Datatype-Generic Reasoning

Roland Backhouse

School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, England, rcb@cs.nott.ac.uk

Abstract. Datatype-generic programs are programs that are parameterised by a datatype. Designing datatype-generic programs brings new challenges and new opportunities. We review the allegorical foundations of a methodology of designing datatype-generic programs. The effectiveness of the methodology is demonstrated by an extraordinarily 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, particularly among functional programmers. This has led to the development of a new programming concept, called *(datatype-)generic* programming [21, 14, 15, 22]. 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 the specific syntactic structure of the programs to achieve a higher compression ratio [15].

The idea of making data structure a parameter opens up new challenges and new opportunities. A major new insight is to consider the algebraic structure of data structures — how complex data structures are built from simpler components. In this paper, we review the theoretical foundations of reasoning about datatype-generic programs. We review the notion of "F-reductivity", introduced by Doornbos [11, 9, 12], and show its application to establishing the

well-foundedness of the occurs-in relation in a datatype-generic unification algorithm [20, 7].

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 write $R :: I \leftarrow J$ (read "R has type 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, and $I \leftarrow J$ is called its *type*. 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.

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

Functions are "single-valued" relations; a relation 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 $f.x$ rather than say x^f (that is the function to the left of its argument). A sensible consequence is that type arrows point from right to left.

2.2 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 right domain $R^{\rhd}$ of a

relation R of type $I \leftarrow J$ is a coreflexive of type $I \leftarrow I$ satisfying the property that

$$\langle \forall A \,:\, A \subseteq \mathsf{id}_I \,:\, A \cdot R = R \equiv R^{<} \subseteq A \rangle \quad . \tag{1}$$

Given a coreflexive A, $A \subseteq \mathsf{id}_I$, 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 symetrically 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 Galois connection, for all relations S,

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

The operator $\backslash$ is called a *division* operator (because of the similarity of the above rule to the rule of division in ordinary arithmetic). The relation $R \backslash T$ is called a *residual* or a *factor* of the relation T. Interpreting relations as specifications, the above Galois connection 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* [16].

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

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

Again, we use a division-like notation, rather than "wlp", to emphasise the similarity with division in normal arithmetic.

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" [13] 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 [8]. (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:

$$F.R \text{ has type } F.I \xleftarrow{\mathcal{C}} F.J \text{ whenever } R \text{ has type } I \xleftarrow{\mathcal{D}} J. \tag{3}$$

$$F.R \cdot F.S = F.(R \cdot S) \quad \text{for each } R \text{ and } S \text{ of composable type,} \tag{4}$$

$$F.\mathsf{id}_A = \mathsf{id}_{F.A} \quad \text{for each object } A, \tag{5}$$

$$F.R \subseteq F.S \;\Leftarrow\; R \subseteq S \quad \text{for each } R \text{ and } S \text{ of the same type,} \tag{6}$$

$$(F.R)^{\cup} = F.(R^{\cup}) \quad \text{for each } R. \tag{7}$$

For example, List is a unary relator, and product is a binary relator. If R is a relation of type $I \leftarrow J$ then $\mathsf{List}.R$ relates a list of Is to a list of Js whenever the two lists have the same length and corresponding elements are related by R. The relation $R \times S$ relates two pairs if the first components are related by R and the second components are related by S. List is an example of an inductively-defined datatype; in [2] it was observed that all inductively-defined datatypes are relators.

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. Formally, relation R of type $I \leftarrow J$ is *total* iff $\mathsf{id}_J \subseteq R^{\cup} \cdot R$. 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 [8] omit (7) and define a relator to be a monotonic functor. However, their theorem 5.1, which purports to justify the omission, is false.)

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 [27, 28]. 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 G is denoted by $F \hookleftarrow G$ and defined by

$$\alpha :: F \hookleftarrow G \quad \equiv \quad (F.R \cdot \alpha_J \supseteq \alpha_I \cdot G.R \qquad \text{for each } R :: I \leftarrow J) \ . \tag{8}$$

A relationship between naturality in the allegorical sense and in the categorical sense is the following [17]. 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 (8).

4 A Programming Paradigm

4.1 Hylo programs

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.

In standard treatments of the discipline of sequential programming, the class of programs considered is the class of **while** programs; it has long been accepted that arbitrary **goto** programs are undesirable. But, whilst theoretically expressive enough, **while** programs are inadequate to express many elegant and well-known *recursive* programs, like quicksort. On the other hand, arbitrary recursion is also undesirable. Restriction to a more limited class of recursive programs is desirable for a sound discipline of datatype-generic programming.

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 [25], the name having been coined by Meijer [26]. Unlike [25], however, the current paper is not restricted to functional programs.

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

Space does not allow us to give detailed examples of hylo programs here. Briefly, 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 $\langle X :: I + X \rangle$ encapsulates repetition, $\langle X :: I + X \times X \rangle$ encapsulates a divide and conquer strategy, and $\langle X :: F.(I \times X) \rangle$ encapsulates primitive recursion. A first step in the design of hylo programs is the choice of the relator [11]. 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 (even in the most general case) [5].

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

$$(\!|R|\!) :: A \leftarrow I \quad \text{if } R \text{ has type } A \leftarrow F.A \quad , \tag{9}$$

$$(\!|\mathsf{in}|\!) = \mathsf{id}_I \quad \text{, and} \tag{10}$$

$$(\!|R|\!) \cdot (\!|S|\!)^{\cup} \;=\; \langle \mu X :: R \cdot F.X \cdot S^{\cup} \rangle \quad . \tag{11}$$

That is, $(\!|R|\!) \cdot (\!|S|\!)^{\cup}$ 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 [23, 24] 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 [6]). Suppose F is an endorelator on a locally-complete, tabular allegory $\mathcal{A}$. Let F' denote the endofunctor obtained by restricting F to the objects and arrows of $Map(\mathcal{A})$. Then, in is an initial F'-algebra equivales it is a relational initial F-algebra. $\square$

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 (11) is the property that is most often understood as the "hylo theorem". Property (9) is a necessary prerequisite; essentially it states that catamorphisms are well-defined on relations given that they are well-defined on functions. Property (10) 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 straightforward-to-use techniques for guaranteeing termination of programs. For datatype-generic programs this is provided by the theory of so-called "reductivity" [11, 12] . The 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 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, $A \subseteq S \backslash\!\!\cdot F.B$. Since, B is the maximal set of such states, A, and since the semantics defines the input-ouput 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 $\langle \mu A :: S \backslash\!\!\cdot F.A \rangle$. 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 $\langle \mu A :: S \backslash\!\!\cdot F.A \rangle = \mathsf{id}_I$. $\Box$

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

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

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 [18, 17] 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 $\mathsf{mem} :: I \leftarrow F.I$ is a membership relation of relator F if and only if $F.A = \mathsf{mem} \backslash\!\!\cdot A$ for all coreflexives A , $A \subseteq I$. $\Box$

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

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

Now, well-foundedness of relation R of type $I \leftarrow I$ is the condition that the least prefix point of the function $\langle A :: R \backslash\!\!\cdot A \rangle$ is I [10], whereas reductivity of $S :: F.I \leftarrow I$ is the condition that the least prefix point of the function

$\langle A :: S \mathbin{\backslash\!\!\cdot} F.A \rangle$ is 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,

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

Conversely,

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

Summarising, we have:

Theorem 2. 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. □

Bird and De Moor [8, 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

Theorem 3. The converse of an initial F-algebra is F-reductive.

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

$$I \subseteq A \quad \Leftarrow \quad \mathsf{in}^{\cup} \mathbin{\backslash\!\!\cdot} F.A \subseteq A \quad .$$

We start with the antecedent and derive the consequent:

$$\begin{array}{ll} & \mathsf{in}^{\cup} \mathbin{\backslash\!\!\cdot} F.A \subseteq A \\ = & \quad \{ \quad \text{for function } f \text{ and coreflexive } B,\ f \mathbin{\backslash\!\!\cdot} B = f^{\cup} \cdot B \cdot f, \\ & \qquad \mathsf{in}^{\cup} \text{ is a function and } F.A \text{ is a coreflexive} \quad \} \\ & \mathsf{in} \cdot F.A \cdot \mathsf{in}^{\cup} \subseteq A \\ \Rightarrow & \quad \{ \quad \text{hylo theorem} \quad \} \\ & (\!|\mathsf{in}|\!) \cdot (\!|\mathsf{in}|\!)^{\cup} \subseteq A \\ = & \quad \{ \quad \text{identity rule: (10), } \mathsf{in} :: I \leftarrow F.I \text{ is an initial } F\text{-algebra} \quad \} \\ & I \subseteq A \quad . \end{array}$$

□

Theorem 4. Let Q be G-reductive and S be a natural transformation of type $F \hookleftarrow \mathsf{Id}$, where Id denotes the identity relator. Then $F.Q \cdot S$ is $(F \circ G)$-reductive.

Proof We prove the stronger:

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

First, we observe a general fact about natural transformations α of type $F \hookleftarrow H$, namely, for all objects I and all coreflexives A such that $A \subseteq I$,

$$H.A \;\subseteq\; \alpha_I \mathbin{\backslash\!\!\!\cdot} F.A \qquad , \tag{12}$$

since

$$\begin{array}{ll} & H.A \subseteq \alpha_I \mathbin{\backslash\!\!\!\cdot} F.A \\ = & \quad \{ \quad \text{factors: (2)} \quad \} \\ & (\alpha_I \cdot H.A)^{<} \subseteq F.A \\ = & \quad \{ \quad \text{domains: (1)} \quad \} \\ & F.A \cdot \alpha_I \cdot H.A = \alpha_I \cdot G.A \\ = & \quad \{ \quad \alpha \text{ has type } F \hookleftarrow H \text{. Thus, } F.A \cdot \alpha_I \supseteq \alpha_I \cdot H.A \text{.} \\ & \qquad\quad A \text{ is a coreflexive, so } H.A \cdot H.A = H.A \quad \} \\ & F.A \cdot \alpha_I \cdot H.A \subseteq \alpha_I \cdot H.A \\ = & \quad \{ \quad F.A \subseteq \mathsf{id}_{F.I} \quad \} \\ & \mathsf{true} \quad . \end{array}$$

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

$$\begin{array}{ll} & (F.Q \cdot S) \mathbin{\backslash\!\!\!\cdot} F.(G.A) \\ = & \quad \{ \quad \text{factors: (2)} \quad \} \\ & S \mathbin{\backslash\!\!\!\cdot} (F.Q \mathbin{\backslash\!\!\!\cdot} F.(G.A)) \\ \supseteq & \quad \{ \quad \text{factors: (2)} \quad \} \\ & S \mathbin{\backslash\!\!\!\cdot} F.(Q \mathbin{\backslash\!\!\!\cdot} G.A) \\ \supseteq & \quad \{ \quad S \text{ has type } F \hookleftarrow \mathsf{Id} \text{, (12)} \quad \} \\ & Q \mathbin{\backslash\!\!\!\cdot} G.A \quad . \end{array}$$

□

6 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 [19] and is further elaborated in [7]. 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 [7]. Although the one presented here was the first to be developed, it was considered expedient at the time not to burden the reader of [7] 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,

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

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 $\mathsf{inl}_{A,B}$ denote the injection function of type $A+B \leftarrow A$. (We will drop subscripts from now on for simplicity.) Then we can define the relation $\mathsf{occurs\_properly\_in}$ of type $F^\star V \leftarrow F^\star V$ 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 5. The relation $\mathsf{occurs\_properly\_in}$ is well-founded.

Proof

$\mathsf{occurs\_properly\_in}$ is well-founded

$=$ { definition of $\mathsf{occurs\_properly\_in}$,
R is well-founded $\equiv$ R^+ is well-founded }

$\mathsf{mem} \cdot (\mathsf{in} \cdot \mathsf{inl})^\cup$ is well-founded

$\Leftarrow$ { $\mathsf{mem} \cdot R$ is well-founded $\equiv$ R is F-reductive }

$(\mathsf{in} \cdot \mathsf{inl})^\cup$ is F-reductive

$=$ { $(\mathsf{in} \cdot \mathsf{inl})^\cup = \mathsf{inl}^\cup \cdot \mathsf{in}^\cup$, }

$\mathsf{inl}^\cup \cdot \mathsf{in}^\cup$ is F-reductive

$\Leftarrow$ { theorem 4 }

$\mathsf{in}^\cup$ is F_V-reductive $\wedge$ $\mathsf{inl}^\cup :: F \hookleftarrow F_V$

$\Leftarrow$ { theorem 3, definition of $\hookleftarrow$ }

$\mathsf{true} \wedge \langle \forall R :: F.R \cdot \mathsf{inl}^\cup{}_J \supseteq \mathsf{inl}^\cup{}_I \cdot F_V.R \rangle$

$=$ { $F_V.A = F.A + \mathsf{id}_V$, converse and defn. of inl }

true .

□

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.

Acknowledgements

This work was supported by EPSRC grant GR/S27085/01, Data-type generic programming.

References

1. Jeuring, J., Jansson, P.: Polytypic programming. In Launchbury, J., Meijer, E., Sheard, T., eds.: Proceedings of the Second International Summer School on Advanced Functional Programming Techniques, Springer-Verlag (1996) 68–114 LNCS 1129.
2. Hinze, R.: Polytypic values possess polykinded types. Science of Computer Programming **43**(2-3) (2002) 129–159
3. Hinze, R., Jeuring, J., Löh, A.: Type-indexed data types. Science of Computer Programming **51**(1-2) (2004) 117–151
4. Löh, A., Clarke, D., Jeuring, J.: Dependency-style Generic Haskell. In Shivers, O., ed.: Proceedings of the International Conference, ICFP'03, ACM Press (2003) 141–152
5. Doornbos, H., Backhouse, R.: Induction and recursion on datatypes. In Möller, B., ed.: Mathematics of Program Construction, 3rd International Conference. Volume 947 of LNCS., Springer-Verlag (1995) 242–256
6. Doornbos, H.: Reductivity arguments and program construction. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science (1996)
7. Doornbos, H., Backhouse, R.: Reductivity. Science of Computer Programming **26**(1–3) (1996) 217–236
8. Jansson, P., Jeuring, J.: Functional pearl: Polytypic unification. Journal of Functional Programming (1998)
9. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming. An introduction. In Swierstra, S., ed.: 3rd International Summer School on Advanced Functional Programming, Braga, Portugal, 12th-19th September, 1998. Volume LNCS 1608., Springer Verlag (1999) 28–115
10. Hoare, C., He, J.: The weakest prespecification. Fundamenta Informaticae **9** (1986) 51–84, 217–252
11. Freyd, P., Ščedrov, A.: Categories, Allegories. North-Holland (1990)
12. Backhouse, R.: Naturality of homomorphisms. Lecture notes, International Summer School on Constructive Algorithmics, vol. 3, 1989 (1989)
13. Backhouse, R., Bruin, P.d., Malcolm, G., Voermans, T., Woude, J.v.d.: Relational catamorphisms. In B., M., ed.: Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications, Elsevier Science Publishers B.V. (1991) 287–318
14. Backhouse, R., Woude, J.v.d.: Demonic operators and monotype factors. Mathematical Structures in Computer Science **3**(4) (1993) 417–433
15. Bird, R.S., de Moor, O.: Algebra of Programming. Prentice-Hall International (1996)
16. Backhouse, R., Bruin, P.d., Hoogendijk, P., Malcolm, G., Voermans, T., Woude, J.v.d.: Polynomial relators. In Nivat, M., Rattray, C., Rus, T., Scollo, G., eds.: Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91, Springer-Verlag, Workshops in Computing (1992) 303–326
17. Reynolds, J.: Types, abstraction and parametric polymorphism. In Mason, R., ed.: IFIP '83. Elsevier Science Publishers (1983) 513–523
18. Wadler, P.: Theorems for free! In: 4'th Symposium on Functional Programming Languages and Computer Architecture, ACM, London. (1989)
19. Hoogendijk, P.: A Generic Theory of Datatypes. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology (1997)

20. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: FPCA '91: Functional Programming Languages and Computer Architecture. Number 523 in LNCS, Springer-Verlag (1991) 124–144
21. Meijer, E.: Calculating Compilers. PhD thesis, University of Nijmegen (1992)
22. Backhouse, R., Doornbos, H.: Mathematics of recursive program construction. Internet publication available from http://www.cs.nott.ac.uk/ rcb/MPC/papers (2001)
23. Malcolm, G.: Algebraic data types and program transformation. PhD thesis, Groningen University (1990)
24. Malcolm, G.: Data structures and program transformation. Science of Computer Programming **14**(2–3) (1990) 255–280
25. Backhouse, R., Hoogendijk, P.: Final dialgebras: From categories to allegories. Theoretical Informatics and Applications **33**(4/5) (1999) 401–426
26. Hoogendijk, P., de Moor, O.: Container types categorically. Journal of Functional Programming **10**(2) (2000) 191–225
27. Doornbos, H., Backhouse, R., van der Woude, J.: A calculation approach to mathematical induction. Theoretical Computer Science **179** (1997) 103–135
28. Jansson, P., Jeuring, J.: PolyP - a polytypic programming language extension. In: POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (1997) 470–482