Datatype-Generic Termination Proofs

Roland Backhouse and Henk Doornbos

rcb@cs.nott.ac.uk
School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, England,
henk.doornbos@questance.com
Questance, Agricolastraat 32, 9711 TS Groningen, The Netherlands

Abstract. Datatype-generic programs are programs that are parameterised by a datatype. We review the allegorical foundations of a methodology of designing datatype-generic programs. The notion of F-reductivity, where F parametrises a datatype, is reviewed and a number of its properties are presented. The properties are used to give concise, effective proofs of termination of a number of datatype-generic programming schemas. The paper concludes with a concise proof of the well-foundedness of a datatype-generic occurs-in relation.

Keywords: datatype, generic programming, relation algebra, allegory, programming methodology

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 attention —without loss of generality— to just the class of **while** programs. It is neither necessary nor desirable to consider arbitrary **goto** programs.

The systematic use of induction on the structure of datatypes is another such discipline; defining and exploiting application-specific datatypes is sound practice, as is well known. This has led to the development of a new programming concept, called *(datatype-)generic* programming [23, 16, 17, 24, 15]. Datatype-generic programs are programs that are parameterised by a data structure. For example, the compression of data can be much more effective if the specific structure of the data is known in advance — the compression of, *eg*, computer programs can exploit their specific syntactic structure to achieve a higher compression ratio [17].

Nowadays, there is a vast amount of literature on the *post hoc* verification of programs. That is not our concern. Our concern is with identifying concepts and methods that assist the practising programmer in the *construction* of programs. The focus in this paper is on the concepts fundamental to guaranteeing the termination of programs. The idea of making data structure a parameter of termination properties —the development of a *datatype-generic* theory of program termination— is the major new insight that we explore in depth.

Of course, program termination can always be guaranteed by limiting the syntactic expressivity of a programming language. Programs limited to **for** loops are guaranteed to terminate, as are programs limited to list comprehension on finite lists or to so-called "folds" ("catamorphisms") on inductively defined datatypes. But such constraints are —rightly— rejected by the practising programmer. Also —in theory— program termination can always be guaranteed by exhibiting a function from the program state to a natural number (a so-called "bound function") and demonstrating that the value of the function is always strictly decreased in the course of the computation. But this elevates (induction over) the datatype of natural numbers to a canonical position that it does not occupy in the work of the practising programmer.

For the practising programmer, induction does play a central rôle in program construction, but it is certainly not limited to the natural numbers, and the form in which it is used is often only implicit in the program structure. Thus, at the core of any algorithm is induction over some structure, but this may be concealed by a variety of programming mechanisms such as additional parameters, some form of preprocessing or by transformations from one structure to another. In this paper, we develop a calculus of "F-reductivity" which embodies a datatype-generic discipline of program construction, emphasising in particular sound guarantees of program termination. The word "reductivity" refers to the basic process of making progress in a computation by reduction; the parameter "F" reflects the structure, or datatype, to which reduction is applied.

The notion of F-reductivity was introduced by the authors in [10, 8, 11]. The starting point in [11] was the notion of an initial F-algebra, which is the basis for the use of "folds" on a datatype. Our goal was to explore generalisations of initiality that better reflect the less restricted style of programming practice. To this end, we identified and compared three different datatype-generic properties of a relation — "F-reductive", "F-well-founded" and "F-inductive". We presented the theorem that these three notions coincide when the relation in question is a (functional) bijection. (The property of being a bijection is what we wanted to abandon.) The conclusion of the comparison was that the property most relevant to the practising programmer is F-reductivity. We showed, for example, that F-well-foundedness guarantees that a recursive specification has a unique solution, but that this does not guarantee that an operational interpretation of the specification will terminate. The current paper focuses on the calculus of F-reductivity, which was briefly mentioned at the end of this earlier paper.

The main contributions of the current paper begin in section 5. Earlier sections review relation algebra (section 2), allegory theory and relators (section 3; "relators" are essentially datatypes) and the definition of reductivity. Much of the material in these sections has been published elsewhere, but its reproduction here helps to make the paper relatively self-contained. Section 4.2 also contains a short discussion of the generic notion of "membership" of a datatype, the characteristic feature of a so-called "collection type" [20, 19], so that we can give a concrete interpretation of F-reductivity for such datatypes.

The discussion of the calculus of F-reductivity in section 5 begins with what can be described as the "core" theorem, specifically that the converse of every initial F-algebra is F-reductive. Indeed, every recursive computation has at its "core" the converse of an initial F-algebra, but that may not be immediately obvious because of the additional supporting computations. The remaining theorems in this section can be loosely described as ways of transforming reductive relations to reductive relations, possibly involving a change of the type of the reductivity (that is, a change in the parameter "F"). For example, theorem 5 captures the condition under which preprocessing of input values preserves termination properties, whilst theorem 9 does the same for when the structure of the data is modified by a polymorphic computation (formally, a natural transformation between datatypes). The theorems, all of which are very general and generic in the datatype, are prefaced by familiar concrete examples which demonstrate their application.

Section 6 reformulates the definitions of "F-well-founded" and "F-inductive" introduced in [11], in a typed, as opposed to untyped, framework. Two new theorems are stated and proved, sharpening and reinforcing the argument in [11] for the focus on reductivity. The problem of parsing context-free grammars provides a non-trivial, concrete example.

The paper is concluded by establishing the well-foundedness of the occurs-in relation in a dataype-generic unification algorithm [22]. Comparison of the —much shorter— proof presented here with that in [6] demonstrates our thesis that the commonplace reliance on induction on numbers can be inappropriate and ineffective. The practising programmer needs to be conversant with a much broader, datatype-generic class of programming principles.

2 Relation algebra

2.1 Basic Definitions

Although much recent work on datatype-generic programming has been conducted within the paradigm of *functional* programming, there are far-reaching arguments for adopting a relational framework. Two directly relevant to the current paper are: specifications are typically nondeterministic (i.e. relations, not functions) and termination arguments are almost always conducted within the framework of well-founded relations. So, for us, a 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 (as in functional programming). Formally, a (binary) relation is a triple consisting of a pair of types I and J, say, and a subset of the cartesian product $I{\times}J$. We say *R has type $I{\leftarrow}J$* (read "I from J"), the left-pointing arrow indicating that we view I as the set of possible outputs and J as the set of possible inputs. I is called the *target* and J the *source* of the 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 converse of relation R is denoted by $R^{\cup}$. Relations of the same type are ordered by set inclusion denoted in the conventional way by the infix $\subseteq$ operator. The relations of a given type $I{\leftarrow}J$ form a complete lattice under this ordering. The smallest relation of type $I{\leftarrow}J$ is the empty relation, denoted here by $\perp\!\!\!\perp_{I\leftarrow J}$, and the largest relation of type $I{\leftarrow}J$ is the universal relation, which we denote by $\top\!\!\!\top_{I\leftarrow J}$. (We use this notation for the empty and universal relations because the conventional notation $\top$ for the universal relation is easily confused with T, a sans serif letter T, particularly in hand-written documents.)

For each set I, there is an identity relation which we denote by id_I. Thus id_I has type $I{\leftarrow}I$. Relations of type $I{\leftarrow}I$ contained in id_I will be called *coreflexives under* I (or just *coreflexives* if the type is evident). By convention, we use R, S, T to denote arbitrary relations and A, B and C to denote coreflexives. Clearly, the coreflexives under I are in one-to-one correspondence with the subsets of I; we exploit this correspondence by identifying subsets of I with the coreflexives under I.

Functions are total, single-valued relations. Formally, relation R of type $I \leftarrow J$ is *total* iff $\mathsf{id}_J \subseteq R^{\cup} \cdot R$; also, R is *single-valued* if $R \cdot R^{\cup} \subseteq \mathsf{id}_I$ where I is the target of R. We use an infix dot to denote function application. Thus $f.x$ denotes application of function f to argument x. Dual to the notion of single-valued 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 "single-valued" 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 $f.x$ (with the function to the left of its argument) rather than say x^f . A sensible consequence is that type arrows point from right to left.

We use several infix operators throughout the paper. Our precedence convention is that subscripting, superscripting and all unary operators have the highest priority; next in priority is function application, followed by "arithmetic-like" operators (eg. cartesian product "$\times$") and then composition. The subset and equality relations and the logical operators have lowest precedence, in the usual way.

2.2 Domains and Division Operators

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

$$(1) \quad \langle \forall A :: A \cdot R = R \equiv R{<} \subseteq A \rangle \quad .$$

Given a coreflexive A under I, the relation $A \cdot R$ can be viewed as the relation R restricted to outputs in the set A. Thus, in words, the left domain of R is the least coreflexive A that maintains R when R is restricted to outputs in the set A. The *right domain* $R{>}$ is defined symmetrically by reversing the composition $R\cdot A$. The left/right domain should not be confused with the target/source of the relation.

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 property that, for all relations S of type $J{\leftarrow}K$,

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

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. Interpreting relations as specifications, the above defines $R\backslash T$ to be the “weakest” specification of a program S such that executing R after S satisfies specification T. With this interpretation, $R\backslash T$ has been called a *weakest prespecification* [18].

The *weakest liberal precondition* operator will be denoted here by the symbol “ $\mathbin{\backslash\!\!\shortmid}$”. Formally, if R is a relation of type $I{\leftarrow}J$ and A is a coreflexive under I then $R\mathbin{\backslash\!\!\shortmid}A$ is a coreflexive under J characterised by the property that, for all coreflexives B under J,

$$(2)\qquad (R\cdot B){<}\subseteq A \;\equiv\; B\subseteq R\mathbin{\backslash\!\!\shortmid}A \quad .$$

Again, we use a division-like notation, rather than “wlp”, to emphasise the similarity with division in normal arithmetic. (The corresponding “multiplication” operator is the function that maps R and B to $(R\cdot B){<}$.) Informally, $R\mathbin{\backslash\!\!\shortmid}A$ is the set of inputs that are related by R to outputs in A only.

Two immediate consequences of (2) that we use frequently are:

$$(3)\qquad (R\cdot R\mathbin{\backslash\!\!\shortmid}A){<}\subseteq A \quad ,$$

and

$$(4)\qquad R\mathbin{\backslash\!\!\shortmid}(S\mathbin{\backslash\!\!\shortmid}A) \;=\; (S\cdot R)\mathbin{\backslash\!\!\shortmid}A \quad .$$

Property (3) is obtained by instantiating B to $R\mathbin{\backslash\!\!\shortmid}A$ in (2); property (4) is a simple application of indirect equality combined with the properties of the domain operators. See [4] for details.

3 Allegories and Relators

We assume that the reader is familiar with the most basic notions of category theory, namely objects, arrows, functors, natural transformations and (initial) algebras We use *Fun* to denote the category with sets as objects and functions between sets as arrows. We use *Rel* to denote the category with sets as objects and binary relations as arrows. We also assume familiarity with the relevance of these concepts to functional programming: functors correspond to type constructors and natural transformations correspond to polymorphic functions.

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" [14] extends the notion of a category in order to better capture the essential properties of relations, and the notion of a "relator" [1, 3, 4] 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. For practical purposes, more is needed. A *locally-complete, tabulated, unitary, division allegory* is an allegory such that, for each pair of objects A and B, the partial ordering on the set of arrows of type $A{\leftarrow}B$ is complete ("locally-complete"), the division operators introduced in section 2.2 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". "Tabulated" captures the fact that relations are subsets of the cartesian product of a pair of sets [7]. (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" [1]. A *relator* is a functor whose source and target are both allegories, and 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) $F.R$ has type $F.I \xleftarrow{\mathcal{C}} F.J$ whenever R has type $I \xleftarrow{\mathcal{D}} J$.

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

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

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

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

For example, List is a unary relator, and product and sum are binary relators. List is an example of an inductively defined datatype; in [2] it was observed that all inductively defined datatypes are relators. If R is a relation of type $I \leftarrow J$, $\mathsf{List}.R$ 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 $R{\times}S$ (called the *product* of R and S) relates two pairs if the first components are related by R and the second components are related by S; it has type $I{\times}J \leftarrow K{\times}L$ if R has type $I \leftarrow K$ and S has type $J \leftarrow L$. Similarly, the relation $R{+}S$ (called the *sum* of R and S) has type $I{+}J \leftarrow K{+}L$ if R has type $I \leftarrow K$ and S has type $J \leftarrow L$. It relates two tagged values if they have the same tag and either their common tag indicates that the output value is in I and the input value is in K and the output and input values are related by R, or their common tag indicates that the output value is in J and the input value is in L and the output and input values are related by S.

A common device, used to construct relators, is so-called *sectioning* of a binary relator. For example, if I is a type, the *section* $(I+)$ denotes the relator that maps type J to $I{+}J$, and relation R (of type $J \leftarrow K$) to the relation $\mathsf{id}_I{+}R$ of type $I{+}J \leftarrow I{+}K$. Similarly, $(I\times)$ and $(\times J)$ denote sections of the product relator.

Of course, relators of compatible types can be composed, in just the same way that functors are composed. If F and G are relators, $F{\circ}G$ denotes their composition.

A design requirement, that dictates the above definition of a relator, 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.

Recall that 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 $Map(\mathcal{A})$. In particular, $Map(Rel)$ is the category Fun. Now, the desired property of relators is that relator F of type $\mathcal{A} \leftarrow \mathcal{B}$ is a functor of type $Map(\mathcal{A}) \leftarrow Map(\mathcal{B})$. It is easily shown that our definition of relator guarantees this property.

(Bird and De Moor [7] omit (9) and define a relator to be a monotonic functor. However, their proof of their theorem 5.1, which purports to justify the omission, is incorrect; it is an open question whether (9) can indeed be omitted.)

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 natural transformations [32, 33]. However, caution is needed when extending the categorical notion of natural transformation to allegories. In the latter context, the term *lax natural transformation* is sometimes used. The collection of lax natural transformations to relator F from relator G is denoted by $F \hookleftarrow G$ and defined by

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

A relationship between naturality in the allegorical sense and in the categorical sense is the following [19]. Recall that relators respect functions, i.e. relators are functors on the sub-category Map. Then, in the case that all elements of the collection α are *functions*,

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

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 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 A Programming Paradigm

4.1 Hylo programs

As discussed in the introduction, a programming methodology is characterised by a discipline that maximises confidence in the end product by constraining the construction methods. The methods should be simple and easy to use, whilst not forming an impediment to program construction.

The programs in the class on which our discipline is based are called *hylomorphisms*. The fact that many recursively defined functional programs are hylomorphisms was identified by Fokkinga, Meijer and Paterson [29], the name having been coined by Meijer [30]. Unlike [29], however, the current paper is not restricted to functional programs.

Definition 1 (Hylos). Let F be a relator and let R and S be relations of type $I \leftarrow F.I$ and $F.J \leftarrow J$, respectively. An equation in X (of type $I \leftarrow J$) of the form $X = R \cdot F.X \cdot S$ is said to be a *hylo equation* or *hylo program*.
□

The hylo recursion scheme offers substantial 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 $(I+)$ encapsulates repetition — it maps relation X to $\mathsf{id}_I{+}X$, which expresses a choice ("+") between terminating the repetition ("id_I") and repeating X . Similarly, $(I+)\circ Square$ (where $Square.X = X{\times}X$) encapsulates a divide and conquer strategy (choose between terminating and dividing the problem into two subproblems), and $F\circ(I\times)$ encapsulates primitive recursion (structural induction, the form of which is given by the relator F, on the input value, of which a copy is retained ("$I\times$")). A first step in the design of hylo programs is thus the choice of the relator [10]. Extending hylo programs to allow relations as components is also a significant advance on the functional paradigm. Relations on strings, like the prefix, suffix, subsequence and segment relations are easy to express as hylo equations, as can quite complex problems like context-free language recognition.

Crucial to developing a discipline of hylo programming is that the meaning of a hylo equation is well-understood, both as a specification of a relation, and operationally as a program that can be executed. The operational meaning demands an understanding of how hylo equations are executed, including when they are guaranteed to terminate. This is discussed in section 4.2. The specificational meaning can be understood in several ways. One is to extrapolate from the now well-understood notion of a catamorphism on an initial F-algebra. This is captured by theorem 1, below. The definition of a "relational initial F-algebra" is needed first.

Definition 2. Assume that F is an endorelator. Then (I, in) is a *relational initial* F-algebra iff in has type $I \leftarrow F.I$ (and thus is an F-algebra), and there is a mapping $(\!|\_|\!)$ defined on all F-algebras such that

(11) $(\!|R|\!)$ has type $I \leftarrow J$ if R has type $J \leftarrow F.J$,

(12) $(\!|\mathsf{in}|\!) = \mathsf{id}_I$, and

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

I.e., $(\!|R|\!) \cdot (\!|S|\!)^{\cup}$ is the smallest solution of the equation in X, $R \cdot F.X \cdot S^{\cup} \subseteq X$. □

Definition 2 makes use of the "banana brackets", $(\!|\_|\!)$, introduced by Malcolm [25, 26] to denote a functional/relational catamorphism. In categorical terms, catamorphisms are the unique arrows from the initial object in the category of F-algebras; in programming terms, catamorphisms are programs defined by structural induction on a datatype. The definition extends the categorical notion of an initial F-algebra to allegories in a way that is made precise by the hylo theorem below. Recall that $Map(\mathcal{A})$ denotes the sub-category of functions in the allegory $\mathcal{A}$. For clarity, we distinguish between the endorelator F and the corresponding endofunctor, F', defined on $Map(\mathcal{A})$.

Theorem 1 (Hylo Theorem [5][1]). Suppose F is an endorelator on a locally-complete, tabular allegory $\mathcal{A}$. Let F' denote the endofunctor obtained by restricting F to the objects and arrows of $Map(\mathcal{A})$. Then, (I, in) is an initial F'-algebra iff 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 (13) is the property that is most often understood as the "hylo theorem". Property (11) is a necessary prerequisite; essentially it states that catamorphisms are well-defined on relations given that they are well-defined on functions. Property (12) 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.

4.2 Reductivity

A discipline of programming should always provide the programmer with easy-to -use techniques for guaranteeing termination of programs. For datatype-generic programs this is provided by the theory of so-called "reductivity" [10, 11] . The

[1] The theorem proved in [5] is actually about final algebras, rather than initial algebras. This is the harder theorem to prove because, in an allegory, the duality between least and greatest fixed points breaks down, reasoning about greatest being generally harder than reasoning about least fixed points. The proofs given in [5] can all be dualised.

major innovatory aspect of this concept is that it is parameterised by a relator, making it possible to explore how properties of termination are induced by properties of datatypes and (natural) transformations between datatypes.

A hylo program, $X = R \cdot F.X \cdot S$, is executed by first unfolding the equation and then computing the argument for the recursive call by executing S. This procedure is repeated until a base case is reached and no further unfoldings are necessary. Then the output is computed by executing R as often as the equation was unfolded. Assuming R and S are both guaranteed to terminate, termination of the recursion is thus dependent only on S, and not on R. Furthermore, if S is nondeterministic, a demonic semantics demands termination irrespective of which output from the unfoldings of S is chosen. This is the familiar execution scheme applied by the implementations of imperative, logical and functional languages. Because of this execution scheme, the computed input-output relation is the least solution of the hylo program.

Suppose that execution begins in a state described by the coreflexive A, and suppose B describes the "safe set" of the hylo program: the maximal set of states from which execution is guaranteed to terminate. Then, execution of S must guarantee that recursive calls begin from a state in B. That is, $(S \cdot A){<} \subseteq F.B$, or, equally, $A \subseteq S \backslash F.B$. Since B is the maximal set of such states, A, and since the semantics defines the input-output relation to be the least solution of the hylo equation, the safe set of program $X = R \cdot F.X \cdot S$ is the coreflexive $\langle \mu A :: S \backslash F.A \rangle$. Termination is guaranteed if this is the identity relation on the domain of S. Hence, the definition of reductivity:

Definition 3 (F-reductivity). Relation S of type $F.I \leftarrow I$ is said to be F-*reductive* if and only if $\langle \mu A :: S \backslash F.A \rangle \ = \mathsf{id}_I$. $\square$

Alternative characterisations of F-reductivity are sometimes more convenient. The following theorem gives three different ways to express F-reductivity. The first is the one already given; it is the most compact, and the most suited to abstract reasoning about the notion. The second form, 2(b), is closest to the way proof by induction is normally presented. The third alternative, 2(c), is formally weaker than the other two; hence, it is often useful to *prove* that a given relation is F-reductive.

Theorem 2 (Characterisations). The following are equivalent characterisations of the F-reductivity of relation S of type $F.I \leftarrow I$.

(a) $\langle \mu A :: S \backslash F.A \rangle \ = \ \mathsf{id}_I$
(b) $\langle \forall A :: S \backslash F.A \subseteq A \ \Rightarrow \ \mathsf{id}_I \subseteq A \rangle$
(c) $\langle \forall A :: S \backslash F.A = A \ \Rightarrow \ S{>} \subseteq A \rangle$

(In each case, the dummy A ranges over coreflexives under I.)

Proof The proof is by cyclic implication. That (a) implies (b) is an immediate consequence of the Knaster-Tarski fixed-point theorem. That (b) implies (c) is also easy: by reflexivity of $\subseteq$ and the fact that $S{>} \subseteq \mathsf{id}_I$ The more difficult step is that from (c) to (a); we show the (formally) stronger: clause (c) implies that *every* fixed point of the function $\langle A :: S\backslash F.A\rangle$ is id_I.

Assume A is a coreflexive under I such that $S\backslash F.A = A$. Also, assume (c). Then

$$
\begin{array}{ll}
 & \mathsf{id}_I = A \\
= & \quad \{ \quad A \subseteq \mathsf{id}_I,\ \text{assumption: } S\backslash F.A = A \quad \} \\
 & \mathsf{id}_I \subseteq S\backslash F.A \\
= & \quad \{ \quad \text{factors: (2)} \quad \} \\
 & S{\scriptstyle <} \subseteq F.A \\
= & \quad \{ \quad S = S \circ S{\scriptstyle >} \quad \} \\
 & (S \circ S{\scriptstyle >}){\scriptstyle <} \subseteq F.A \\
= & \quad \{ \quad \text{factors: (2)} \quad \} \\
 & S{\scriptstyle >} \subseteq S\backslash F.A \\
= & \quad \{ \quad \text{assumption: } S\backslash F.A = A \quad \} \\
 & S{\scriptstyle >} \subseteq A \\
\Leftarrow & \quad \{ \quad \text{(c)} \quad \} \\
 & S\backslash F.A = A \\
= & \quad \{ \quad \text{assumption: } S\backslash F.A = A \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

□

Let us now check that the notion of F-reductivity is compatible with more familiar accounts of program termination.

A programmer proves termination by using well-founded relations: they prove that the argument of every recursive call is "smaller" than the original argument. For program $X = R \cdot F.X \cdot S$ this means that all values stored in an output F-structure of S have to be smaller than the corresponding input of S. 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

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

for some well-founded ordering $\prec$. That is, a relation S is F-reductive if and only if there is a well-founded relation $\prec$ such that whenever an F-structure is related by S 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 [20, 19] 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 :

Definition 4 (Membership). Relation mem of type $I \leftarrow F.I$ is a *membership relation* of relator F iff $F.A = \mathsf{mem}\mathbin{\backslash\!\!\!\downharpoonright} A$, for all coreflexives A under I. $\Box$

Using this definition of membership we get a precise relationship between reductivity and well-foundedness. Indeed, for coalgebra S of type $F.I \leftarrow I$ and coreflexive A under I, we have:

$$\begin{array}{ll}
 & S \mathbin{\backslash\!\!\!\downharpoonright} F.A \\
= & \quad\{ \quad \text{definition 4} \quad \} \\
 & S \mathbin{\backslash\!\!\!\downharpoonright} (\mathsf{mem} \mathbin{\backslash\!\!\!\downharpoonright} A) \\
= & \quad\{ \quad \text{factors (2)} \quad \} \\
 & (\mathsf{mem} \cdot S) \mathbin{\backslash\!\!\!\downharpoonright} A \quad .
\end{array}$$

Now, well-foundedness of relation R of type $I \leftarrow I$ is the condition that the least prefix point of the function $\langle A :: R \mathbin{\backslash\!\!\!\downharpoonright} A\rangle$ is id_I [9], whereas reductivity of S of type $F.I \leftarrow I$ is the condition that the least prefix point of the function $\langle A :: S \mathbin{\backslash\!\!\!\downharpoonright} F.A\rangle$ is id_I. So, for coalgebra $S \;::\; F.I \leftarrow I$, the statement that S is F-reductive is equivalent to the statement that $\mathsf{mem} \cdot S$ is well-founded. Formally,

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

Conversely,

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

Summarising, we have:

Theorem 3. Suppose mem is the membership relation for relator F. Then the functions $\langle S :: \mathsf{mem} \cdot S\rangle$ and $\langle R :: \mathsf{mem} \backslash R\rangle$ form a Galois connection between the F-reductive relations, S, and the well-founded relations, R. $\Box$

Bird and De Moor [7, chapter 6] avoid the introduction of the notion of reductivity by always requiring that $\mathsf{mem} \cdot S$ is well-founded whenever F-reductivity of S is required. 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. The rules presented in section 5 are of this nature.

5 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.

5.1 Basic F-reductive relations

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

Theorem 4. The converse of an initial F-algebra is F-reductive.

Proof Let in of type $I \leftarrow F.I$ be an initial F-algebra and A an arbitrary coreflexive under I. Using theorem 2, it suffices to show that

$$\mathsf{id}_I \subseteq A \quad \Leftarrow \quad \mathsf{in}^{\cup} \searrow F.A \;\subseteq\; A \enspace .$$

We start with the antecedent and derive the consequent:

$$\begin{array}{ll}
 & \mathsf{in}^{\cup} \searrow F.A \;\subseteq\; A \\
= & \qquad \{ \qquad \text{for function } f \text{ and coreflexive } B,\ f \searrow B = f^{\cup} \cdot B \cdot f, \\
 & \qquad\qquad\quad \mathsf{in}^{\cup} \text{ is a function and } F.A \text{ is coreflexive} \quad \} \\
 & \mathsf{in} \cdot F.A \cdot \mathsf{in}^{\cup} \subseteq A \\
\Rightarrow & \qquad \{ \qquad (13),\ \text{fixed-point calculus} \quad \} \\
 & (\!|\mathsf{in}|\!) \cdot (\!|\mathsf{in}|\!)^{\cup} \;\subseteq\; A \\
= & \qquad \{ \qquad \text{identity rule: } (12),\ \mathsf{id}_I = \mathsf{id}_I{}^{\cup} = \mathsf{id}_I \cdot \mathsf{id}_I \quad \} \\
 & \mathsf{id}_I \;\subseteq\; A \enspace .
\end{array}$$

□

An immediate corollary of theorem 4 is that the cata program

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

is terminating. Also, by theorem 14 which we prove later, the solution of the equation is unique for all relations 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:

(14) $\mathsf{slsrt} = \mathsf{in} \cdot \mathsf{id}_{\mathbb{1}} + \mathsf{id}_I \times \mathsf{slsrt} \cdot \mathsf{in}^{\cup} \cdot \mathsf{select} \quad .$

Relation select holds between two lists if both are the empty list, or both are non-empty and 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. Relation in here is an initial $(\mathbb{1}+I\times)$-algebra. The program is interpreted as follows: 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 ($\mathsf{in}^{\cup}$), the tail is sorted recursively ($\mathsf{id}_I \times \mathsf{slsrt}$), finally the head, i.e. the minimum of the input, is added to the result of the recursive call (in).

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. Suppose $\oplus$ is a binary relator. Suppose also that there are mappings T from objects to objects and in from objects to arrows such that, for each I, in_I is an initial $(I\oplus)$-algebra of type $T.I \leftarrow I \oplus T.I$. Then the function mapping R of type $I {\leftarrow} J$ to the $(J\oplus)$ catamorphism $(\!|\mathsf{in}_I \cdot R \oplus \mathsf{id}_{T.I}|\!)$ extends the mapping T to a mapping on objects and arrows having the properties of a relator. The relator T is often called a *tree relator* [6, 7].

Theorem 5. 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 $\langle \mu A :: (\mathsf{in}^{\cup} \cdot T.\top\!\!\top) \backslash (\mathsf{id}_I \oplus A)\rangle$. Then

$\mathsf{in}^{\cup} \cdot T.\top\!\!\top \quad$ is $(I\oplus)$-reductive

$= \qquad \{ \quad \mathsf{in}^{\cup} \cdot T.\top\!\!\top$ has type $I \oplus T.I \leftarrow T.I$, definition 3 $\}$

$\mathsf{id}_{T.I} \subseteq B$

$\Leftarrow \qquad \{ \quad \mathsf{in}^{\cup}$ is $(I\oplus)$-reductive. $\}$

$\mathsf{in}^{\cup} \backslash (\mathsf{id}_I \oplus B) \subseteq B$

$= \qquad \{ \quad$ by the rolling rule (see eg [27]) and definition of B,

$\qquad\qquad \mathsf{in}^{\cup} \backslash (\mathsf{id}_I \oplus B) = \langle \mu A :: \mathsf{in}^{\cup} \backslash (\mathsf{id}_I \oplus (T.\top\!\!\top \backslash A))\rangle \quad \}$

$\langle \mu A :: \mathsf{in}^{\cup} \backslash (\mathsf{id}_I \oplus (T.\top\!\!\top \backslash A))\rangle \;\subseteq\; \langle \mu A :: (\mathsf{in}^{\cup} \cdot T.\top\!\!\top) \backslash (\mathsf{id}_I \oplus A)\rangle$

$\Leftarrow \qquad \{ \quad$ for all A, $\mathsf{in}^{\cup} \backslash (\mathsf{id}_I \oplus (T.\top\!\!\top \backslash A)) \subseteq (\mathsf{in}^{\cup} \cdot T.\top\!\!\top) \backslash (\mathsf{id}_I \oplus A)$

$\qquad\qquad$ (for proof, see below) monotonicity of $\mu \quad \}$

$\mathsf{true} \quad .$

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

$(\mathsf{in}^{\cup} \cdot T.\top\!\!\top) \backslash (\mathsf{id}_I \oplus A)$

$$
\begin{array}{lll}
= & \{ & T.\top\!\!\top \;=\; (\!|\mathsf{in}\cdot\top\!\!\top\oplus\mathsf{id}_{T.I}|\!) \quad , \\
 & & \text{and}\quad \top\!\!\top_{I\leftarrow I} = (\top\!\!\top_{I\leftarrow I})^{\cup}\;. \\
 & & \text{Thus}\quad \mathsf{in}^{\cup}\cdot T.\top\!\!\top \;=\; \top\!\!\top\oplus T.\top\!\!\top\cdot\mathsf{in}^{\cup} \quad \} \\
 & \multicolumn{2}{l}{(\top\!\!\top\oplus T.\top\!\!\top\cdot\mathsf{in}^{\cup})\searrow(\mathsf{id}_I\oplus A)} \\
= & \{ & \text{factors: (2)} \quad \} \\
 & \multicolumn{2}{l}{\mathsf{in}^{\cup}\searrow((\top\!\!\top\oplus T.\top\!\!\top)\searrow(\mathsf{id}_I\oplus A))} \\
\supseteq & \{ & \text{relators distribute over coreflexive factors;} \\
 & & \text{this also holds for binary relators} \quad \} \\
 & \multicolumn{2}{l}{\mathsf{in}^{\cup}\searrow((\top\!\!\top\searrow\mathsf{id}_I)\oplus(T.\top\!\!\top\searrow A))} \\
= & \{ & R\searrow\mathsf{id}_I = \mathsf{id}_I \text{ (for all } R \text{ of type } I\leftarrow I\text{); } R := \top\!\!\top_{I\leftarrow I} \quad \} \\
 & \multicolumn{2}{l}{\mathsf{in}^{\cup}\searrow(\mathsf{id}_I\oplus(T.\top\!\!\top\searrow A)) \quad .}
\end{array}
$$

□

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 6 is therefore that reductivity is preserved under refinement.

Theorem 6. If R is F-reductive and $S\subseteq R$ then S is F-reductive.

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

Now we can return to the proof of termination of selection sort. We have that $\mathsf{select}\subseteq\mathsf{List}.\top\!\!\top$ since select is a relation on equal-length lists. By theorem 6, relation $\mathsf{in}^{\cup}\cdot\mathsf{select}$ is $(\mathbb{1}+I\times)$-reductive if $\mathsf{in}^{\cup}\cdot\mathsf{List}.\top\!\!\top$ is, which is a consequence of theorem 5 obtained by taking List for map, and $\mathsf{id}_{\mathbb{1}}+(R\times S) = R\oplus S$.

5.2 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 [13]. The square rule can clearly be extended to an nth power rule. The corresponding reductivity lemma is the following:

Theorem 7 (Power Rule). Suppose R 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

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

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

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

$=$ { definition of g^{n+1} }

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

$\subseteq$ { induction hypothesis, monotonicity of $R\backslash$ and F }

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

$\subseteq$ { factors: (2), F distributes through composition }

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

$=$ { factors: (2) }

$$(F.(f.n) \cdot R) \backslash F.(F^n.A)$$

$=$ { definition }

$$f.(n{+}1) \backslash F^{n+1}.A \quad .$$

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

$f.n$ is F^n-reductive

$=$ { definition }

$$\langle \mu A :: f.n \backslash F^n.A \rangle \;\supseteq\; \mathsf{id}_I$$

$\Leftarrow$ { above, monotonicity of the fixed-point operator }

$$\langle \mu A :: (R\backslash \circ F)^n.A \rangle \;\supseteq\; \mathsf{id}_I$$

$\Leftarrow$ { $\mu(g^n) \supseteq \mu g$ }

$$\langle \mu A :: (R\backslash \circ F).A \rangle \;\supseteq\; \mathsf{id}_I$$

$=$ { definition of composition and reductivity }

R is F-reductive .

□

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 transforms G-structures into F-structures without affecting the contents of the structures —the only thing that can happen is that elements are copied or discarded— results in a reductive relation. In order to state the theorem precisely we need to formalise what is often loosely described as "plumbing".

Definition 5. Relation R is a *plumbing* to relator F from relator G, written $R : F \stackrel{.}{\leftsquigarrow} G$, iff R has type $F.I \leftarrow G.I$, for some I, and for all coreflexives A under I:

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

□

Natural transformations are families of plumbing relations:

Theorem 8. 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 under I. Then

$$\begin{array}{ll}
 & G.A \subseteq \alpha_I \backslash F.A \\
= & \qquad \{ \qquad \text{factors: (2)} \quad \} \\
 & (\alpha_I \cdot G.A)^{<} \subseteq F.A \\
= & \qquad \{ \qquad \text{domains} \quad \} \\
 & F.A \cdot \alpha_I \cdot G.A = \alpha_I \cdot G.A \\
= & \qquad \{ \qquad \alpha : F \hookleftarrow G.\ \text{Thus, } F.A \cdot \alpha_I \;\supseteq\; \alpha_I \cdot G.A. \\
 & \qquad \qquad \quad G.A \cdot G.A = G.A \quad \} \\
 & F.A \cdot \alpha_I \cdot G.A \;\subseteq\; \alpha_I \cdot G.A \\
= & \qquad \{ \qquad F.A \subseteq \mathsf{id}_{F.I} \quad \} \\
 & \mathsf{true} \quad .
\end{array}$$

□

We can now formulate our theorem.

Theorem 9. Let Q be G-reductive and $S : F \stackrel{.}{\leftsquigarrow} \mathsf{Id}$, where Id denotes the identity relator. Then $F.Q \cdot S$ is $(F{\circ}G)$-reductive.

Proof We prove the stronger:

$$\langle \mu A :: Q \backslash G.A \rangle \;\subseteq\; \langle \mu A :: (F.Q \cdot S) \backslash F.(G.A) \rangle \quad .$$

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

$$\begin{array}{ll}
 & (F.Q \cdot S) \backslash F.(G.A) \\
= & \qquad \{ \qquad \text{factors: (2)} \quad \} \\
 & S \backslash (F.Q \backslash F.(G.A)) \\
\supseteq & \qquad \{ \qquad \text{factors: (2)} \quad \} \\
 & S \backslash F.(Q \backslash G.A) \\
\supseteq & \qquad \{ \qquad S : F \stackrel{.}{\leftsquigarrow} \mathsf{Id} \quad \} \\
 & Q \backslash G.A \quad .
\end{array}$$

□

A typical use of theorems (6) and (9) is: R is F-reductive follows from the fact that there is a well-founded relation Q and a relation $S : F \stackrel{\cdot}{\leftsquigarrow} \mathsf{Id}$ 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 which relates a number x to a list of arbitrary length containing only copies of x. This relation certainly has the property $\mathsf{fan} \cdot A \subseteq \mathsf{List}.A \cdot \mathsf{fan}$ for all A: 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 9, it follows that R is List-reductive.

This argument is, in fact, an instance of the generic discussion of membership in section 4.2. Associated with each container type F there is a family of fan relations such that fan_I has type $F.I \leftarrow I$. 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 of type $I \leftarrow I$, 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 [20, 19] for further details. Thus, by applying theorem 3, $F.R \cdot \mathsf{fan}_I$ is F-reductive if R is well-founded.

An important and commonly occurring pattern in program construction is structural recursion on just one of possibly several input parameters of a program. The abstract theorem that captures the termination properties of such programs is the following.

Theorem 10. Suppose R is F-reductive, and suppose S is such that $S : H{\circ}G \stackrel{\cdot}{\leftsquigarrow} G{\circ}F$, where G is 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 $G.\mathsf{id}_I \subseteq \langle \mu A :: (S \cdot G.R) \backslash H.A\rangle$, assuming that R is F-reductive. We prove the stronger: for all F-coalgebras R

(15) $G.\langle \mu A :: R \backslash F.A\rangle \;\subseteq\; \langle \mu A :: (S \cdot G.R) \backslash H.A\rangle$.

The theorem then follows from the assumed F-reductivity of R. Because G is a lower adjoint in a Galois connection, property (15) follows by fixpoint fusion [27] from the fact that, for all A,

$$\begin{array}{ll}
 & (S \cdot G.R) \backslash H.(G.A) \\
= & \quad \{ \quad \text{factors: (4)} \quad \} \\
 & G.R \backslash (S \backslash H.(G.A)) \\
\supseteq & \quad \{ \quad S : H{\circ}G \stackrel{\cdot}{\leftsquigarrow} G{\circ}F, \text{ monotonicity} \quad \} \\
 & G.R \backslash G.(F.A)
\end{array}$$

$$\supseteq \qquad \{ \qquad \text{factors: (2), } G \text{ is a relator}^2 \quad \}$$

$$G.(R \setminus F.A) \quad .$$

□

The restriction on relator G in this theorem is satisfied by the sections $(J\times)$ and $(\times J)$ of the product relator. It is this instantiation of G that allows one to prove termination of programs with several parameters that are defined by structural recursion on one of the parameters.

There are 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

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

where X is the function being defined and f and g are known functions. In hylomorphism form,

$$X \;=\; \mathsf{comb} \cdot (\mathsf{id}_{\mathbb{1}} + X) \times \mathsf{id} \cdot \mathsf{pass} \cdot \mathsf{inNat}^{\cup} \times \mathsf{id}_J \quad .$$

Here inNat is the initial algebra with carrier the natural numbers. The function pass is a function of type $(\mathbb{1} + (I{\times}J)) \times J \leftarrow (\mathbb{1} + I) \times J$ that is polymorphic in the types I and J; its task is to make a copy of its second argument (of type J), which is passed to the recursive call. The function comb is a combination of the functions f and g which is applied to the result of the recursive call and the "passed" second argument.

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 $+\!\!\!+$), the definition has the form:

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

Here inList is the initial algebra with carrier lists, and pass is a function of type

$$(\mathbb{1} + (I{\times}(J{\times}K))) \times K \;\leftarrow\; (\mathbb{1} + (I{\times}J)) \times K$$

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:

$$(16)\quad X \;=\; R \cdot F.X \times \mathsf{id}_P \cdot F.(\mathsf{id}_I{\times}S) \times \mathsf{id}_P \cdot \mathsf{pass} \cdot \mathsf{in}^{\cup} \times \mathsf{id}_P \quad .$$

Here id_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 $F.(I{\times}P) \times P \leftarrow F.I \times P$. Its function is to copy the parameter, and at the same time pass it to all values stored in an F-structure. (The latter is also called a "broadcast" [19] or a "strength" [31].) 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 in such a way that, for all S,

$$F.(\mathsf{id}_P{\times}S) \times \mathsf{id}_P \cdot \mathsf{pass}$$

is a plumbing relation with type

$$({\times}P) \circ F \circ ({\times}P) \;\dot{\leadsto}\; ({\times}P) \circ F \quad .$$

Furthermore, $({\times}P)$ is a relator which distributes over all unions of coreflexives. By theorem 10, it now follows that

$$F.(\mathsf{id}_P{\times}S) \times \mathsf{id}_P \cdot \mathsf{pass} \cdot \mathsf{in}^{\cup} \times \mathsf{id}_P$$

is a $(({\times}P) \circ F)$-reductive relation. Hence, program (16) 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.

From theorem 10, the next theorem follows as a simple corollary.

Theorem 11. If R is F-reductive and $S : H \;\dot{\leadsto}\; F$ then $S \cdot R$ is H-reductive.

Proof Instantiate theorem 10 with the identity relator (which, of course, is the lower adjoint in a Galois connection with itself as upper adjoint).
□

Two datatype-generic applications of theorem 11 are to so-called *paramorphisms* and *mutumorphisms*.

Paramorphisms were introduced by Meertens [28] as a (datatype-generic) abstraction of the "eliminators" in intuitionistic type theory. The general form of a paramorphism is a solution of the equation

$$X \;::\; X = R \cdot F.(\mathsf{id}_I{\times}X) \cdot F.\mathsf{double} \cdot \mathsf{in}^{\cup}$$

where $\mathsf{double}.x = (x,x)$ and R is an arbitrary relation (of the appropriate type). Applying 11, it is straightforward to show that execution of a para program always terminates. Specifically, relation $\mathsf{in}^{\cup}$ is F-reductive. Furthermore, we have (for all coreflexives A under I)

$$F.A \;\subseteq\; F.\mathsf{double} \backslash F.(\mathsf{id}_I {\times} A)$$

since

$$A \;\subseteq\; \mathsf{double} \backslash (\mathsf{id}_I {\times} A)$$

and relators distribute through composition and are monotonic. This means that relation $F.\mathsf{double}$ is a plumbing relation of type $F \circ (I{\times}) \overset{\cdot}{\leftsquigarrow} F$. It now follows by corollary 11 that $F.\mathsf{double} \cdot \mathsf{in}^{\cup}$ is an $(F \circ (I{\times}))$-reductive relation. Hence, the para program is terminating (and has, by theorem 14 proved later, a unique solution).

Mutumorphisms were introduced by Fokkinga [12] as an abstraction of mutual recursion. A mutu program is defined by an equation of the form:

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

The proof that such programs are terminating is similar to the proof for paramorphims. One needs to check that double is of type $G \overset{\cdot}{\leftsquigarrow} F$, where relator G is defined by $G.Y = F.Y \times F.Y$, i.e. one has to show that

$$F.A \subseteq \mathsf{double} \backslash (F.A \times F.A)$$

which follows immediately from the definitions of double and the product relator.

5.3 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 12. Let R of type $F.I \leftarrow I$ be an F-reductive relation. Suppose f, of type $I \leftarrow J$, is a single-valued relation. Then $F.f^{\cup} \cdot R \cdot f$ is F-reductive.

Proof With dummy A ranging over coreflexives under J, we have:

$$\langle \mu A :: (F.f^{\cup} \cdot R \cdot f) \backslash F.A \rangle$$

$$= \qquad \{ \qquad \text{factors: (2)} \quad \}$$

$$\begin{array}{ll}
 & \langle \mu A :: f \backslash ((F.f^{\cup} \cdot R) \backslash F.A)\rangle \\
= & \{ \quad \text{rolling rule (see eg [27])} \quad \} \\
 & f \backslash \langle \mu A :: (F.f^{\cup} \cdot R) \backslash F.(f \backslash A)\rangle \\
= & \{ \quad \text{factors: (2)} \quad \} \\
 & f \backslash \langle \mu A :: R \backslash F.f^{\cup} \backslash F.(f \backslash A)\rangle \\
\supseteq & \{ \quad \text{factors: (2), } F \text{ is a relator, monotonicity} \quad \} \\
 & f \backslash \langle \mu A :: R \backslash F.(f^{\cup} \backslash (f \backslash A))\rangle \\
= & \{ \quad \text{factors: (2)} \quad \} \\
 & f \backslash \langle \mu A :: R \backslash F.((f \cdot f^{\cup}) \backslash A)\rangle \\
\supseteq & \{ \quad f \cdot f^{\cup} \subseteq \mathsf{id}_I \text{, antimonotonicity of } \backslash \text{,} \\
 & \quad\;\; \text{monotonicity of the other operators} \quad \} \\
 & f \backslash \langle \mu A :: R \backslash F.A\rangle \quad .
\end{array}$$

So, if R is F-reductive, $\langle \mu A :: (F.f^{\cup} \cdot R \cdot f) \backslash F.A\rangle \;\supseteq\; f \backslash \mathsf{id}_I$. The result follows from the fact that $S \backslash \mathsf{id}_I$ equals id_J for all S of type $I \leftarrow J$.
□

It now follows by theorem 6 that, if R and f satisfy the conditions of theorem 12, 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 13. 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. □

6 Connections to other concepts

The notion of F-reductivity is original 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 iff 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 6.1 where it is shown that every F-reductive relation is F-well-founded. It is shown, however, that not every F-well-founded relation is F-reductive.

We also explore in section 6.2 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.

6.1 Well-foundedness generalised

In general, a relation on some state space is well-founded iff it admits induction. Point-free formulations of these concepts have been given in [9]. 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) [9]. This is easily generalised to the property that the equation $X{::}\ X = S \cdot X \cdot R$ has a unique solution, for all relations S. The generic notion of well-foundedness we propose focuses on this unicity of the solution of equations.

Definition 6 (F-well-founded). Relation R of type $F.I \leftarrow I$ is *F-well-founded* iff, for all relations S of type $I \leftarrow F.I$ and X of type $I \leftarrow I$,

$$X = S \cdot F.X \cdot R \quad \equiv \quad X = \langle \mu Y :: S \cdot F.Y \cdot R \rangle \quad .$$

□

As mentioned above, a relation is Id-well-founded iff it is well-founded in the traditional sense [8]. 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 of type $F.I \leftarrow I$ is an F-reductive relation then, for any relation S of type $I \leftarrow F.I$, the function $\langle Y :: S \cdot F.Y \cdot R \rangle$ 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 $\langle Y :: S \cdot F.Y \cdot R \rangle$. We have to show that $X \subseteq Z$ under the assumption that R is F-reductive.

$$\begin{array}{ll}
 & X \subseteq Z \\
= & \quad \{ \quad R \text{ is } F\text{-reductive, i.e } \langle \mu A :: R \setminus F.A \rangle = \mathsf{id}_I \quad \} \\
 & X \cdot \langle \mu A :: R \setminus F.A \rangle \ \subseteq\ Z \\
\Leftarrow & \quad \{ \quad \mu\text{-fusion (see eg [27]); } Z \text{ is least fixed point} \quad \} \\
 & \langle \forall A \ :: \ X \cdot R \setminus F.A \ \subseteq\ S \cdot F.(X \cdot A) \cdot R \rangle \\
= & \quad \{ \quad X \text{ is a fixed point; } F \text{ distributes over composition} \quad \} \\
 & \langle \forall A \ :: \ S \cdot F.X \cdot R \cdot R \setminus F.A \ \subseteq\ S \cdot F.X \cdot F.A \cdot R \rangle \\
\Leftarrow & \quad \{ \quad \text{monotonicity} \quad \} \\
 & \langle \forall A \ :: \ R \cdot R \setminus F.A \ \subseteq\ F.A \cdot R \rangle \\
= & \quad \{ \quad \text{cancellation of factors} \quad \} \\
 & \mathsf{true} \quad .
\end{array}$$

This completes the proof of the following theorem.

Theorem 14. An F-reductive relation is F-well-founded. $\Box$

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. Indeed, suppose we define the relator F by $F.X = X\times X$. Then, if R is a non-empty Id-well-founded relation of type $I\leftarrow I$, the relation $\mathsf{id}_I\vartriangle R$ of type $I\times I\leftarrow I$, which (non-deterministically) maps argument x into a pair (x,y) where y stands in the relation R to x, is F-well-founded but not F-reductive. Informally, execution of the hylo program $X = S\cdot X\times X\cdot \mathsf{id}_I\vartriangle R$ will not terminate because of the (demonicly chosen) infinite recursion on the copy of the input parameter. However, the equation has exactly one solution because R is well-founded. See [8] for a detailed proof.

Because an F-reductive relation is also F-well-founded, a terminating hylo-equation has a unique solution (i.e. defines a unique input-output relation).

Theorem 15. If R is F-reductive, the hylo equation $X = S\cdot F.X\cdot R$ has a unique solution.

Proof Combine theorem 14 and definition 6.
$\Box$

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

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

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

$$\mathsf{Stree} \quad = \quad \mathbb{1} + (A\times\mathsf{Stree}\times B\times\mathsf{Stree}) + (B\times\mathsf{Stree}\times A\times\mathsf{Stree}) \quad .$$

Here, $A=\{a\}$ and $B=\{b\}$. It is an initial F-algebra where the relator F maps X to $\mathbb{1} + (A\times X\times B\times X) + (B\times X\times A\times X)$. 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 an F-catamorphism $(\!|\mathsf{unp}|\!)$ (where the details of unp need not concern us). 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

$$S \quad = \quad (\!|\mathsf{unp}|\!)\cdot(\!|\mathsf{unp}|\!)^{\cup} \quad .$$

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, we also have the hylo program

$$S \quad = \quad \mathsf{unp} \;\cdot\; \mathsf{id}_{\mathbb{1}} + (\mathsf{id}_A\times S\times\mathsf{id}_B\times S) + (\mathsf{id}_B\times S\times\mathsf{id}_A\times S) \;\cdot\; \mathsf{unp}^{\cup} \quad .$$

This is a program that works by (nondeterministically) choosing to check whether the word is the empty word, or can be split into four segments either of the form $aXbY$ (i.e. a followed by a word X followed by b followed by a word Y) or the form $bXaY$. Subsequently any segments so constructed are recombined into one.

The hylo program corresponding to this grammar is clearly terminating. Formally, this is a consequence of theorem 12: 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

$$(17) \quad X{::} \quad X \;=\; \{\varepsilon\} \cup \{a\}X\{b\}X \cup \{b\}X\{a\}X \quad .$$

The language generated by this grammar is in fact the set of all words with an equal number of as and bs. 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 (17), 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 (17). Since (17) has unique solution $L.S$ it follows that M equals $L.S$.

Now consider the grammar

$$S \quad ::= \quad \varepsilon \;|\; aSb \;|\; bSa \;|\; SS \quad .$$

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.

6.2 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 inNat. 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 the successor function, succ, if 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 whenever

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

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

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

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

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

□

There is another way of justifying the definition of inductivity which we will just sketch. Recall that termination of a hylo program 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 iff 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 16. Let R be an F-reductive relation such that $R{<} \subseteq F.R{>}$. Then $R^{\cup}$ is F-inductive.

Proof Suppose R has type $F.I \leftarrow I$. Then, for all coreflexives A under I,

$$\begin{array}{cl}
 & (R^{\cup}){<} \subseteq A \\
= & \qquad \{ \qquad X{<} = (X^{\cup}){>} \quad \} \\
 & R{>} \subseteq A \\
= & \qquad \{ \qquad \text{factors: (2)} \quad \} \\
 & \mathsf{id}_I \subseteq R{>} \mathbin{\backslash\!\!\!\downharpoonleft} A \\
\Leftarrow & \qquad \{ \qquad R \text{ is } F\text{-reductive} \quad \} \\
 & R \mathbin{\backslash\!\!\!\downharpoonleft} F.(R{>} \mathbin{\backslash\!\!\!\downharpoonleft} A) \subseteq R{>} \mathbin{\backslash\!\!\!\downharpoonleft} A \\
\Leftarrow & \qquad \{ \qquad \text{factors (2), relators} \quad \} \\
 & R{>} \cdot R \mathbin{\backslash\!\!\!\downharpoonleft} (F.R{>} \mathbin{\backslash\!\!\!\downharpoonleft} F.A) \subseteq A \\
= & \qquad \{ \qquad \text{factors (2)} \quad \} \\
 & R{>} \cdot (F.R{>} \cdot R) \mathbin{\backslash\!\!\!\downharpoonleft} F.A \subseteq A \\
= & \qquad \{ \qquad \text{assumption: } R{<} \subseteq F.R{>} \quad \} \\
 & R{>} \cdot R \mathbin{\backslash\!\!\!\downharpoonleft} F.A \subseteq A \\
= & \qquad \{ \qquad \text{domains: } R{>} \cdot R \mathbin{\backslash\!\!\!\downharpoonleft} F.A = (R \cdot R \mathbin{\backslash\!\!\!\downharpoonleft} F.A){>} \quad \} \\
 & (R \cdot R \mathbin{\backslash\!\!\!\downharpoonleft} F.A){>} \subseteq A \\
\Leftarrow & \qquad \{ \qquad \text{factors: } R \cdot R \mathbin{\backslash\!\!\!\downharpoonleft} F.A \subseteq F.A \cdot R \quad \} \\
 & (F.A \cdot R){>} \subseteq A \\
= & \qquad \{ \qquad X{<} = (X^{\cup}){>} \quad \} \\
 & (R^{\cup} \cdot F.A){<} \subseteq A \quad .
\end{array}$$

$\square$

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 17. If R of type $I \leftarrow F.I$ is an injective F-inductive relation, then $R^{\cup}$ is F-reductive.

Proof We use characterisation (c) of F-reductivity given in theorem 2. Assume A is a coreflexive under I. Then

$$\begin{array}{lll}
 & (R^{\cup})_{>} \subseteq A & \\
\Leftarrow & \quad \{ \quad R \text{ is } F\text{-inductive, } (R^{\cup})_{>} = R_{<}\text{, definition 7} \quad \} & \\
 & (R \cdot F.A)_{<} \subseteq A & \\
= & \quad \{ \quad X_{<} = (X^{\cup})_{>}\text{; distribution properties of } ^{\cup} \quad \} & \\
 & (F.A \cdot R^{\cup})_{>} \subseteq A & \\
= & \quad \{ \quad \text{for single-valued } S \text{ and all coreflexives } A, & \\
 & \qquad\quad S_{>} \cdot S \setminus A \;=\; (A \cdot S)_{>} \;\; ; & \\
 & \qquad\quad R^{\cup} \text{ is single-valued (i.e. } R \text{ is injective)} \quad \} & \\
 & (R^{\cup})_{>} \cdot R^{\cup} \setminus F.A \;\subseteq\; A & \\
\Leftarrow & \quad \{ \quad (R^{\cup})_{>} \subseteq \mathsf{id}_{I} \quad \} & \\
 & R^{\cup} \setminus F.A \;\subseteq\; A & \\
\Leftarrow & \quad \{ \quad \text{reflexivity of } \subseteq \quad \} & \\
 & R^{\cup} \setminus F.A \;=\; A \;\; . &
\end{array}$$

□

To find a relation that is inductive but whose converse is not reductive we therefore have to look at non-injective inductive relations. To this end, consider the datatype $\mathsf{Join}.I$ of join lists with elements of type I. Let F be the relator that maps X to $(X{\times}X)+(I{+}\mathbb{1})$. Let join be the function that constructs a list of type $\mathsf{Join}.I$ by joining two lists of type $\mathsf{Join}.I$; let τ be the function that maps a value x of type I to the singleton list $[x]$ of type $\mathsf{Join}.I$, and let nill map the single element of the unit type $\mathbb{1}$ to the nil list of type $\mathsf{Join}.I$. Then $\mathsf{join} \triangledown (\tau \triangledown \mathsf{nill})$, the function that chooses to apply join, τ or nill depending on the type of its argument, is an initial F-algebra of type $\mathsf{Join}.I \leftarrow F.(\mathsf{Join}.I)$. That it is F-inductive is equivalent to the well-known induction rule on lists: consider three cases, the join of two lists, singleton lists and lists that are the empty list. However, its converse, $(\mathsf{join} \triangledown (\tau \triangledown \mathsf{nill}))^{\cup}$, is not reductive. This is because $(\mathsf{join} \triangledown (\tau \triangledown \mathsf{nill}))^{\cup}$ holds between a tagged pair of lists and their join. The tag inl injects the pair into the left component of the disjoint sum. Now, the relation $\mathsf{exl} \cdot \mathsf{inl}^{\cup}$ (where exl extracts the left component of a pair) is a natural transformation of type $\mathsf{Id} \hookleftarrow F$. So, if $(\mathsf{join} \triangledown (\tau \triangledown \mathsf{nill}))^{\cup}$ were F-reductive, the relation

$$\mathsf{exl}_{\mathsf{Join}.I \,\times\, \mathsf{Join}.I} \;\cdot\; (\mathsf{inl}_{(\mathsf{Join}.I \,\times\, \mathsf{Join}.I)\,+\,(I+\mathbb{1})})^{\cup} \;\cdot\; (\mathsf{join} \triangledown (\tau \triangledown \mathsf{nill}))^{\cup}$$

would be Id-reductive by corollary 11 — in other words, it would be well-founded. However, since the join of two empty lists is the empty list, this relation relates the empty list to the empty list, and so is not well-founded.

7 Generic Unification

In this section, we apply the notion of F-reductivity to a key lemma in the proof of correctness of a generic unification algorithm. Such an algorithm was first formulated by Jeuring and Jansson [21] and is further elaborated in [6]. The algorithm is "generic" in the sense that it is parameterised by a relator F that specifies the structure of expressions to be unified.

Here, 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.

(The reader is invited to compare the proof presented here with the one given in [6]. Although the one presented here was the first to be developed, it was considered expedient at the time not to burden the reader of [6] with too many new ideas, and to present a more conventional proof instead.)

In its generic form, unification is expressed as follows. 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 $F.X + \mathsf{id}_V$. Then we assume that in is an initial F_V-algebra with carrier $F^\star V$. That is, in has type

$$F^\star V \leftarrow F.F^\star V + V \quad .$$

The relator $F^\star$ (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 $F^\star V$ 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 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.

A fundamental lemma in a proof of correctness of unification 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 $\mathsf{inl}_{I,J}$ denote the injection function of type $I{+}J \leftarrow I$. (We will drop subscripts from now on for simplicity.) Then we can define the relation $\mathsf{occurs\_properly\_in}$ of type $F^\star V \leftarrow F^\star V$ 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 $F.(F^\star V) \leftarrow F^\star V$) destructs an element of $F^\star V$ into an F-structure and then mem identifies the data stored in that F-structure. Thus $\mathsf{mem}\cdot(\mathsf{in}{\cdot}\mathsf{inl})^{\cup}$ destructs an element of $F^{\star}V$ into a number of immediate subcomponents. Application of the transitive-closure operation repeats this process thus breaking the structure down into all its subcomponents.

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 reductivity calculus.

Theorem 18. The relation $\mathsf{occurs\_properly\_in}$ is well-founded.

Proof

$$
\begin{array}{ll}
 & \mathsf{occurs\_properly\_in} \text{ is well-founded} \\
= & \quad \{ \quad \text{definition of } \mathsf{occurs\_properly\_in}, \\
 & \quad\quad R \text{ is well-founded} \;\equiv\; R^{+} \text{ is well-founded} \quad \} \\
 & \mathsf{mem}\cdot(\mathsf{in}{\cdot}\mathsf{inl})^{\cup} \text{ is well-founded} \\
= & \quad \{ \quad \mathsf{mem}\cdot R \text{ is well-founded} \;\equiv\; R \text{ is } F\text{-reductive} \quad \} \\
 & (\mathsf{in}{\cdot}\mathsf{inl})^{\cup} \text{ is } F\text{-reductive} \\
= & \quad \{ \quad (\mathsf{in}{\cdot}\mathsf{inl})^{\cup} = \mathsf{inl}^{\cup}\cdot\mathsf{in}^{\cup}, \quad \} \\
 & \mathsf{inl}^{\cup}\cdot\mathsf{in}^{\cup} \text{ is } F\text{-reductive} \\
\Leftarrow & \quad \{ \quad \text{theorems 8 and 9} \quad \} \\
 & \mathsf{in}^{\cup} \text{ is } F_V\text{-reductive} \;\wedge\; \mathsf{inl}^{\cup} : F \hookleftarrow F_V \\
\Leftarrow & \quad \{ \quad \text{theorem 4, definition of } \hookleftarrow \\
 & \quad\quad (\text{where } R \text{ has type } I{\leftarrow}J, \text{ for some } I \text{ and } J) \quad \} \\
 & \mathsf{true} \;\wedge\; \langle\forall R :: F.R\cdot\mathsf{inl}^{\cup} \supseteq \mathsf{inl}^{\cup}\cdot F_V.R\rangle \\
= & \quad \{ \quad \text{definition of } F_V \quad \} \\
 & \langle\forall R :: F.R\cdot\mathsf{inl}^{\cup} \supseteq \mathsf{inl}^{\cup}\cdot F.R + \mathsf{id}_V\rangle \\
= & \quad \{ \quad \text{converse and defn. of } \mathsf{inl} \quad \} \\
 & \mathsf{true} \;.
\end{array}
$$

□

Note that the proof is entirely algebraic and 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 F-algebra is F-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.

8 Conclusion

This paper has demonstrated how to reason effectively about computations where the structure of the data is a parameter — so-called datatype-generic reasoning. 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. Evidence has been provided for why program development should be based on relation algebra, even when the desired implementation vehicle is a functional programming language.

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 5 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 5 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 worthwhile.

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 5 and could be useful in gaining a better understanding of the design of logic programs and distributed programs.

References

1. R.C. Backhouse. Naturality of homomorphisms. Lecture notes, International Summer School on Constructive Algorithmics, vol. 3, 1989.
2. 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.
3. 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.

4. R.C. Backhouse and J. van der Woude. Demonic operators and monotype factors. *Mathematical Structures in Computer Science*, 3(4):417–433, December 1993.
5. Roland Backhouse and Paul Hoogendijk. Final dialgebras: From categories to allegories. *Theoretical Informatics and Applications*, 33(4/5):401–426, 1999.
6. 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.
7. Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall International, 1996.
8. H. Doornbos. *Reductivity arguments and program construction*. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, June 1996.
9. H. Doornbos, R.C. Backhouse, and J. van der Woude. A calculational approach to mathematical induction. *Theoretical Computer Science*, 179:103–135, 1997.
10. 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.
11. Henk Doornbos and Roland Backhouse. Reductivity. *Science of Computer Programming*, 26(1–3):217–236, 1996.
12. Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, The Netherlands, 1992.
13. 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.
14. P.J. Freyd and A. Ščedrov. *Categories, Allegories*. North-Holland, 1990.
15. Jeremy Gibbons. Patterns in datatype-generic programming. In Jörg Striegnitz and Kei Davis, editors, *Multiparadigm Programming*, volume 27. John von Neumann Institute for Computing (NIC), 2003. First International Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL).
16. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.
17. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, 2004.
18. C.A.R. Hoare and Jifeng He. The weakest prespecification. *Fundamenta Informaticae*, 9:51–84, 217–252, 1986.
19. Paul Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997.
20. Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.
21. 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.
22. P. Jansson and J. Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 1998.
23. J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Proceedings of the Second International Summer School on Advanced Functional Programming Techniques*, pages 68–114. Springer-Verlag, 1996. LNCS 1129.

24. Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Olin Shivers, editor, *Proceedings of the International Conference, ICFP'03*, pages 141–152. ACM Press, August 2003.
25. G. Malcolm. *Algebraic data types and program transformation*. PhD thesis, Groningen University, 1990.
26. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, October 1990.
27. Eindhoven University of Technology Mathematics of Program Construction Group. Fixed point calculus. *Information Processing Letters*, 53(3):131–136, February 1995.
28. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
29. 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.
30. Erik Meijer. *Calculating Compilers*. PhD thesis, University of Nijmegen, 1992.
31. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
32. J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E. Mason, editor, *IFIP '83*, pages 513–523. Elsevier Science Publishers, 1983.
33. P. Wadler. Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture, ACM, London*, September 1989.