Reductivity

Henk Doornbos and Roland Backhouse

Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands

The notion of reductivity of a relation with respect to a datatype is introduced and related to inductivity and initiality. The use of reductivity in constructing terminating programs is discussed. A calculus of reductivity, discussed in more detail in a companion paper, is introduced.

A well-established method of designing **while** statements involves a loop invariant coupled with a loop variant and a test for termination: the desired postcondition is decomposed into the invariant and the termination condition, and the loop body is designed to maintain the invariant whilst making progress towards the termination condition by ensuring that at each iteration the variant function is decreased.

In this paper we discuss how to extend the methodology of choosing variant functions so as to include the iteration structure as part and parcel of the methodology. That is, we discuss the design of not just **while** statements, where the iteration structure is linear, but also, for example, divide and conquer algorithms, where the iteration structure is non-linear. (Non-linear iteration schemes are commonly known as "recursion schemes" in the computing science literature. We use the words "iteration" and "recursion" synonymously.) The contribution of the paper is to formulate a notion of *reductivity*, generalising the notion of decreasing a variant function.

Our work is a contribution to the development of what has recently been dubbed "polytypic" programming. Polytypic programming is the methodology of designing programs that are parameterised by datatypes. The fundamental notions underlying the methodology, of which reductivity is one, are thus also so parameterised. Although the name "polytypic" programming has only recently come into being the principal ideas began to chrystallize around the mid to late eighties, and are evident in several published papers [7,12,8,2,5,9].

The current paper is complementary to the paper presented at the 1995 MPC conference [3]. The latter paper was primarily concerned with presenting a

Preprint submitted to Elsevier Preprint *6 October 1995*

reductivity calculus and not so much emphasis was placed on motivating the notion. In this paper the goal is primarily to motivate the notion in the context of program design and the reductivity calculus is only mentioned where it is needed to justify some claim.

We motivate the notion of reductivity by showing how it is a generalisation of the notion of "initiality" of an algebra. Indeed, in addition to reductivity we consider "inductivity" and "well-foundedness" as alternative generalisations of initiality and we argue why reductivity is the most appropriate to program construction. (The terms in inverted commas are all relative to a datatype and thus are themselves polytypic generalisations of more standard notions. Their precise definitions are given later.)

In order to make the paper accessible to as broad an audience as possible we introduce all the notions we need more or less from scratch even where these notions are "well-known" (to particular sections of the computing science community). In particular, we introduce the notions of "relator", "algebra" and "initial algebra". These are introduced with the aid of examples rather than with precise formal definitions. We begin with relators and algebras.

1 Relators and Algebras

1.1 Preliminaries

Before we can properly begin we need to introduce some notational conventions.

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

We use a minimum of notation. We write $x\langle R\rangle y$ if the pair (x,y) is an element of relation R. We use an infix dot to denote function application. Thus $f.x$ denotes application of function f to argument x. Functions are particular sorts of relations; a relation f is *functional* if $y\langle f\rangle x$ and $z\langle f\rangle x$ together imply that $y = z$. If this is the case we write $f.x$ for the unique y such that $y\langle f\rangle x$. We use a raised infix dot to denote relational composition.

Thus $R{\bullet}S$ denotes the composition of relations R and S (the relation defined by $x\langle R{\bullet}S\rangle z$ equivales $\exists(y{::}\ x\langle R\rangle y \wedge y\langle S\rangle z)$). The converse of relation R is denoted by $R^{\cup}$. Thus, $x\langle R^{\cup}\rangle y$ equivales $y\langle R\rangle x$.

A relation is a set of ordered pairs; the "state space" from which the elements of each pair are drawn will remain anonymous. The identity, empty and universal relations are denoted by I, $\perp\!\!\!\perp$ and $\top\!\!\!\top$, respectively. (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.) It is assumed that I, $\perp\!\!\!\perp$ and $\top\!\!\!\top$ are all distinct so that the state space is non-empty. The union and intersection of two relations R and S are denoted by $R{\cup}S$ and $R{\cap}S$, respectively. Relations are ordered by set inclusion denoted in the conventional way by the infix $\subseteq$ operator.

Relations contained in the identity relation will be called *monotypes*. (Others use the terminology *partial identity relation* or *coreflexive*.) By convention, we use R, S, T to denote arbitrary relations and A, B and C to denote monotypes. A monotype A thus has the property that if $x\langle A\rangle y$ then $x=y$. Clearly, the monotypes are in one to one correspondence with the subsets of the state space; we shall exploit this correspondence by identifying subsets of the state space with the monotypes. Specifically, by an abuse of notation, we write $x{\in}A$ for $x\langle A\rangle x$ (on condition that A is a monotype). We also identify monotypes with predicates, particularly when discussing induction principles (which are traditionally formulated in terms of predicates rather than sets). So we shall say "x has property A" meaning formally that $x\langle A\rangle x$. Continuing this abuse of notation, we use $\neg A$ to denote the monotype containing just those elements not in A. Thus, $x\langle\neg A\rangle y$ equivales the conjunction of $x{=}y$ and not $x\langle A\rangle x$.

Each relation has a canonical typing given by the left and right *domain* operators. The left domain of relation R, denoted $R{<}$, is a monotype defined by

$$x\in R{<}\ \equiv\ \exists(y{::}\,x\langle R\rangle y)\quad .$$

The right domain of R is the left domain of $R^{\cup}$. The left domain of R is the set of possible outputs of R and its right domain is the set of possible inputs. We write $R\in A\sim B$ whenever $A\bullet R=R=R\bullet B$. The domain operators are "canonical" in the sense that

$$R\in A\sim B\ \equiv\ R{<}\subseteq A\ \wedge\ R{>}\subseteq B\quad .$$

The expression $A\sim B$ is thus a typing of relation R and $R{<}\sim R{>}$ is a minimal typing: $R\in A\sim B$ means that the outputs of R are all included in the set A and its inputs are all included in the set B.

Computing scientists use the terminology "weakest liberal precondition of statement R with respect to postcondition A" for the the weakest condition B guaranteeing that R will map any input satisfying B into an output value satisfying A. We write $R\backslash A$ rather than the more usual $\mathsf{wlp}.R.A$. Formally, the wlp operator, $\backslash$, is defined by the Galois connection

$$(R\bullet B)< \;\subseteq\; A \;\equiv\; B \subseteq R\backslash A \quad ,$$

for all relations R and monotypes A and B. We call the function $B \mapsto (R\bullet B)<$ the *lower adjoint* and the function $A \mapsto R\backslash A$ the *upper adjoint* of this Galois connection. At point level the meaning of $R\backslash A$ is given by

$$x \in R\backslash A \;\equiv\; \forall(y\colon y\langle R\rangle x\colon y\in A) \quad .$$

For more discussion of the domain operators and weakest liberal preconditions the reader is referred to the companion paper [3].

Our precedence conventions are as follows. Unary operators always take precedence over binary operators. The precedence of the binary operators begins at the bottom end with the logical operators ($\equiv$, $\Leftarrow$, *etc.*, with their conventional relative precedence) followed by the set operators $\cap$, $\cup$ and $\subseteq$ (again with the conventional relative precedence) and concluded by the weakest liberal precondition operator and the composition operator which has the highest precedence. Other operators introduced later in the paper have higher precedence than composition.

1.2 Relators

The informal notion of an algebra is presumably well-known to the reader. Examples are monoids, groups, rings, vector spaces etc. A specific example of an algebra is the set of natural numbers with operations zero and successor.

In Universal Algebra an algebra is defined as a set, called the *carrier* of the algebra, together with a number of finitary operations on the set. In Category Theory a more abstract definition is given in which the algebraic structure is made prominent in the form of a so-called *functor*. In this paper we employ the notion of a "relator" instead of a functor. A relator is in one sense a specialisation and in another sense a generalisation of the categorical notion of a functor; the exact relationship between the two notions is not of relevance to this paper.

It is straightforward to demonstrate the relevance of relators to programming, so we can be quite brief in our discussion.

If one tries to understand the notion of "datatype" as it is used in programming then, in the first instance, one is likely to identify the notion with the notion of a set —thus in our terminology a monotype. For example, the set of natural numbers is a monotype. Also, the set of lists of natural numbers is a monotype. But "list" is not a datatype according to this definition — rather, as the example just given illustrates, "list" is a monotype transformer (i.e. a *function* from monotypes to monotypes). Since we want to regard "list" (and "tree", "forest", "heap" *etc.*) as a datatype let us postulate that a datatype is a monotype transformer rather than a monotype. This still allows us to consider monotypes themselves as datatypes if we are prepared to always identify a monotype, A, with the constant function that maps any monotype into A. In other words, there is a natural (1–1) embedding of the monotypes in the datatypes.

The datatypes used in computing science are such that the monotype obtained by applying a datatype to a monotype (for instance the monotype $\mathsf{List}.\mathsf{Nat}$ obtained by applying List to Nat) are classes of structures containing elements. This means that it is possible to "map" functions and relations elementwise over the elements of a structure. Taking the example of lists (where the terminology "map" comes from), for each relation R of type $A{\sim}B$ there is a relation $\mathsf{map}.R$ of type $\mathsf{List}.A \sim \mathsf{List}.B$ such that two lists stand in the relation $\mathsf{map}.R$ to each other if and only if they have the same length and corresponding elements are related to each other by the relation R. That is, for lists $[a,b,c,\ldots]$ and $[z,y,x,\ldots]$ of the same length,

$$\begin{aligned} & [a,b,c,\ldots]\,\langle \mathsf{map}.R\rangle\,[z,y,x,\ldots] \\ \equiv\ & a\langle R\rangle z \wedge b\langle R\rangle y \wedge c\langle R\rangle x \wedge \ldots \quad . \end{aligned}$$

Characteristic of the function map are four properties: it is monotonic (with respect to set inclusion), it commutes with (relational) converse, it distributes over (relational) composition, and applied to the identity relation on set A it yields the identity relation on $\mathsf{List}.A$. The first three of these properties can be stated directly in relation algebra. In addition, in relation algebra where monotypes can be identified with partial identity relations, the fact that a *monotonic* function maps (partial) identity relations into (partial) identity relations is easily expressed by the requirement that the identity relation on the universe of values is mapped into a subset of itself. These four properties form the notion of a *relator*.

Definition 1 A *relator* is a function, F, from relations to relations satisfying the four properties:

(converse) $F.(R^{\cup}) = (F.R)^{\cup}$,
(composition) $F.(R \bullet S) = F.R \bullet F.S$,
(monotonicity) $F.R \subseteq F.S \Leftarrow R \subseteq S$,
(identity) $F.I \subseteq I$.

□

It is straightforward to establish from the definition that a relator maps monotypes to monotypes (i.e. $F.A \subseteq I \Leftarrow A \subseteq I$) and functions to functions (i.e.

$$F.f \bullet (F.f)\cup \subseteq I \;\Leftarrow\; f \bullet f\cup \subseteq I$$

). Also, if A is a monotype then the constant function $\hat{A}$ (defined by $\hat{A}.R = A$) is a relator. Less straightforward to establish (but nevertheless do-able) is that relators commute with the domain operators (i.e. $F.(R<) = (F.R)<$ and $F.(R>) = (F.R)>$). This is another way of saying that if relation R has type $A \sim B$ then $F.R$ has type $F.A \sim F.B$.

From now on, rather than use the notation $\mathsf{map}.R$ we use the notation $\mathsf{List}.R$. $\mathsf{List}.R$ is thus a relation holding between two lists whenever they have equal length and corresponding elements are related by R . In particular, letting Nat denote the partial identity relation on the natural numbers, $\mathsf{List.Nat}$ is the identity relation on lists of natural numbers. That is, List maps the monotype Nat into the monotype $\mathsf{List.Nat}$.

Summarising this subsection we shall, for the purposes of this paper, identify the notion of a datatype with that of a relator.

1.3 Algebras

Having defined the notion of a relator, the notion of an algebra has a very compact definition: if F is a relator then an *F-algebra* is a relation R such that $F.R< \subseteq R>$. The left domain of an F-algebra is called the *carrier* of the algebra.

In conventional terms an algebra consists of a monotype, A , —the carrier of the algebra— and a *functional* relation, R , such that $R \in A \sim F.A$ and $R> = F.A$. Our definition relaxes several conditions in this definition. The most important relaxation is that we do not demand functionality. Less significant is that we do not require the right domain to have the form $F.A$ for some A . It turns out that this requirement is not needed, although in practice the requirement is always satisfied. (In fact, advanced texts on Universal Algebra relax the conditions in the same way.)

We shall also need the notion of an F-coalgebra: a relation is said to be an *F-coalgebra* if and only if its converse is an F-algebra.

The compactness of the definition of an algebra can be disconcerting at first sight. How does the definition capture the idea that an algebra includes a *number* of *finitary* operations? The key to this question is the use of two binary relators, disjoint sum and cartesian product.

The disjoint sum $A{+}B$ of the sets A and B consists of elements of the form $\mathsf{tag0}.x$ and $\mathsf{tag1}.y$ where x is an element taken from A and y is an element taken from B. The tagging operations are such that it is decidable whether an element z in $A{+}B$ has the form $\mathsf{tag0}.x$ or the form $\mathsf{tag1}.y$. Furthermore, it must be possible to strip the tag from its argument. In other words, the two tagging operations must each have an inverse. As a consequence, it is possible to distinguish between the two cases "z represents an element of A" and "z represents an element of B".

Disjoint sum can be axiomatised in relation algebra by postulating the existence of a *sum generator*, $\blacktriangledown$, mapping pairs of relations to a relation; auxiliary definitions are the *cogenerator*, $\triangledown$, which is defined to be the converse conjugate of the generator,

(2) $$U{\triangledown}V = (U^{\cup} \blacktriangledown V^{\cup})^{\cup} \quad ,$$

the tags, $\mathsf{tag0}$ and $\mathsf{tag1}$, which are special cases of the generator,

(3) $$\mathsf{tag0} = I{\blacktriangledown}\perp\!\!\perp \;\wedge\; \mathsf{tag1} = \perp\!\!\perp{\blacktriangledown}I \quad ,$$

and disjoint sum, defined by

(4) $$R{+}S = (\mathsf{tag0}{\bullet}R){\triangledown}(\mathsf{tag1}{\bullet}S) \quad .$$

(We omit the axioms relating these operators since they are not explicitly used in the paper.)

Operator $\blacktriangledown$ "generates" disjoint sums in the sense that when applied to relations with the same right domain it constructs a relation between the disjoint sum of the left domains of the argument relations and the common right domain:

$$R{\blacktriangledown}S \in A{+}B{\sim}C \quad \text{whenever } R \in A{\sim}C \text{ and } S \in B{\sim}C \;.$$

Informally, its interpretation is: nondeterministically decide whether to apply R or S and after the application tag the result to indicate which choice was made. Formally, this boils down to

$$R{\blacktriangledown}S \;=\; (\mathsf{tag0} \bullet R) \cup (\mathsf{tag1} \bullet S) \quad .$$

Disjoint sum $R{+}S$, with typing

$$R{+}S \in A{+}B \sim C{+}D \quad \text{for } R \in A{\sim}C \text{ and } S \in B{\sim}D \;,$$

is to be interpreted as "apply to each element". That is: first untag, then apply the proper argument of the sum, then tag again.

Disjoint sum is a *binary relator*. That is, converse commutes with disjoint sum (equally the converse conjugate operator is the identity function on disjoint sums), disjoint sum distributes over composition in the sense that

$$(R{+}S) \bullet (U{+}V) = (R{\bullet}U) + (S{\bullet}V)$$

for all relations R, S, U and V, disjoint sum is monotonic in both arguments, and finally

$$I{+}I \subseteq I \quad .$$

To see how disjoint sum can be used to encode a number of operations let us consider the algebra with carrier the natural numbers, Nat, and operations zero and succ (successor). First, the constant zero can be seen as a function from the unit set $\mathbb{1}$ (the set containing exactly one anonymous element) to Nat; second, succ is a function of type $\mathsf{Nat}{\leftarrow}\mathsf{Nat}$. Now we can combine these two functions into one of type $\mathsf{Nat} \leftarrow \mathbb{1}{+}\mathsf{Nat}$. Their combination $\mathsf{zero}{\triangledown}\mathsf{succ}$ has the two characteristic properties

$$\mathsf{zero}{\triangledown}\mathsf{succ}\bullet\mathsf{tag0} = \mathsf{zero} \quad \text{and} \quad \mathsf{zero}{\triangledown}\mathsf{succ}\bullet\mathsf{tag1} = \mathsf{succ} \quad .$$

In other words, $\mathsf{zero}{\triangledown}\mathsf{succ}$ is a case statement: depending on whether its supplied argument is tagged by $\mathsf{tag0}$ or by $\mathsf{tag1}$ it applies the function zero or the function succ.

Now $(\mathbb{1}{+})$ is a relator (in general fixing one of the arguments of a binary relator to be a monotype results in a unary relator). So the pair $(\mathsf{Nat}\,,\,\mathsf{zero}{\triangledown}\mathsf{succ})$ is a $(\mathbb{1}{+})$-algebra.

Another instance of a $(\mathbb{1}{+})$-algebra is the pair $(\mathsf{Bool}\,,\,\mathsf{true}{\triangledown}\mathsf{not})$ where Bool is the two-element set $\{\mathsf{T},\mathsf{F}\}$, true is the function of type $\mathsf{Bool}{\leftarrow}\mathbb{1}$ that is constantly T, and not is the function that maps T to F and F to T.

To encode operations of arity greater than one the cartesian product relator is used. Like disjoint sum, this binary relator is axiomatised in relation algebra by postulating the existence of a *product generator*, $\vartriangle$, mapping pairs of relations to a relation; the corresponding auxiliary operators are the *cogenerator*, $\blacktriangle$, defined to be the converse conjugate of the generator,

$$U{\blacktriangle}V = (U{\cup} \vartriangle V{\cup}){\cup} \quad ,$$

the *projections*, $\mathsf{out0}$ and $\mathsf{out1}$, which are special cases of the cogenerator,

$$\mathsf{out0} = I{\blacktriangle}\top\!\!\top \;\wedge\; \mathsf{out1} = \top\!\!\top{\blacktriangle}I \quad ,$$

and cartesian product, defined by

$$R{\times}S = (R{\bullet}\mathsf{out0}){\vartriangle}(S{\bullet}\mathsf{out1}) \quad .$$

(Again we omit the axioms relating these operators since they are not explicitly used in the paper.)

The product generator is such that $(x,y)\langle R{\vartriangle}S\rangle z$ equivales $x\langle R\rangle z$ and $y\langle S\rangle z$; the action of $R{\vartriangle}S$ is thus to generate a pair (an element of a cartesian product type) from a single value. The product $R{\times}S$ behaves like a parallel composition of R and S; it maps a pair to a pair such that $(u,v)\langle R{\times}S\rangle(x,y)$ equivales $u\langle R\rangle x$ and $v\langle S\rangle y$.

Now suppose A is a monotype. Then we can define a relator, F, by

$$F.R = \mathbb{1}{+}A{\times}R \quad .$$

Interpreting this informally, on sets (monotypes) F maps set B to the set $\mathbb{1}{+}A{\times}B$, and on functions F maps f to the function $\mathsf{id}_{\mathbb{1}}+\mathsf{id}_A\times f$. If f has type $B{\leftarrow}C$ the function $\mathsf{id}_{\mathbb{1}}+\mathsf{id}_A{\times}f$ has type $\mathbb{1}{+}A{\times}B \leftarrow \mathbb{1}{+}A{\times}C$. In the case that the argument to $\mathsf{id}_{\mathbb{1}}+\mathsf{id}_A\times f$ is an element of the 0th component of the disjoint sum the function is the identity function. In the other case, after stripping off the tag a pair is obtained; the function maps this pair into a tagged pair, the left component being a copy of the given left component and the right component being obtained from the given right component by applying the function f.

An example of a $(\mathbb{1}{+}A{\times})$-algebra is given by the pair $(\mathsf{List}.A\,,\,\mathsf{nil}{\triangledown}\mathsf{cons})$. Here $\mathsf{List}.A$ is the set of all finite lists over A and $\mathsf{nil}{\triangledown}\mathsf{cons}$ is the function with domain $\mathbb{1}{+}A{\times}\mathsf{List}.A$ and range $\mathsf{List}.A$ that maps an element of the unit type to the empty list and a pair consisting of an element a of A and a list l to the list $a\,\mathsf{cons}\,l$ formed by adding ("cons"ing in Lisp jargon) a to the front of the list l.

An example of a $(\mathbb{1}{+}\mathsf{Nat}{\times})$-algebra is the pair $(\mathsf{Nat}\,,\,\mathsf{zero}{\triangledown}\mathsf{add})$ where add adds two numbers together.

1.4 Initial Algebras

Of the two $(\mathbb{1}{+})$-algebras mentioned earlier — $\mathsf{zero}{\triangledown}\mathsf{succ}$ and $\mathsf{true}{\triangledown}\mathsf{not}$ [1] — the former is rather special because zero and succ are the constructors of the datatype Nat; it is a so-called *initial* $(\mathbb{1}{+})$-algebra. Similarly, the algebra $\mathsf{nil}{\triangledown}\mathsf{cons}$ is an initial $(\mathbb{1}{+}I{\times})$-algebra.

There are several equivalent definitions of an initial algebra. In this section we present a definition borrowed from category theory. Alternative definitions are presented later as part of the search for a suitable definition of reductivity.

When we define a relation on the natural numbers (i.e. a relation with right domain the natural numbers and arbitrary left domain) we often do so by induction on the structure of the natural numbers: we say which values are related to 0 and we say how to form values related to $n{+}1$ given knowledge about those values related to n. That such information *uniquely* characterises the relation we want to define is what makes $\mathsf{zero}{\triangledown}\mathsf{succ}$ an initial algebra. For

[1] We omit the carrier of the algebra because in each case it is deducible from the given relation.

example, the predicate even is uniquely characterised by the fact that 0 is even and $n{+}1$ is not even if and only if n is even.

Formally, we can write the definition of even as two equations:

$$\mathsf{even} \bullet \mathsf{zero} = \mathsf{true} \quad \text{and} \quad \mathsf{even} \bullet \mathsf{succ} = \mathsf{not} \bullet \mathsf{even} \quad .$$

Better still, using the disjointness of disjoint sum (specifically, $R \triangledown S = T \triangledown U$ equivales $R{=}T$ and $S{=}U$), these two equations can be combined into one:

$$(\mathsf{even}{\bullet}\mathsf{zero}) \triangledown (\mathsf{even}{\bullet}\mathsf{succ}) = \mathsf{true} \triangledown (\mathsf{not}{\bullet}\mathsf{even})$$

"Defusing" this equation (using $R \bullet (S \triangledown T) = (R{\bullet}S) \triangledown (R{\bullet}T)$ on the left side and $(R{\bullet}S) \triangledown (T{\bullet}U) = (R \triangledown T) \bullet (S{+}U)$ on the right side, the so-called *fusion* properties) —

$$\mathsf{even} \bullet \mathsf{zero} \triangledown \mathsf{succ} \;=\; \mathsf{true} \triangledown \mathsf{not} \bullet \mathbb{1}{+}\mathsf{even}$$

— we identify three components in the equation, the relator $(\mathbb{1}{+})$, and the two $(\mathbb{1}{+})$-algebras $\mathsf{zero} \triangledown \mathsf{succ}$ and $\mathsf{true} \triangledown \mathsf{not}$.

In the same way, a relation on lists is uniquely defined if we say which elements are related to the empty list and which elements are related to a list that is the "cons" of an element with a list. For example, the relation $\mathsf{prefixes}$ is uniquely characterised by the two equations

$$\mathsf{nill} \langle \mathsf{prefixes} \rangle \mathsf{nill}$$

and

$$x \langle \mathsf{prefixes} \rangle (a \ \mathsf{cons}\ y) \;\equiv\; x = \mathsf{nill} \vee \exists (z\colon x = a \ \mathsf{cons}\ z\colon z \langle \mathsf{prefixes} \rangle y) \quad .$$

Expressed as one, point-free equation this is

$$\mathsf{prefixes} \bullet \mathsf{nil} \triangledown \mathsf{cons} \;=\; \mathsf{nil} \triangledown (\mathsf{nil} {\cup} \mathsf{cons}) \bullet \mathbb{1}{+}I{\times}\mathsf{prefixes} \quad .$$

(In order to be formally correct we have used nill to denote the empty list and nil to denote the function with right domain $\mathbb{1}$ that returns the value nill.) Again we recognise a relator and two algebras: in this case the relator is $(\mathbb{1}{+}I{\times})$ and the two $(\mathbb{1}{+}I{\times})$-algebras are $\mathsf{nil} \triangledown \mathsf{cons}$ and $\mathsf{nil} \triangledown (\mathsf{nil} {\cup} \mathsf{cons})$.

The categorical definition of an initial algebra generalises from these two examples.

Definition 5 An F-algebra, R, is said to be *initial* if there is a function $S \mapsto (\!|S|\!)$ mapping F-algebras to relations such that, for all F-algebras S and all relations $X \in S{<} \sim R{<}$,

$$X = (\!|S|\!) \;\equiv\; X \bullet R = S \bullet F.X \quad .$$

□

As mentioned earlier, $\mathsf{zero}{\triangledown}\mathsf{succ}$ is an initial $(\mathbb{1}{+})$-algebra and $\mathsf{nil}{\triangledown}\mathsf{cons}$ is an initial $(\mathbb{1}{+}I{\times})$-algebra. We shall not prove these facts here; their validity will emerge from the alternative characterisations of initiality given in the next section.

2 Induction Principles

Initiality of an algebra is a well-known and well-studied property but it is also a highly restrictive property. There are many algebras of interest that are not in any sense initial. In this section and the next we explore three generalisations of initiality, which we call "inductivity", "reductivity" and "well-foundedness".

In the next subsection we recall a well-known property of initial algebras, the so-called "no junk and no confusion" property. It is this property that we want to abandon in the generalisation process. What is left over is an induction principle; our goal is to identify the formulation that is the most appropriate to programming applications. Inductivity, reductivity and well-foundedness are three contenders. Formally, if an algebra has the no junk and no confusion property then being inductive, reductive or well-founded are all equivalent properties; moreover, the conjunction of the no junk and no confusion property with inductivity, reductivity or well-foundedness is equivalent to initiality.

2.1 No Junk and No Confusion

The Peano definition of the natural numbers consists of two parts: the fact that zero and succ construct the natural numbers, and an induction principle (simple mathematical induction). In this subsection we consider what it means to "construct" the natural numbers. In the next subsection we formulate the notion of inductivity. Like the preceding sections, this section belongs to the "well-known" material.

The sense in which zero and succ "construct" the natural numbers is commonly split into two conditions, the so-called "no-junk" and "no-confusion" properties.

The former property is that Nat, the carrier of $\mathsf{zero}{\triangledown}\mathsf{succ}$, contains no "junk", i.e. elements not constructed by zero or succ. In other words, $\mathsf{zero}{\triangledown}\mathsf{succ}$ is *onto* Nat.

The latter property is that there is no "confusion" about how elements of Nat are constructed, i.e. $\mathsf{zero}{\triangledown}\mathsf{succ}$ is an invertible function. (It is always possible to differentiate between zero and a number that is a successor of another, and between numbers that are successors of different numbers.)

The combination of the no junk and no confusion properties is that $\mathsf{zero}\triangledown\mathsf{succ}$ is a bijection between Nat and $\mathbb{1}{+}\mathsf{Nat}$. This is a necessary condition for an algebra to be initial (and known to category theoreticians as "Lambek's lemma" [6]).

Lemma 6 (Lambek's Lemma) An initial F-algebra with carrier A is a bijection between $F.A$ and A.

□

As mentioned in the introduction to this section, it is this property of initial algebras that we want to abandon. That is, we seek a property that when conjoined with this bijectivity property is equivalent to initiality. In the remaining subsections we formulate the notions of inductivity and reductivity with respect to a datatype. Our approach is pragmatic: we consider standard examples of induction principles and formulate those principles in relation algebra.

2.2 Inductivity

Our goal in this section is to abstract from the principle of simple mathematical induction in order to obtain a general notion of inductivity with respect to a datatype.

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

The question we have to tackle is how to generalise the notion of "invariance" to an arbitrary F-algebra. We propose calling a monotype B an *invariant* of F-algebra R iff

$$(R\bullet F.B)< \subseteq B \quad .$$

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

$$\forall(x{:}\,\exists(y{:}\; x\langle R\rangle y{:}\; y\in F.B){:}\, x{\in}B) \quad .$$

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

Our notion of an F-algebra, R, "admitting induction", or being "inductive" with respect to F, is that it is possible to deduce that the carrier of R satisfies some property B whenever B is a invariant of the algebra.

Definition 7 An F-algebra, R, is said to be F-*inductive* if, for all $B \subseteq R{<}$,

$$R{<} \subseteq B \;\Leftarrow\; (R \bullet F.B){<} \subseteq B \quad .$$

Equivalently, in the predicate calculus,

$$\forall(x{::}\, x{\in}B) \Leftarrow \forall(x{:}\ \exists(y{:}\ x\langle R\rangle y{:}\ y \in F.B){:}\ x{\in}B) \quad .$$

(Here the dummy x in both quantifications ranges over $R{<}$.)

□

We leave the reader to check that $\mathsf{zero}{\triangledown}\mathsf{succ}$ being $(1\!\!1+)$-inductive is equivalent to the principle of simple mathematical induction.

Another example of F-inductivity is provided by lists. Suppose F is the relator that maps X to $1\!\!1{+}(I{+}(X{\times}X))$ and R the function $\mathsf{nil}{\triangledown}(\tau{\triangledown}\mathsf{join})$ where τ maps a value into a singleton list and join joins two lists into one list. Then R is F-inductive. Indeed, this is precisely the statement that a property is true of a list if it is true of the empty list, all singleton lists, and the join of any two lists which themselves have the given property.

The last example provides evidence that F-inductivity is *not* so useful for guaranteeing termination of programs. It can be shown that the equation

$$X{::}\ X = R \bullet 1\!\!1{+}(I{+}(X{\times}X)) \bullet (\mathsf{nil}{\triangledown}(\tau{\triangledown}\mathsf{join}))^{\cup}$$

has a unique solution for all R. The relation $(\mathsf{nil}{\triangledown}(\tau{\triangledown}\mathsf{join}))^{\cup}$ does not however necessarily decrease the length of a list since the join of two empty lists is again an empty list. Trying to "execute" this equation may thus result in a non-terminating computation.

2.3 Reductivity

By an elementary monotonicity argument, it is easy to see that if B is an invariant of $\mathsf{zero}{\triangledown}(>)$ then B is an invariant of $\mathsf{zero}{\triangledown}\mathsf{succ}$. Since $\mathsf{zero}{\triangledown}\mathsf{succ}$ is $(1\!\!1+)$-inductive it follows that $\mathsf{zero}{\triangledown}(>)$ is also $(1\!\!1+)$-inductive.

Our reason for being interested in the inductivity of $\mathsf{zero}{\triangledown}(>)$ is that we want to explain the principle of *strong* mathematical induction in terms of a general property satisfied by $\mathsf{zero}{\triangledown}(>)$. (Recall that in a proof by strong mathematical induction the induction hypothesis is that *all* numbers below the number in question have the hypothesised property.) The fact that $\mathsf{zero}\triangledown(>)$ is $(\mathbb{1}+)$-inductive does not, however, correspond to the principle of strong mathematical induction. It corresponds to the statement that a property is true of all natural numbers if it can be proved for zero and for all larger numbers, n, under the assumption that the property holds for *some* number below n.

The simplicity of the step from $(\mathbb{1}+)$-inductivity of $\mathsf{zero}\triangledown\mathsf{succ}$ to $(\mathbb{1}+)$-inductivity of $\mathsf{zero}\triangledown(>)$ suggests a strategy for defining an alternative notion of inductivity. What we seek is a notion of "inductivity" in which the roles of $\mathsf{zero}\triangledown\mathsf{succ}$ and $\mathsf{zero}\triangledown(>)$ are turned around: From the fact that $\mathsf{zero}\triangledown(>)$ gives rise to an induction principle it should be straightforward to observe that $\mathsf{zero}\triangledown\mathsf{succ}$ does so too.

We can turn the roles around by postulating a notion that is anti-monotonic in the relation R rather than monotonic as is the case for F-inductivity. The theory of Galois connections gives a clue how to do this. Noting that the function mapping A to $(R\bullet A)_<$ is monotonic in R it is an elementary fact that its upper adjoint —the function mapping A to $R\backslash A$— is anti-monotonic. The general idea is thus to define a notion dual to inductivity by simply replacing one function by its upper adjoint.

For technical reasons the execution of this idea is a little more complicated. One complication is that we replace the function mapping monotypes B *under* $R_>$ to $(R\bullet B)_<$ (thus a monotype under $R_<$) by its upper adjoint —the function mapping monotype B under $R_<$ to $R_>\bullet R\backslash B$ (a monotype under $R_>$). The second complication is that we define the notion on *co-algebras* rather than on algebras. (This is motivated by issues yet to be discussed.)

Definition 8 An F-coalgebra, R, is said to be F*-reductive* if, for all $B\subseteq R_>$,

$$R_>\subseteq B \;\Leftarrow\; R_>\bullet R\backslash F.B \subseteq B \;\; .$$

Equivalently, in the predicate calculus,

$$\forall(x{::}\,x{\in}B) \Leftarrow \forall(x{:}\;\forall(y{:}\;y\langle R\rangle x{:}\;y\in F.B){:}\;x{\in}B) \;\; .$$

(Here the dummy x in both quantifications ranges over the right domain of R.)

□

We recommend that the reader compare the definition of F-inductivity of relation R with F-reductivity of the converse of R noting how one function has been replaced by its upper adjoint in the point-free definition (equivalently, how an existential quantification has been replaced by a universal quantification).

(Note that definition 8 is not one of those mentioned in [3]. The equivalence is easily established using the techniques illustrated in the latter paper. The definition given here has been chosen to facilitate the comparison with inductivity.)

A proof that $\mathsf{zero}^{\cup}\blacktriangledown(<)$ (the converse of $\mathsf{zero}\triangledown(>)$) is $(\mathbb{1}+)$-reductive proceeds as follows. We first observe the lemmas

$$(R\blacktriangledown S)\backslash(A+B) = R\backslash A \bullet S\backslash B \quad \text{and} \quad \mathsf{zero}^{\cup}\backslash\mathbb{1} = I \quad .$$

(Both of these are straightforward calculations.) Thus,

$$(\mathsf{zero}^{\cup}\blacktriangledown(<))\backslash(\mathbb{1}+A) = (<)\backslash A$$

and

$$\begin{array}{ll}
 & \mathsf{zero}^{\cup}\blacktriangledown(<) \text{ is } (\mathbb{1}+)\text{-reductive} \\
\equiv & \qquad \{ \quad \text{definition 8, } (\mathsf{zero}^{\cup}\blacktriangledown(<))^{>} = \mathsf{Nat} \quad \} \\
 & \forall(A\colon\ A\subseteq\mathsf{Nat}\colon\ \mathsf{Nat}\subseteq A \Leftarrow \mathsf{Nat}\bullet(\mathsf{zero}^{\cup}\blacktriangledown(<))\backslash(\mathbb{1}+A)\subseteq A) \\
\equiv & \qquad \{ \quad \text{above} \quad \} \\
 & \forall(A\colon\ A\subseteq\mathsf{Nat}\colon\ \mathsf{Nat}\subseteq A \Leftarrow \mathsf{Nat}\bullet(<)\backslash A\subseteq A) \quad .
\end{array}$$

But the latter is the principle of strong mathematical induction. (Specifically, it is the statement that

$$\forall(A\colon A\subseteq\mathsf{Nat}\colon \forall(m::m\in A) \Leftarrow \forall(m\colon\forall(n\colon n<m\colon n\in A)\colon m\in A))$$

where the dummies m and n range over the natural numbers.) Thus we have shown that the $(\mathbb{1}+)$-reductivity of $\mathsf{zero}^{\cup}\blacktriangledown(<)$ is equivalent to the principle of strong mathematical induction. (Our reductivity calculus allows in addition a derivation of strong mathematical induction from first principles. Including it here would, however, be a bit overdone.)

We have argued that inductivity is not the correct generalisation of the notion of an initial algebra because it doesn't capture termination properties. The example we used to support our argument was the inductive relation $\mathsf{nil}\triangledown(\tau\triangledown\mathsf{join})$. The converse of this relation is not reductive. To see why, take B in definition 8 to be the set containing just the empty list. Clearly, not every list is an element of B. But with R equal to $(\mathsf{nil}\triangledown(\tau\triangledown\mathsf{join}))^{\cup}$ and $F.B$ equal to $\mathbb{1}+(I+B\times B)$ it is the case that

$$\forall(x\colon\forall(y\colon\ y\langle R\rangle x\colon\ y\in F.B)\colon x\in B)$$

(in words, every list that is constructed by the function $\mathsf{nil}\triangledown(\tau\triangledown\mathsf{join})$ and is the join of two empty lists is itself an empty list).

There are of course circumstances when the notions of inductivity and reductivity coincide. In particular, we have:

Theorem 9 If R is a bijection between $R<$ and $F.R<$ then

$$R\cup \text{ is } F\text{-reductive} \equiv R \text{ is } F\text{-inductive} \quad .$$

More generally, the converse of any *injective* F-inductive relation is F-reductive and the converse of any F-reductive relation R such that $R< = F.R>$ is F-inductive.

□

Recalling that R is an initial F-algebra if and only if it is a bijection between $R<$ and $F.R<$ and it is F-inductive, it follows that R is an initial F-algebra if and only if it is a bijection between $R<$ and $F.R<$ and $R\cup$ is F-reductive. Thus, for initial algebras reductivity and inductivity are indistinguishable.

3 Well Foundedness

We have argued in the previous section that inductivity and reductivity are equivalent notions for initial algebras but not (necessarily) equivalent otherwise. We have also presented one argument why reductivity is a more appropriate notion to study, namely that reductivity rather than inductivity captures the notion of strong mathematical induction. In addition we have argued that inductivity does not guarantee termination properties. In this section we want to argue that reductivity does guarantee termination in the context of a broad class of programs.

Characteristic of the definition of the initiality of an algebra is the uniqueness requirement on the solution of certain equations. Unicity is also evident in the notion of well-foundedness of a relation: a relation R is well-founded if and only if the equation $X{::}\ X{=}X{\bullet}R$ has the unique solution $X = \perp\!\!\perp$. (See, for example, [10]) This suggests that we try to generalise the notion of well-foundedness (to well-foundedness with respect to a datatype) by focussing on this aspect of initiality. We shall indeed show that initiality of an F-algebra can be characterised by a combination of bijectivity (the no confusion and no junk properties mentioned earlier) and a unicity property in such a way that when F is the identity relator the unicity property is equivalent to the conventional notion of well-foundedness. We shall also observe, however, that the proposed notion of well-foundedness does not guarantee termination of programs whereas reductivity does. In this way we hope to provide more evidence for the importance of reductivity.

At this stage in our work we have only investigated the notions of well-foundedness and reductivity in the context of a certain (broad) class of equations. The motivation for considering this class of equations is presented in the next subsection.

3.1 Hylo Equations

The statement R; **while** A **do** S ;T consisting of a **while** statement, initialisation, R, and finalisation, T, corresponds to the relation

$$T \bullet \neg A \bullet (S \bullet A)^* \bullet R \quad .$$

The imperative style of designing such a decomposition of specification X consists of first designing an invariant, Y, finalisation, T, and termination condition, $\neg A$, such that

$$T \bullet \neg A \bullet Y \subseteq X \quad .$$

(The finalisation is not usually made explicit in programming texts. In the case that the invariant, Y, is obtained from the specification, X, by the technique of "replacing a constant by a variable" the finalisation is a projection from the state space of the program onto the output variables. Indeed, it is typically the case that the finalisation is a projection or other simple transformation of the state space.)

The next step is to design the loop body, S, so that Y is indeed maintained invariant. That is, we design S so that

$$S \bullet A \bullet Y \subseteq Y \quad .$$

The third step is to design the initialisation, R, to establish the invariant, i.e. so that

$$R \subseteq Y \quad .$$

The design method used in functional programming is somewhat different. First, in the general case, some preprocessing, R, is done to transform the given specification, X, to a specification, Z, that has a recursive solution. That is, we design Z and R so that

$$Z \bullet R \subseteq X \quad .$$

Next, a case analysis is performed on Z to identify a base case and a recursive case. That is, we design T, A and S such that

$$T \bullet \neg A \subseteq Z$$

(the base case) and

$$Z \bullet S \bullet A \subseteq Z \quad .$$

Common to both solution strategies is that the specification, X, is refined into T, A, S and R such that

$$T \bullet \neg A \bullet (S \bullet A)^* \bullet R \subseteq X \quad .$$

The difference resides in the two parsings of the left side of this inequality as

$$(T \bullet \neg A) \bullet ((S \bullet A)^* \bullet R)$$

or as

$$(T \bullet \neg A \bullet (S \bullet A)^*) \bullet R$$

and the fact that U^* is the least solution of both the equation

(10) $\qquad V{::}\;\; V = I \cup U \bullet V$

and the equation

(11) $\qquad V{::}\;\; V = I \cup V \bullet U \quad .$

The imperative style of programming works backwards from the specification, the functional style works forwards to the specification.

In order to develop a theory that is independent of which programming style is adopted it is useful to recognize (10) and (11) as instances of the same general form. Specifically, using the fusion properties mentioned earlier, we can rewrite (10) as

(12) $\qquad V{::}\;\; V = I \triangledown U \bullet I + V \bullet I \blacktriangledown I$

and (11) as

(13) $\qquad V{::}\;\; V = I \triangledown I \bullet I + V \bullet I \blacktriangledown U \quad .$

The general form of these equations is

(14) $\qquad V{::}\;\; V = R \bullet F.V \bullet S$

where R and S are relations and F is a relator. Such equations are called *hylo* equations and the least solution to such an equation is called a *hylomorphism*.

It has been observed that many programs take the form of hylo equations. (See [8,11,3] for several examples.) An illuminating insight into the importance of such equations is provided by the *hylo* theorem that the least solution of (14) is the composition of the (unique) solution of the equation

(15) $\qquad V{::}\;\; V \bullet \mathsf{in} = R \bullet F.V$

after the (unique) solution of the equation

(16) $\qquad V{::}\;\; V \bullet \mathsf{out} = F.V \bullet S$

where in is any initial F-algebra and out is its inverse. The solution of (15) (called a *catamorphism*) breaks down an element of the carrier of the initial algebra and the solution of (16) (called an *anamorphism*) builds this element up. (*Cata* and *ana* mean down and up in Greek [8].) The (carrier of the) initial algebra thus acts as an intermediate or so-called "virtual" datatype [11].

Two examples of virtual datatypes are provided by **while** statements and the factorial function. In the case of **while** statements the virtual datatype is an initial $(I+)$-algebra (see (12) or (13)) the carrier of which is isomorphic to $I{\times}\mathsf{Nat}$ —thus an element of the carrier is a pair consisting of an element of the state space and a natural number, the latter being a "virtual" count of the number of times the loop body is executed. In the case of the factorial function, the standard recursive definition can be rewritten in the form of the hylo equation:

$$n! = \hat{1} \triangledown \mathsf{times} \bullet I{+}(I \times n!) \bullet I{+}(I \vartriangle \mathsf{succ}^{\cup}) \bullet (=0) \blacktriangledown (\neq 0) \quad .$$

The "virtual" datatype is thus a list (an initial $I{+}(I{\times})$-algebra) of natural numbers from n down to 1 and the hylo theorem states that the factorial of n can be calculated by constructing this list (the anamorphism phase) and then multiplying the numbers together (the catamorphism phase).

The design of hylomorphisms combines sequential programming with functional programming. The design of the anamorphism involves the construction of an invariant and variant function whilst the construction of the catamorphism is a recursive problem decomposition. For example, merge sort consists of two phases. In the first phase a binary tree structure is formed with the invariant property that the subtrees are balanced. In the second phase the tree is broken down into a sorted list whereby two lists are joined by the standard "merging" process. Of course, in the usual implementations of merge sort the construction of the tree is "virtual", i.e. not made explicit.

3.2 F-well-foundedness

Well-foundedness of relation R is the property that the equation $X{::}\ X{=}X{\bullet}R$ has the unique solution $X{=}\perp\!\!\!\perp$ (where $\perp\!\!\!\perp$ denotes the empty relation). Equivalently, R is well-founded if and only if

$$\forall(S,X{::}\ X = S \bullet X \bullet R \equiv X = \perp\!\!\!\perp) \quad .$$

Taking account of the theorem that an initial F-algebra is a bijection, we also have that if R is an initial F-algebra then there is a function $S{\mapsto}(\![S]\!)$ such that

$$\forall(S,X{::}\ X = S \bullet F.X \bullet R^{\cup} \equiv X = (\![S]\!))$$

Abstracting from these examples we propose the following definition:

Definition 17 The relation R is said to be F-well-founded if and only if, for all relations S, the equation

$$X{::}\ X = S \bullet F.X \bullet R$$

has a unique solution.

□

By design, R is well-founded (in the conventional sense) if and only if it is id-well-founded, where id is the identity relator. Moreover, the converse of any initial F-algebra is F-well-founded. A stronger statement can be made:

Theorem 18 Suppose R is an F-coalgebra that is a bijection between $F.R{>}$ and $R{>}$. Then the following are all equivalent:

(a) R is F-well-founded,
(b) R is F-reductive,
(c) $R^{\cup}$ is an initial F-algebra.

□

(The equivalence between (b) and (c) has already been observed.)

One of the fundamental properties of reductivity is that it implies well-foundedness. This is theorem 7 of [3]. The converse is not true. Let R be a non-empty, well-founded relation (for example the relation $\mathsf{succ}^{\cup}$ on natural numbers). Then it is easy to show that, for all X and all S,

$$X = S \bullet X{\times}X \bullet I{\vartriangle}R \ \equiv\ X{=}\perp\!\!\perp \quad .$$

The relation $I{\vartriangle}R$ is thus F-well-founded, where $F.X = X{\times}X$. It is not F-reductive. Informally, this is because $I{\vartriangle}R$ constructs a pair from a given element x of which the first component is just a copy of x; for the relation to be F-reductive it is required that both components are "smaller" than x (i.e. have a "reduced" size relative to x).

This example is instructive because it demonstrates that the uniqueness of solutions of equations is not the same as a guarantee of termination. "Executing" the equation

$$X{::}\ X = S \bullet X{\times}X \bullet I{\vartriangle}R$$

corresponds to a recursive computation that builds an infinitely branching binary tree that is then broken down by the relation S. Each interior node spawns two new nodes, one of which is a copy of itself and the other has a value that is "smaller" (i.e. related to its own value by the well-founded relation R). Such an execution cannot compute a solution to the equation because well-foundedness of R guarantees that the process of spawning a smaller value cannot continue indefinitely. (That is the only solution to the equation is the empty relation.) If, however, the computation is so implemented that the copying process is always carried out then the computation will not terminate.

4 A Calculus of Reductivity

Much of the companion paper [3] was about developing a calculus of reductivity — that is calculational rules for constructing basic reductive relations and for transforming relations reductive with respect to one relator to relations reductive with respect to another relator. It is not the purpose of this paper to repeat the theorems there. A brief explanation of the strategy used to derive the calculus is nevertheless in order.

Our approach throughout this paper has been to try to generalise properties of initial algebras by abandoning one condition necessary for an algebra to be initial, namely that the algebra be a bijection. We have sought a property of algebras that when conjoined with this bijectivity property is equivalent to initiality. In so doing we have identified three possible contenders, inductivity, well-foundedness and reductivity, and we have argued why reductivity is possibly the most relevant to programming.

An advantage of this approach is that in further investigations of the properties of reductive relations we can always draw upon what is already known about initial algebras. In constructing a calculus of reductivity we have drawn upon the fact that initial algebras are themselves a generalisation of the notion of a least prefix point in lattice theory, and that a calculus of initial algebras can be obtained by generalising fixed point calculus [1].

As an elementary example of the sort of properties this leads to we may cite the following theorem. (A more sophisticated theorem called the "iterated square theorem" has been formulated and proved by Freyd in [4]. In the theory of preorders we have the theorem that if the "square" f^2 of monotonic endofunction f has a least prefix point then so does f itself [2]. Indeed, the least prefix point of f^2 is the least prefix point of f. The proof consists of showing that the prefix points of f form a subset of the prefix points of f^2 and that the *least* prefix point of f^2 is also a prefix point of f. The generalisation of this theorem is that if α is an F-algebra then $\alpha \bullet F.\alpha$ is an F^2-algebra; moreover, any initial F-algebra is mapped in this way onto an initial F^2-algebra. (In the general context of category theory the existence of products is needed to establish the latter part of this theorem.)

A concrete instance of this theorem in action is the definition of integer division by two: Zero and one divided by two are both zero; $n{+}2$ divided by two is equal to n divided by two. That this defines division by two uniquely is a straightforward application of the fact that if α is an initial F-algebra then $\alpha \bullet F.\alpha$ is an initial F^2-algebra. It suffices to instantiate F to $(\mathbb{1}{+})$ and α to

[2] f^2 is defined by $f^2.x = f.(f.x)$. A prefix point of f is a value x such that $f.x \leq x$.

$\mathsf{zero} \triangledown \mathsf{succ}$; then observe that

$$\mathsf{zero} \triangledown \mathsf{succ} \bullet \mathbb{1} + (\mathsf{zero} \triangledown \mathsf{succ}) = \mathsf{zero} \triangledown (\mathsf{succ} \bullet \mathsf{zero} \triangledown \mathsf{succ} \bullet \mathsf{succ}) \quad .$$

Rather than confine attention to initial algebras we can decompose the theorem into two parts. The first part is the rather trivial observation that the mapping $\alpha \mapsto F.\alpha \bullet \alpha$ preserves bijectivity properties ("no junk and no confusion"). ($F.\alpha$ and α have been reversed for the technical reason that reductivity is a property of co-algebras instead of algebras.) The second part is the more interesting: the mapping transforms an F-reductive relation into an F^2-reductive relation.

The companion paper [3] makes a start on a systematic study of reductivity transformations that can be derived in this way. For a more detailed investigation and proofs of all the claims made in this paper the reader is referred to the first author's forthcoming Ph.D. thesis.

References

[1] R.C. Backhouse, M. Bijsterveld, R. van Geldrop, and J.C.S.P. van der Woude. Categorical fixed point calculus. In D. Pitt, D.E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference*, volume 953 of *LNCS*, pages 159–179. Springer-Verlag, 1995.

[2] R.S. Bird, O. de Moor, and P. Hoogendijk. Generic functional programming with types and relations. *J. of Functional Programming*, 1995. To appear.

[3] H. Doornbos and R.C. 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.

[4] 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.

[5] J. Jeuring. Polytypic pattern matching. In S. Peyton Jones, editor, *Proceedings Functional Programming Languages and Computer Architecture, FPCA '95*, June 1995.

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

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

[8] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[9] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 228–266. Springer-Verlag, May 1995.

[10] G. Schmidt and T. Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg, 1993.

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

[12] P. Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, June 1990.