledown \mathsf{join}_J) \;\cdot\; J+(\mathbb{1}+(A\times A)))_< \;\subseteq\; A \\
\equiv & \qquad \{ \qquad \triangledown\text{-}+\text{-fusion},\; \tau_J \cdot J = \tau_J\,,\; \mathsf{nill}_J \cdot \mathbb{1} = \mathsf{nill}_J \quad \} \\
 & (\tau_J \;\triangledown\; (\mathsf{nill}_J \;\triangledown\; (\mathsf{join}_J \cdot A\times A)))_< \;\subseteq\; A \\
\equiv & \qquad \{ \qquad (X \triangledown Y)_< = X_< \cup Y_< \quad \} \\
 & (\tau_J)_< \subseteq A \;\;\wedge\;\; (\mathsf{nill}_J)_< \subseteq A \;\;\wedge\;\; (\mathsf{join}_J \cdot A\times A)_< \subseteq A \quad .
\end{array}
$$

The first two conjuncts can be interpreted as: all singleton lists and the empty list satisfy property A. This is the base case of a familiar type of inductive argument. The second conjunct is the induction step: if two lists satisfy property A the join of the two lists also satisfies A. Clearly, if the base case and induction step are true, all lists satisfy A. This last fact can be restated as the consequent of (67) because the left domain of R equals the type of lists: any list can be constructed as either a singleton, the empty list or the concatenation of two lists. Therefore, with this choice of F and R formula (67) holds true: the relation $\tau\triangledown(\mathsf{nill}\triangledown\mathsf{join})$ is $(X\mapsto I+(\mathbb{1}+(X\times X)))$-inductive.

If we substitute the relator $(\mathbb{1}+)$ for F and the relation $\mathsf{zero}\triangledown\mathsf{succ}$ for R in the antecedent of (67), a calculation similar to the previous one can be carried out. It can then be verified that the resulting formula is equivalent to the familiar formulation of the base case and "step" of the induction principle for the natural numbers. So, the statement that $\mathsf{zero}\triangledown\mathsf{succ}$ is $(\mathbb{1}+)$-inductive is equivalent to the statement that standard induction over the natural numbers is allowed.

There is another way of justifying the definition of inductivity which we will just sketch. Recall that definition 43 of termination depends on the assumption that if, due to non-determinism there is, at a certain point during the execution, more than one possibility to proceed, only one of those possibilities is chosen. Had we adopted the other assumption, viz. that all possible continuations of the executions are pursued, it would have turned out that the maximal safe set for coalgebra R should be a solution of the equation $B=(F.B\cdot R)>$. The argument in this case is that a set A is safe if and only if a computation of R started in set A has at least one output for which every recursive call is in the safe set B. That is,

$$A\subseteq B \;\equiv\; A\subseteq (F.B\cdot R)> \quad .$$

Thus inductivity corresponds to an angelic notion of termination whereas reductivity is demonic.

Recall that reductivity was meant to formalise strong induction, that is it should be in a sense stronger than inductivity. Since inductivity is a property of algebras and reductivity is a property of coalgebras, the right question to ask is: is the converse of a reductive relation inductive? This turns out to be almost true.

Theorem 68 Let R be an F-reductive relation such that $R< \subseteq F.R>$. Then $R^{\cup}$ is F-inductive.

Proof Let I be the carrier of R. Then, for all coreflexives A under I,

$$\begin{array}{ll} & R^{\cup}< \subseteq A \\ \equiv & \{ \qquad X< = X^{\cup}> \qquad \} \end{array}$$

$$
\begin{array}{ll}
 & R{>} \subseteq A \\
\equiv & \{ \quad \text{factors: (1)} \quad \} \\
 & I \subseteq R{>} \setminus A \\
\Leftarrow & \{ \quad R \text{ is } F\text{-reductive} \quad \} \\
 & R \setminus F.(R{>} \setminus A) \subseteq R{>} \setminus A \\
\Leftarrow & \{ \quad \text{factors (1), relators} \quad \} \\
 & R{>} \cdot R \setminus (F.R{>} \setminus F.A) \subseteq A \\
\equiv & \{ \quad \text{factors (1)} \quad \} \\
 & R{>} \cdot (F.R{>} \cdot R) \setminus F.A \subseteq A \\
\equiv & \{ \quad \text{assumption: } R{<} \subseteq F.R{>} \quad \} \\
 & R{>} \cdot R \setminus F.A \subseteq A \\
\Leftarrow & \{ \quad R{>} \cdot R \setminus F.A = (R \cdot R \setminus F.A){>}, \quad \text{factors: (2)} \quad \} \\
 & (F.A \cdot R){>} \subseteq A \\
\equiv & \{ \quad X{<} = X^{\cup}{>} \quad \} \\
 & (R^{\cup} \cdot F.A){<} \subseteq A \quad .
\end{array}
$$

□

An immediate corollary is that the converse of a *total*, reductive coalgebra is inductive. This totality restriction is not severe and, indeed, is often desirable.

Next, we address the question whether reductivity is really stronger than inductivity. Does there exist an inductive relation such that its converse is not reductive? To find such a counter example we first prove a theorem that gives a sufficient condition such that inductive implies reductive. The theorem can be read as: the converse of an inductive injection is reductive.

Theorem 69 If $R :: I \leftarrow F.I$ is an F-inductive relation such that $R^{\cup} \cdot R \subseteq I$, then $R^{\cup}$ is F-reductive.

Proof

$$
\begin{array}{ll}
 & I \subseteq A \\
\Leftarrow & \{ \quad R \text{ is } F\text{-inductive, definition 66} \quad \} \\
 & (R \cdot F.A){<} \subseteq A \\
\equiv & \{ \quad X{<} = X^{\cup}{>}\text{; distribution properties of } {}^{\cup} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
 & (F.A \cdot R^{\cup})_{>} \subseteq A \\
\equiv & \{ \quad \text{for single-valued } f \text{ and all } A, \quad f_{>} \cdot f \backslash A \;=\; (A \cdot f)_{>} \;, \\
 & \quad \text{by assumption } R^{\cup} \text{ is single-valued} \quad \} \\
 & R^{\cup}{}_{>} \cdot R^{\cup} \backslash F.A \;\subseteq\; A \\
\Leftarrow & \{ \quad R^{\cup}{}_{>} \subseteq I \quad \} \\
 & R^{\cup} \backslash F.A \subseteq A \quad .
\end{array}
$$

□

To find a relation that is inductive but whose converse is not reductive we therefore have to look at non-injective inductive relations. The function $\tau \triangledown (\mathsf{nill} \triangledown \mathsf{join})$ introduced earlier in this section offers just such an example. The converse of this relation is not reductive. It is easy to check that the following holds:

$$\mathsf{exl} \cdot \mathsf{inr}^{\cup} \cdot \mathsf{inr}^{\cup} \;\; :: \;\; \mathsf{Id} \hookleftarrow F$$

where F is the relator $X \mapsto I+(\mathbb{1}+(X\times X))$. So, if $(\tau \triangledown (\mathsf{nill} \triangledown \mathsf{join}))^{\cup}$ is F-reductive, the relation

$$\mathsf{exl} \cdot \mathsf{inr}^{\cup} \cdot \mathsf{inr}^{\cup} \cdot (\tau \triangledown (\mathsf{nill} \triangledown \mathsf{join}))^{\cup}$$

is Id-reductive by corollary 59 — in other words, it is well-founded. However this relation is equal to $\mathsf{exl} \cdot \mathsf{join}^{\cup}$, a relation that relates the empty list to the empty list, which is clearly not well-founded.

10 Generic Unification

In this section we provide yet more evidence for the practical relevance of the notion of reductivity. The example is drawn from the emerging field of “generic programming” [5]. It concerns the proof of correctness of a generic unification algorithm. Such an algorithm has been formulated by Jeuring and Jansson [17] but without proof of correctness. The algorithm is “generic” in the sense that it is parameterised by a functor F that specifies the structure of expressions to be unified.

Here we consider just one small aspect of such a proof of correctness of the algorithm. Specifically, we show that the “occurs-properly-in” relation on expressions is well-founded. Particularly remarkable about our proof is that it is very simple. This is a result of its not requiring the definition of a size function on expressions in any way, the key to the proof being instead the fact that the converse of an initial F-algebra is F-reductive.

In its generic form, unification takes the following form. A parameter is a relator F. A second parameter is a type V, elements of which are called *variables*. Given these two we may define a relator F_V which maps relation X to $\mathsf{F}.X + \mathsf{id}_V$. Then we assume that in is an initial F_V-algebra with carrier $\mathsf{F}^\star V$. That is,

$$\mathsf{in} \ :: \ \mathsf{F}^\star V \leftarrow \mathsf{F}.\mathsf{F}^\star V + V \quad .$$

The relator $\mathsf{F}^\star$ (together with appropriately defined unit and multiplier) is a monad which, as the Kleene star-like notation suggests, is obtained by repeated application of the relator F. Elements of $\mathsf{F}^\star V$ are called *expressions*; the parameter F limits the way that new expressions are built up out of subexpressions. Substitution of an expression for a variable can now be defined in such a way that the composition of substitutions is Kleisli composition in the monad. The ordering "more general than" on substitutions is then defined in the usual way. Generic unification is then the problem of finding a substitution that unifies two expressions and is more general than any other unifier.

As remarked earlier, Jeuring and Jansson have formulated a generic unification algorithm but without proof of correctness. One element of such a proof of correctness is to show that if a variable occurs in an expression then the variable and expression are not unifiable. The way to do this is to define an "occurs-properly-in" relation between expressions, show that this relation is well-founded (and thus is irreflexive) and finally show that it is preserved by substitution. Here we will just show the first two of these steps as an illustration of the reductivity calculus.

Suppose mem is the membership relation of the relator F. Let $\mathsf{inl}_{A,B}$ denote the injection function of type $A+B \leftarrow A$. (We will drop subscripts from now on for simplicity.) Then we can define the relation $\mathsf{occurs\_properly\_in}$ of type $\mathsf{F}^\star V \leftarrow \mathsf{F}^\star V$ by

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

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

The $\mathsf{occurs\_properly\_in}$ relation has a very simple structure. We ought to be able to see that it is well-founded almost directly just from that structure. Indeed this is what the reductivity calculus allows us to do. The lemma and its proof follow. The first step involves a well-known property of well-founded relations. Otherwise, every non-trivial step uses the theory of reductivity developed in this paper.

Lemma 70 The relation $\mathsf{occurs\_properly\_in}$ is well-founded.

Proof

$$
\begin{array}{ll}
 & \mathsf{occurs\_properly\_in} \ \text{is well-founded} \\
\equiv & \{ \quad \text{definition of } \mathsf{occurs\_properly\_in}\,, \\
 & \quad\; R \ \text{is well-founded} \ \equiv \ R^{+} \ \text{is well-founded} \quad \} \\
 & \mathsf{mem}\cdot(\mathsf{in}\cdot\mathsf{inl})^{\cup} \ \text{is well-founded} \\
\Leftarrow & \{ \quad \mathsf{mem}\cdot R \ \text{is well-founded} \ \equiv \ R \ \text{is } F\text{-reductive} \quad \} \\
 & (\mathsf{in}\cdot\mathsf{inl})^{\cup} \ \text{is } F\text{-reductive} \\
\equiv & \{ \quad (\mathsf{in}\cdot\mathsf{inl})^{\cup} = \mathsf{inl}^{\cup}\cdot\mathsf{in}^{\cup}\,, \quad \} \\
 & \mathsf{inl}^{\cup}\cdot\mathsf{in}^{\cup} \ \text{is } F\text{-reductive} \\
\Leftarrow & \{ \quad \text{If } R :: G.I \leftarrow I \text{ is } G\text{-reductive and } S :: F \mathrel{\dot{\leftsquigarrow}} G \\
 & \quad\; \text{then } S\cdot R \text{ is } F\text{-reductive.} \quad \} \\
 & \mathsf{in}^{\cup} \ \text{is } F_V\text{-reductive} \ \wedge \ \mathsf{inl}^{\cup} :: F \mathrel{\dot{\leftsquigarrow}} F_V \\
\Leftarrow & \{ \quad \text{the converse of an initial } G\text{-algebra is } G\text{-reductive,} \\
 & \quad\; \text{definition of } \dot{\leftsquigarrow} \quad \} \\
 & \mathsf{true} \ \wedge \ \forall\langle A{::}\ F_V.A \subseteq \mathsf{inl}^{\cup} \searrow F.A\rangle \\
\equiv & \{ \quad F_V.A \ = \ F.A + \mathsf{id}_V\,, \text{ factors (1) and computation rule} \quad \} \\
 & \mathsf{true} \ .
\end{array}
$$

□

Note that the proof is entirely algebraic. We have given an "intuitive" justification for the definition of occurs_properly_in but nowhere do we rely on that "intuition". Note also that the proof does not involve any notion of the "size" of expressions. Many well-foundedness arguments are based on defining a variant function with range the natural numbers and exploiting their well-foundedness. The above proof is based on the basic reductivity theorem that the converse of an initial G-algebra is G-reductive, a consequence of which theorem is that the natural numbers are well-founded. Introducing the natural numbers into the proof would be introducing unnecessary detail.

11 Conclusion

This paper has introduced a methodology for recursive program design that extends the well-established design methodology for imperative programming. Within the methodology, a relator F plays the role of solution strategy, whether that be simple iteration (as in imperative programming), primitive recursion or some other divide-and-conquer scheme.

Making progress in a recursive computation is modelled by the notion of F-reductivity, also introduced in this paper. Parameterising progress properties by the solution strategy is, we feel, an extremely important innovation since it opens up the possibility of relating one form of reductivity to another. Such rules, combined with the central result that the converse of an initial F-algebra is F-reductive, allow straightforward proofs of program termination, often at a generic level. Several examples have been shown.

Strong evidence has been provided for basing program development on relation algebra, even when the desired implementation vehicle is a functional programming language. A particularly good example of this is the discussion of parsing in section 6.2. A similar discussion would be impossible in a functional programming framework because of the non-determinism in a typical context-free grammar. Using relation algebra, however, it is straightforward to show that parsing is implemented by a hylo program *whatever the form of the context-free grammar*.

The paper has also discussed the relationship between reductivity, well-foundedness and structural induction. Generic formulations of the latter two notions have been presented, and the precise mathematical relationship with reductivity has been explored.

There are several directions in which the current work can be extended. The rules on F-reductivity presented in section 8 are clearly incomplete. More effort needs to be expended on building up a useful collection of rules. For example, it should be possible to develop rules based on the structure of the relator F (whether it is the sum of two relators or the product of two relators, etc.). What is remarkable about the rules presented in section 8 is that, in some cases, they reduce proofs of program termination to a process akin to type checking. The core of the termination argument is the presence of (the converse of) an initial G-algebra in the program, for some G; this is combined with plumbing relations to construct the desired F-reductive relation. This paves the way for the possibility of verifying the termination of hylo programs at the compilation stage. The process will never be complete in a formal sense but there is a good possibility that it is sufficiently powerful to make it worth the while.

The notion of termination of programs is based here on a demonic model of program execution. Our work could be used as inspiration for a study of termination properties based on an angelic model of computation. Such a study would lead to theorems and lemmas like the ones in section 8 and could be useful in gaining a better understanding of the design of logic programs and distributed programs.

Most of the results in this paper are of a generic nature. That is, the design strategy F is a parameter of the theorem or lemma. Generic programming, whereby the structure of the data and or problem-solving strategy is a parameter, has much, as yet unexplored, potential. This paper establishes a theoretical basis for generic programming that is simple and effective.

References

[1] 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.

[2] 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.

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

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

[5] 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.

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

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

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

[9] 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.

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

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

[12] Peter Freyd. Algebraically complete categories. In G. Rosolini A. Carboni, M.C. Pedicchio, editor, *Category Theory, Proceedings, Como 1990*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, 1990.

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

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

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

[16] Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

[17] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

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

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

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

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

[22] 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.

[23] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[24] B. Möller, editor. *Mathematics of Program Construction, 3rd International Conference*, number 947 in LNCS. Springer-Verlag, 1995.

[25] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[26] J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E. Mason, editor, *IFIP '83*, pages 513–523. Elsevier Science Publishers, 1983.

[27] 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.

[28] P. Wadler. Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture, ACM, London*, September 1989.