Relational catamorphisms

Eindhoven University of Technology

Department of Mathematics and Computing Science

RELATIONAL CATAMORPHISMS

by

R.C.Backhouse P.J. de Bruin G.Malcolm
E.Voermans J. van der Woude

Computing Science Note 91/11
Eindhoven, July 1991

COMPUTING SCIENCE NOTES

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
Editors: prof.dr.M.Rem
prof.dr.K.M. van Hee

RELATIONAL CATAMORPHISMS

Roland C. Backhouse[*] Peter J. de Bruin[†] Grant Malcolm[‡]
Ed Voermans[*] Jaap van der Woude[*§]

Abstract

This paper reports ongoing research into a theory of datatypes based on the calculus of relations. A fundamental concept introduced here is the notion of "relator" which is an adaption of the categorical notion of functor. Relational catamorphisms are then introduced and shown to satisfy a unique extension property. Several further properties are discussed including so-called fusion properties. The paper is concluded by showing how new relators can be constructed by an appropriate choice of relational catamorphism.

[*]Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

[†]Department of Computer Science, Rijksuniversiteit Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands.

[‡]Computing Laboratory, Programming Research Group, Oxford University, 8-11 Keble Road, Oxford OX1 3QD, United Kingdom.

[§]CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands. Sponsored by NWO (NF 62.518).

1 Introduction

Since the observation was first made (e.g. by Hoare [30]) that program structure is related to data structure the notion of type has pervaded many theories of program design, so much so that in our view such a notion has become indispensable. In line with its perceived importance there is now an abundance of type theories, each drawing substance from one or more established areas of mathematics — including category theory, intuitionism and the second order lambda calculus. This paper explores yet another type theory, this time based on an axiomatic presentation of the theory of binary relations.

Our reasons for embarking on this exploration involved an element of satisfaction and an element of dissatisfaction with current programming research. The element of satisfaction comprises, first, the ever-growing knowledge and understanding of theories of type, second, the pioneering work of Bird and Meertens on economical notations for functional programming and, third, the now well-established literature on the calculation of imperative programs. The element of dissatisfaction arose from a growing frustration with the fundamental limitations of the *functional* programming paradigm within which almost all type theories have been developed up till now, and with the continuing disparity in scale between formal and informal program development. Let us begin with the element of satisfaction.

Type Theory, Category Theory and the Bird-Meertens Formalism

The history of research into type structure as it pertains to programming is something that we do not care or dare to trace. Our own understanding has, however, been substantially influenced in recent years from two directions: the work of the "intuitionists", in particular Martin-Löf [40], the Göteborg group [46] and the NuPRL group [21] on a theory of types based on the notion of "propositions-as-types" (this work now being known to have strong connections to the Automath project led by de Bruijn [18]), and the work of category theoreticians on algebraic approaches to program specification [28, 43].

Martin-Löf's theory of types can be characterised as a theory of inductively-defined types. A major attraction of his theory is that there is an elegant scheme underlying the definition of individual types that encourages and facilitates the construction of new types. A contribution of members of the current consortium was to recognise and elaborate on this scheme, leading to the publication of [6]; similar ideas have also been pursued by Dybjer [27] and others.

In the categorical approach to type structure so-called "unique extension properties" are used to characterise types as either the "initial" or "terminal" objects in a category. Hagino [29] proposed a method of type-definition based on this characterisation. Most researchers would concede that the two approaches are formally equivalent but would argue that in nature they are quite distinct, the intuitionistic approach being based on the natural-deduction style of proof development whereas the categorical approach is much more equational and often better suited to program development. On the other hand a major innovation of Martin-Löf's theory was the notion of dependent type, which notion does not seem to be so readily expressible within category theory.

Quite independently of the above work Bird and Meertens have been collaborating for many years in the development of an APL-like notation for functional programs which emphasises economy of expression and calculation. The importance of such economy to programming has been eloquently advocated by Meertens [41] and it would not do justice to his work to try to summarise the arguments here. A significant outcome, however, of this collaboration has been an impressive, albeit limited, calculus of program construction based around the notion of homomorphism on a list structure. The calculus has been used to reformulate existing solutions and to develop ingenious new solutions to many list-programming and other problems [12, 13, 15, 14, 16].

Some few years ago, research began with the aim of extending Bird and Meertens' work on lists to arbitrary, inductively-defined, data types. The conjecture we made at that time and which has since been amply confirmed was that the basic concepts and calculational techniques propounded by Bird and Meertens would be equally relevant and powerful in a more general type-theoretic setting. In the process of conducting this research we became more and more familiar with the categorical approach to type definition, and began to appreciate and further the application of unique extension properties. For accounts of this work refer to [2, 37, 36].

So much for the element of satisfaction. Now to the element of dissatisfaction.

Indeterminacy and Notational Issues

Although endowed with many mathematical niceties, there is, we believe, one overriding reason why purely-functional programming can only be a passing phase in the development of computing science: that is the lack of nondeterminism. Functions are by definition deterministic, but nondeterminism — the ability to postpone, sometimes indefinitely, decisions — has long been recognised as a vital component of any programming calculus. Indeed, the inclusion of nondeterminism is a major desideratum within calculi for imperative programming [25]. On the other hand, notions of type within imperative programming languages are grossly impoverished relative to the same notions in functional languages. Type theory has, until now, made the greatest advances within the functional programming paradigm.

In addition to our dissatisfaction with the determinism of functional programming and the type-poverty of imperative programming, we are becoming more and more distressed with what we perceive as a severe notational flaw that pervades the everyday practice of both imperative and functional programming, namely the ubiquitous use of bound variables. As a consequence formal manipulations become long and unwieldy and can indeed obscure rather than elucidate an argument. The minimisation of bound variables has, of course, long been advocated by category theory as well as being fundamental to the Bird-Meertens formalism. However, mathematical practice and programming practice lag far behind theoretical argument, and we continue to find scope for substantial economies in calculation. For more explanation and discussion of our viewpoint see [3].

So much for the element of dissatisfaction.

The Need For a Relational Framework

The relational calculus has been explored in the past as a framework for programming, for example in [9], [10], [23] and [49]. (This list is certainly by no means exhaustive.) Recently Hoare and He [32] have strongly advocated the view of specifications as relations and the programming process as that of refining a given relation into a (possibly functional) implementation. So far as we know, however, none of this research has combined the relational calculus with type theory.

The need to admit relations, rather than functions, in programming was also much in evidence at a summer school held as recently as September, 1989. At this summer school de Moor lectured on his work on applying a relational calculus to various optimisation problems [44] (such problems being by nature nondeterministic since unique optima are exceptional) and to program inversion [45] whilst Sheeran [50] and Jones [33] reported on the use of relations to describe butterfly circuits and the Fast Fourier Transform.

"Needs", "wishes" or "wouldn't-it-be-nice lists" are all very well, but the art of doing research is to recognise out of the great multitude of outstanding issues those few that can be resolved elegantly and effectively using current knowledge and techniques. The incentive for us to investigate a relational theory of types was the (re)discovery by de Bruin of the notion of "naturality" of polymorphism [19]. (As it turns out, this notion was already known to Reynolds [47] much earlier but its full relevance to program calculation does not seem to have been envisaged. De Bruin's and, more or less simultaneously, Wadler's [52] observation was that naturality of polymorphism explains and indeed predicts several of the most fundamental laws in the Bird-Meertens formalism.) In order to express the notion of "naturality" one is obliged to extend the definition of a type functor (a type constructor and corresponding "map" operator) to a mapping from *relations* to *relations.* In other words, relations are essential to meta-reasoning about polymorphic type constructors but there seems to be no reason why their use should be restricted to the meta-level. One is indeed encouraged to replace the categorical notion of "functor" by a (seemingly) stronger notion of "relator". The ideas underlying, the goals of, and preliminary justification for, a type-oriented theory of relational programming were discussed by Backhouse [1] at the above-mentioned summer school.

Relational Programming

The starting point for the present work is the (already-mentioned) notion of "relational programming" as put forward by Hoare and He [32]. In their view, *specifications* and *implementations* are binary relations on input and output values. An implementation f *satisfies* specification R if

$$f \subseteq R$$

(where a binary relation is regarded as a set of pairs). *Programming* is thus the process of calculating an implementation satisfying a given specification.

Which binary relations count as specifications is quite unrestricted: the whole of the language of mathematics may be used as specification language. Which binary relations count as implementations is fluid: the more we discover about what can and what cannot be efficiently automated the more "higher-level" our programming languages will become.

Thus the two notions of specification and implementation are deliberately left vague in order to take account of future developments.

In spite of this vagueness there is still much that can be said about what might constitute a "healthy" theory of relational programming. *Monotonicity*, for example, of the operators in one's implementation language is desirable for "compositionality" of programming: if $\otimes$ is a binary operator, say, on relations monotonicity of $\otimes$ is the statement that

$$R \otimes S \subseteq U \otimes V \quad \Leftarrow \quad R \subseteq U \wedge S \subseteq V$$

From a programming point of view this is the statement that a specification written in the form $U \otimes V$ can be implemented by finding an implementation R of U and — separately — an implementation S of V, and then composing them to form $R \otimes S$.

Given the foregoing preamble, it will come as no surprise to the reader to learn that our principal "healthiness" criterion is that the theory should support a theory of types that encourages and facilitates the introduction of new type structures. Indeed, this whole paper is devoted to the study of a general mechanism for defining a polymorphic type constructor and associated "catamorphisms" within an axiomatic theory of relations. The sort of type constructors that can be defined using this mechanism are familiar constructors like *List* and *Tree*; in this sense the paper offers no surprises. On the other hand, we do present a whole host of mathematical properties which, we argue, testify to the theory's healthiness both from a theoretical and a practical viewpoint. Moreover, we are particularly encouraged by the economy and clarity of our calculations, which is in our view of paramount importance.

The current paper is a much-abridged version of the theory that we have developed thus far [4, 7]. We begin the current paper in the following section with a summary of an axiom system for binary relations. (The system is not complete and is supplemented in [7] with axioms characterising the unit type, cartesian product and disjoint sum.) With this system as basis we build up in section 3 a vocabulary for discussing our theory. Most of the concepts and laws introduced in sections 2 and 3 can be found in one place or another in the mathematical literature and we claim no originality for their introduction. The most important concept introduced in section 3, that of "relator", does, however, appear to be novel and it is this concept that forms the backbone of our work. The main contribution of the paper begins in section 4 where we introduce and examine the properties of (relational) "catamorphisms". A specific concern in this section is to compare the properties of relational catamorphisms with functional catamorphisms (i.e. homomorphisms with domain an initial algebra). Finally, section 5 shows how parameterised types are defined and explores their junctivity properties.

We conclude this introduction with a short account of the style we use for presenting calculations.

Proof Format

For the presentation of equational proofs we use the style introduced by W.H.J. Feijen in [24]. That is, we write

$$\begin{array}{ll} & R \\ = & \quad \{\, p \,\} \\ & S \\ = & \quad \{\, q \,\} \\ & T \end{array}$$

R,S and T are expressions containing one or more free variables. p and q are most often semi-formal hints why (for all instantiations of the free variables) $R = S$ and $S = T$, respectively; in constructive proofs (discussed shortly) p and q have a formal status.

This format emphasises the transitivity of equality: all the expressions R, S and T are equal, but in particular the first and the last. We use other transitive operators in place of equality: $\equiv$ (equivalence), $\Leftarrow$ (follows from) $\Rightarrow$ (implies), $\sqsupseteq$ and $\sqsubseteq$ (the inclusion operators defined in section 3). In such cases the connectives are used *conjunctively*; for example $R \Leftarrow S \Leftarrow T$ means $(R \Leftarrow S)$ *and* $(S \Leftarrow T)$.

2 The Algebraic Framework

A major component of our endeavour is the development of a calculus of programming that permits and, indeed, encourages clear and economical calculation. For this we need an elegant algebraic setting. Although from the mathematical point of view, there is nothing wrong with a standard set-theoretic approach nor with the algebraically more attractive predicate calculus, we are dissatisfied with the persistent appearance of arguments and dummies in those systems. This invites us to look for a setting one abstraction level higher that fits our manipulative needs.

In order to choose such an abstract setting ("syntax" for short) several design criteria should be established. Here some of ours are mentioned, not as dictates but just for the sake of clarifying our point of view.

- The syntax should reflect the structure of the everyday mathematical view of relations as tightly as possible (excluding historical oddities, inelegancies and prejudice).
- The syntax should be built up in layers. If possible, those layers should be well-known syntactical unities with proven "elegance".
- The meta-language used for juggling with the syntax is the predicate calculus.
- There should be a clear distinction between terms in the meta-language and terms in the syntax.

Fortunately we don't have to start from scratch. The road towards an "axiomatic theory of relations" is already paved with the pioneering work of Tarski [51]. Besides, the above point of view is apparent in most of the curricula nowadays, be it not always explicit. Without further ado we present the most basic part of the syntax.

2.0.1 Plat Calculus and the Knaster-Tarski Theorem

Let $\mathcal{A}$ be a set, the elements of which are to be called *specs*. On $\mathcal{A}$ we impose the structure of a complete, completely distributive, complemented lattice

$$(\mathcal{A},\ \sqcap,\ \sqcup,\ \neg,\ \top\!\!\top,\ \bot\!\!\bot)$$

where "$\sqcap$" and "$\sqcup$" are associative and idempotent, binary infix operators with unit elements "$\top\!\!\top$" and "$\bot\!\!\bot$", respectively, and "$\neg$" is the unary prefix operator denoting complement (or negation). We assume familiarity with the standard definition of a lattice given, for example, by Birkhoff [17]. By "complete lattice" we mean that the extremums

$$\begin{array}{ll} & \sqcup(i:\ i\in\mathcal{I}:\ R_i) \\ \text{and} & \sqcap(i:\ i\in\mathcal{I}:\ R_i) \end{array}$$

exist for all families of specs $\{i\ :\ \ i\in\mathcal{I}\ :\ \ R_i\}$, where the index set $\mathcal{I}$ is completely arbitrary. "Completely distributive lattice" means that

$$\begin{array}{lllll} & R\ \sqcap & \sqcup(i:\ i\in\mathcal{I}:\ S_i) & = & \sqcup(i:\ i\in\mathcal{I}:\ R\sqcap S_i) \\ \text{and} & R\ \sqcup & \sqcap(i:\ i\in\mathcal{I}:\ S_i) & = & \sqcap(i:\ i\in\mathcal{I}:\ R\sqcup S_i) \end{array}$$

for all specs R and all families of specs $\{i\ :\ \ i\in\mathcal{I}\ :\ \ S_i\}$. Finally, "complemented lattice" means that $\neg R$ exists for all specs R and obeys de Morgan's laws and the double negation rule. (Note: the definition of a Boolean algebra requires only the existence of finite extremums and distributivity over such finite extremums. Our requirements are thus stronger.) The ordering relation induced by the lattice structure will be denoted by "$\sqsupseteq$".

This structure is well known from the predicate calculus: for "$\sqcap$" and "$\sqcup$" read conjunction and disjunction, respectively, for "$\top\!\!\top$" and "$\bot\!\!\bot$" read **true** and **false**, and for "$\sqsupseteq$" read "$\Leftarrow$". We call such a structure a *plat*, the "p" standing for power set and "lat" standing for lattice. Since the structure is so well known and well documented we shall assume a high degree of familiarity with it.

Among the more significant properties of such a structure is the (well-known) "Knaster-Tarski fixpoint theorem". Since we shall use the theorem frequently we summarise it here (to the extent and in the form appropriate to our own needs). Specifically, it says that, for arbitrary monotonic function θ, the equation

$$X ::\quad X = \theta.X$$

has a smallest solution, which henceforth we denote by $\mu\theta$, characterised by the two properties:

$$\mu\theta\ =\ \theta.\mu\theta$$

and, for all X,

$$X \sqsupseteq \mu\theta\ \Leftarrow\ X \sqsupseteq \theta.X$$

Moreover, such an equation also has a largest solution, which henceforth we denote by $\nu\theta$, characterised by the properties:

$$\nu\theta \;=\; \theta.\nu\theta$$

and, for all X,

$$X \sqsubseteq \nu\theta \;\Leftarrow\; X \sqsubseteq \theta.X$$

For an excellent account of plat calculus (although that name is not used!), including a modern proof of the Knaster-Tarski theorem and a clear and careful exposition of its implications, we would recommend the reader to refer to [25].

2.0.2 Composition and Factors

The second layer is the monoid structure for composition:

$$(\mathcal{A},\, \circ\,,\, I)$$

where $\circ$ is an associative binary infix operator with unit element I.

The interface between these two layers is: $\circ$ is coordinatewise universally "cup-junctive". I.e. for $V, W \subseteq \mathcal{A}$,

$$(\sqcup V) \circ (\sqcup W) \;=\; \sqcup(P,Q:\; P \in V \wedge Q \in W:\; P \circ Q)$$

In particular,

- $\perp\!\!\perp$ is a left and right zero for $\circ$,
- $\circ$ is monotonic with respect to $\sqsupseteq$.
- $\top\!\!\top \circ \top\!\!\top \;=\; \top\!\!\top$.

Another, less immediate and somewhat unfamiliar consequence of this interface, is the existence of so-called "left" and "right factors" defined as follows.

Definition 1 For specs R and S we define the *right factor* $R\backslash S$ by

(a) $\quad R\backslash S \sqsupseteq X \;\equiv\; S \sqsupseteq R \circ X$

and the *left factor* S/R by

(b) $\quad S/R \sqsupseteq X \;\equiv\; S \sqsupseteq X \circ R$

□

Left and right factors are thus defined to be the largest solutions to inequations in a variable X (the inequation to the right of the equivalence in their respective definitions). Although we shall have no use for it here we mention that the operators "$\backslash$" and "$/$" associate with each other (i.e. $P\backslash(Q/R) = (P\backslash Q)/R$), thus justifying writing $P\backslash Q/R$ and that such is a *factor* of Q.

Equations (1a) and (1b) are instances of what are known as "Galois connections". (See e.g. [35], in particular exercise 1 on p.15.). Our use of the word "factor" is intended to suggest an analogy between composition and multiplication, and between factoring and division. This analogy is further reinforced by the following easily derived *cancellation properties* of factors.

Lemma 2

(a) $S \sqsupseteq R \circ (R \backslash S)$
(b) $S \sqsupseteq (S/R) \circ R$

$\Box$

Evidence for the claim that definitions (1a) and (1b) and, in particular, the calculational possibilities they admit are important but not well known is the fact that they have surfaced in various guises and under various names over the last fifty years beginning, to our knowledge, with [26] (under the names left and right "residuals") and involving diverse application areas such as the structure of natural language [34], regularity properties of generalised-sequential machines [22] (under the name used here of left and right "factors"), the well-known Knuth-Morris-Pratt string searching algorithm [8], and program specification [32] (under the names "weakest pre- and post-specification"). We prefer Conway's [22] more anonymous terminology to that used by Hoare and He [32]. The term "residual", which is also used by Birkhoff [17], would have been equally acceptable. Note, however, that of the above-referenced works, Hoare and He's calculational formulation of the properties of "factors" is the single most significant contribution to the present work.

Remark In addition to the use of different terminology our choice of notation is exactly opposite to Hoare and He's: they would write S/R where we write $R\backslash S$, and vice-versa $R\backslash S$ where we write S/R. Our own choice of notation is justified by the — for us very important — property that in the use of (2a) and (2b) the "cancelled" expressions are adjacent. We reject outright the notation adopted by Birkhoff [17] as unsystematic and inappropriate to compact calculation. *End of Remark*

2.0.3 Reverse

The third layer is the "reverse structure",

$$(\mathcal{A}, \,^{\cup})$$

where "$^{\cup}$" is a unary postfix operator such that it is its own inverse.

The interface with the first layer is that "$^{\cup}$" is an isomorphism of plats. I.e. for all $P, Q \in \mathcal{A}$,

$$P \sqsupseteq Q \;\equiv\; P^{\cup} \sqsupseteq Q^{\cup}$$

Consequently, for all $P, Q \in \mathcal{A}$,

$$\begin{array}{rcl} \neg(P^{\cup}) & = & (\neg P)^{\cup} \\ (P \sqcup Q)^{\cup} & = & P^{\cup} \sqcup Q^{\cup} \\ (P \sqcap Q)^{\cup} & = & P^{\cup} \sqcap Q^{\cup} \\ \top\!\!\top^{\cup} & = & \top\!\!\top \\ \bot\!\!\bot^{\cup} & = & \bot\!\!\bot \end{array}$$

Remark As a rule we shall write the names of unary functions as prefixes to their arguments. A partial justification for making an exception of "$\cup$" is that it commutes with "$\neg$", thus permitting us to write the syntactically ambiguous "$\neg R\cup$". Later we shall see that "$\cup$" also commutes (by definition) with so-called "relators". The latter is the main reason for this choice of notation.
End of Remark

The interface with the second layer is formed by the two rules

$$(R \circ S)\cup \quad = \quad S\cup \circ R\cup$$

and

$$I\cup \quad = \quad I$$

2.0.4 Operator precedence

Some remarks on operator precedence are necessary to enable the reader to parse our formulae. First, as always, operators in the metalanguage have lower precedence than operators in the object language. The principle meta-operators we use are equivalence ("$\equiv$"), implication ("$\Rightarrow$") and follows-from ("$\Leftarrow$") — these all having equal precedence — , together with conjunction ("$\wedge$") and disjunction ("$\vee$") — which have equal precedence higher than that of the other meta-operators. The precedence of the operators in the plat structure follows the same pattern. That is, "$=$", "$\sqsupseteq$" and "$\sqsubseteq$" all have equal precedence; so do "$\sqcup$" and "$\sqcap$"; and, the former is lower than the latter. Composition ("$\circ$") has a yet higher precedence than all of the operators mentioned thus far, whilst the two factoring operators ("/" and "\") have the highest precedence of all the binary operators. Finally, all unary operators in the object language, whether prefix or postfix, have the same precedence which is the highest of all. Parentheses will be used to disambiguate expressions where this is necessary.

2.0.5 The RS and Rotation Rules

To the above axioms we now add an axiom that acts as an interface between all three layers.

The RS Rule

$$\neg Y \;\sqsupseteq\; P \circ \neg X \circ Q \quad \equiv \quad X \;\sqsupseteq\; P\cup \circ Y \circ Q\cup$$

The name "RS" is a mnemonic for "*R*otation and *S*hunting". The "rotation rule" is obtained by making the substitutions $Y := R\cup$, $P := S$, $X := \neg T$ and $Q := I$ and simplifying using the properties of I, reverse and complement.

Rotation Rule

$$\neg R\cup \;\sqsupseteq\; S \circ T \quad \equiv \quad \neg T\cup \;\sqsupseteq\; R \circ S$$

(Note how the variables R, S and T are rotated in going from the left to the right side of the rule.) "Shunting" is the name given by Dijkstra and Scholten [25] to an important rule in the predicate calculus. Specifically, by making the substitutions $Y := U$, $P := I$, $X := V$, and $Q := W$ and simplifying we obtain the rule

$$\neg U \ \sqsupseteq\ \neg V \circ W \quad \equiv \quad V \ \sqsupseteq\ U \circ W \cup$$

Interpreting "$\circ$" as conjunction, "$\cup$" as the identity function, and "$\sqsupseteq$" as follows-from this is the afore-mentioned shunting rule.

It is our experience that the RS rule can meet with considerable resistance for one of two reasons. First, for calculational purposes, a rule with four free variables is (rightly) regarded as approaching, if not outwith, the limits of useability. Second, for those already familiar with the relational calculus, there is resistance to the fact that we have chosen to replace the better known "Schröder" rule which states that the following three statements are all equivalent.

$$\begin{array}{l} R \circ S \ \sqsubseteq\ T \\ R\cup \circ \neg T \ \sqsubseteq\ \neg S \\ \neg T \circ S\cup \ \sqsubseteq\ \neg R \end{array}$$

(See, for example, [48] for historical references.) To counter these arguments we would point out that the RS rule is more compact than the Schröder rule (two statements are equivalent rather than three) and, more importantly, has a clean syntactic form that makes it easy to remember and to apply. The rotation rule shares these advantages as well as involving only three free variables, but suffers the disadvantage that in some calculations two successive uses are required where only one use of the RS rule is necessary. In combination with other laws both rules are equivalent to the Schröder rule. (The Schröder rule can also be reduced to the equivalence of just two statements, making our first argument void, but then it would suffer the same disadvantage as the rotation rule, which is probably the reason why it is always stated in the way that it is.)

2.1 Models

Various models of the above axioms are discussed in [4] with regard to the following questions:

(a) Are the layers and axioms independent?

(b) Are the successive extensions conservative?

(c) Does the axiomatisation characterise the set-theoretic relations completely?

Here we shall content ourselves with a summary of the conclusions, namely: the set-theoretic relations do indeed form a model of the axiom system but the axiom system is not complete for this model; the RS and cone rules are independent of the other axioms but the reverse structure is not.

A final comment with regard to the idiosyncracies of our naming conventions. The following sections must serve a dual purpose. The technical aim is to build up a theory of types based upon the above syntax. To do this in a way that is evidently free from logical inconsistencies necessitates making a clear distinction between the theory itself and the metalanguage. For this reason we have chosen to call elements of $\mathcal{A}$ "specs" rather than "relations" and to use the symbols "$\sqcap$" and "$\sqcup$" etc. rather than "$\cap$" and "$\cup$" etc. To serve the second purpose we intersperse the development with references to the relational model. The reader may prefer to construct their own proofs of the various lemmas, theorems etc. in this one interpretation, but they do so at their own peril.

3 Foundations

The purpose of this (abridged) section is to build up a vocabulary for our later discussion of the properties of catamorphisms. In order to avoid confusion with existing terminology we make a complete reappraisal of what is meant by "type", "function", "type constructor" etc. Nevertheless, it should be emphasised that — with the important exception of the notion of "relator" — the concepts defined here are amply documented in the mathematical literature and we make no claim to originality.

3.1 Monotypes

We say that spec A is a *monotype* iff $I \sqsupseteq A$.

In the relational model, for example, we may assume that the universe $\mathbb{U}$ contains two unequal values **true** and **false**. The monotype $\mathbb{B}$ of booleans is then defined to be the relation

$$\{(\mathbf{true}, \mathbf{true}), (\mathbf{false}, \mathbf{false})\}$$

Note that for monotypes A and B

$$(3) \qquad A \;=\; I \sqcap A \;=\; A\cup \;=\; A \circ A$$

$$(4) \qquad A \circ B \;=\; B \circ A \;=\; A \sqcap B$$

Properties such as (3) and (4) stated here without proof are proven in [4].

We often write

$$R \in S {\sim} T$$

as a synonym for

$$(5) \qquad S \circ R = R = R \circ T$$

Note that (5) defines $S{\sim}T$ to be a subset of $\mathcal{A}$. Typically S and T will be monotypes, but we prefer not to complicate the definition by making such a restriction.

3.2 Imps and Co-imps

In this subsection we define "imps" and "co-imps" as special classes of specs. In the relational model an "imp" is a function.

Definition 6

(a) A spec f is said to be an *imp* if and only if $I \sqsupseteq f \circ f\cup$.

(b) A spec f is said to be a *co-imp* if and only if $f\cup$ is an imp.

□

The intended interpretation is that an "imp" is an "imp"lementation. On the other hand, it is not the intention that all implementations are "imps". Apart from their interpretation imps have an important distributive property not enjoyed by arbitrary specs, namely:

Theorem 7 If f is an imp then, for all non-empty sets of specs V,

$$\sqcap(P: \ P \in V: \ P) \circ f \ = \ \sqcap(P: \ P \in V: \ P \circ f)$$

In particular, for all specs R and S,

$$(R \sqcap S) \circ f \ = \ (R \circ f) \sqcap (S \circ f)$$

□

Dually we have:

Theorem 8 If f is a co-imp then, for all non-empty sets of specs V,

$$f \circ \sqcap(P: \ P \in V: \ P) \ = \ \sqcap(P: \ P \in V: \ f \circ P)$$

In particular, for all specs R and S,

$$f \circ (R \sqcap S) \ = \ (f \circ R) \sqcap (f \circ S)$$

□

In the relational model a monotype is the identity function on that type. More generally, the requirement of being a function is the requirement of being single-valued on some subset of $\mathbb{U}$, the so-called "domain" of the function. The domain and range are made explicit in the following.

Definition 9 For monotypes A and B we define the set $A \longleftarrow B$ by $f \in A \longleftarrow B$ whenever

(a) $f \circ B = f$

(b) $f\cup \circ f \ \sqsupseteq \ B$, and

(c) $A \ \sqsupseteq \ f \circ f\cup$

The nomenclature "$f \in A \longleftarrow B$" is verbalised by saying that "*f is an imp to A from B*".
□

In terms of the relational model, property (9a) expresses the statement that the domain of definition of f is confined to B. Property (9b) expresses the statement that f is ***total*** on domain B, i.e. for each $x \in B$ there is at least one y such that $y \langle f \rangle x$; finally, property (9c) expresses the statement that f maps elements of B to A and that f is ***single-valued***, i.e. for each $x \in B$ there is at most one y such that $y \langle f \rangle x$. . Their combination justifies writing "$f.x$", for each $x \in B$, denoting the unique object y in A such that $y \langle f \rangle x$.

By including the above definition and not simultaneously including a dual notion for co-imps we have introduced an asymmetry into our theory that until now has been totally absent. This expresses a slight bias with an eye to the extension of the theory with cartesian product and disjoint sum later in this section. We hasten to add, nonetheless, that there is no such asymmetry in the theory at this instant and every property we state for imps alone has a dual property for co-imps.

To avoid repeating assumptions and to assist the reader's understanding we continue to use the conventions that capital letters $A, B, C, \ldots$ at the beginning of the alphabet denote monotypes, small letters $f, g, h, \ldots$ denote imps or co-imps, and capital letters $R, S, T, \ldots$ at the end of the alphabet denote arbitrary specs.

Finally, let us remark that the unconventional direction of the arrow in the statement "$f \in A \longleftarrow B$" is entirely dictated by the choice to denote function application with the function name to the left of its argument. (We owe the suggestion to deviate from convention to Meertens [42].)

3.3 Relators

In categorical approaches to type theory a parallel is drawn between the notion of type constructor and the categorical notion of "functor", thereby emphasising that a type constructor is not just a function from types to types but also comes equipped with a function that maps arrows to arrows. For an informative account of this parallel see, for example, [39]. In this subsection we propose a modest extension to the notion of functor to which we give the name "relator".

Definition 10 A *relator* is a function, F, from specs to specs such that

(a) $\quad I \sqsupseteq F.I$

(b) $\quad R \sqsupseteq S \;\Rightarrow\; F.R \sqsupseteq F.S$

(c) $\quad F.(R \circ S) \;=\; F.R \circ F.S$

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

□

In view of (10d) we take the liberty of writing simply "$F.R^{\cup}$" without parentheses, thus avoiding explicit use of the property.

The above ostensibly defines a *unary* relator but we also wish to allow it to serve as the definition of a relator mapping an m-ary *vector* of specs into an n-ary *vector* of specs, for some natural numbers m and n. (This is necessary in order to allow the theory to encompass what are variously called "mutually recursive type definitions" and "many-sorted algebras". More generally, there is no reason why "m" and "n" may not be some fixed but nevertheless arbitrary index sets. However, such a generalisation would complicate the current discussion more than we deem justified.) The mechanism by which we can do this is to assume that all the constants appearing in the definition ("$=$", "$\sqsupseteq$", "I", "$\circ$" and "$\cup$") are silently "lifted" to operate on vectors. For example, if F maps m-ary vectors into n-ary vectors, property (10c) would be written out in the form

$$(F.(R_1 \circ S_1, \ldots, R_m \circ S_m))_j \;=\; (F.(R_1, \ldots, R_m))_j \circ (F.(S_1, \ldots, S_m))_j$$

for all j, $1 \leq j \leq n$, whereby the use of subscripts denotes projection of a vector onto one of its components. It is, however, just such clumsy expressions that we want to avoid.

There are two cases that we make particular use of. The first case has to do with taking fixed points of relators where an obvious requirement is that the arity of the domain vector of the relator is identical to that of its range vector. We call such a relator an *endorelator* in conformance with the terminology "endofunctor" used in category theory. The second case is when F maps a pair of (vectors of) specs into a (vector of) spec(s). We refer to such relators as *binary* relators and choose to denote them by infix operators. Thus, if $\otimes$ denotes a binary relator, its defining properties would be spelt out as follows.

(a) $\quad I \;\sqsupseteq\; I \otimes I$

(b) $\quad R \sqsupseteq S \;\wedge\; U \sqsupseteq V \;\;\Rightarrow\;\; R \otimes U \sqsupseteq S \otimes V$

(c) $\quad (R \circ S) \otimes (U \circ V) \;=\; (R \otimes U) \circ (S \otimes V)$

(d) $\quad (R\cup) \otimes (S\cup) \;=\; (R \otimes S)\cup$

The notational advantage of writing "$\cup$" as a postfix to its argument is, of course, lost in this case.

A property such as (c) we call an "abide" law; we also often refer to this law by saying that binary relators "abide" with composition. The name was coined by Richard Bird (in a different context). His motivation for the name was that it is short for "above/beside" reflecting the following two-dimensional formulation of the law in which the relator and composition are either above or beside each other.

$$\begin{array}{ccccccc} R & \circ & S & & R & & S \\ & \otimes & & = & \otimes & \circ & \otimes \\ U & \circ & V & & U & & V \end{array}$$

(Our first encounter with a two-dimensional depiction of an abide law was in [31]. In the category theory literature the term "interchange" rule (or law) is used.)

The following theorem allows a comparison to be made with our definition of "relator" and the definition of "functor" (in the category of sets).

Theorem 11 If F is a relator then

(a) $\quad A$ is a monotype $\;\Rightarrow\;$ $F.A$ is a monotype

(b) $\quad f$ is an imp $\;\Rightarrow\;$ $F.f$ is an imp

(c) $\quad f$ is a co-imp $\;\Rightarrow\;$ $F.f$ is a co-imp

(d) $\quad f \in A \longleftarrow B \;\Rightarrow\; F.f \in F.A \longleftarrow F.B$

(e) $\quad R \in A \sim B \;\Rightarrow\; F.R \in F.A \sim F.B$

Proof
See the complete paper. □

4 Initial Datatypes and Relational Catamorphisms

A fundamental argument for the use of type information in the design of large programs is that the structure of the program is governed by the structure of the data. A well-established example is the use of recursive descent to structure the parsing (and compilation) of strings defined by a context-free grammar; here the structure of the data is defined by its grammar and the structure of the parsing program is identical. The idea is extended in the denotational description of programming languages where a fundamental initial step is the definition of so-called domain equations; those familiar with denotational semantics know that once this step has been taken the later steps are often relatively mundane and straightforward. Users of strongly-typed languages like Pascal will argue strongly that the effective use of type declarations is extremely important for subsequent program development, and even users of untyped languages like Lisp will admit that the programming errors that they make are often caused by type violations. A fundamental goal of our research is therefore to develop calculi of program construction that lay bare the oneness of program and data structure.

An example of a programming formalism in which this oneness plays the rôle of a major design principle is the theory of types developed by Martin-Löf. In this theory each type is defined by four sets of rules one of which is the set of so-called introduction rules and another is a singleton set containing the so-called elimination rule for the type. (The remaining sets are not relevant to the present discussion.) The introduction rules describe the structure of the elements of the type whereas the elimination rule says how to construct functions over the elements of the type. As has been argued elsewhere [6], the introduction rules completely define the type in the sense that all other rules (including the elimination rule) are systematically derived from them. The elimination rule thus expresses the notion that "nothing else" is in the type other than the elements that can be constructed via the introduction rules by stating that the structure of functions on elements of the type is completely governed by the structure of these rules.

In the algebraic approach that we are currently pursuing a different (although formally equivalent) approach is taken to the definition of data types and in particular to expressing the notion that "nothing else" is in the type other than the elements constructed via its introduction rules. Nevertheless, the underlying principle is that a data type is a structured set of elements that is equipped with a mechanism governed by that structure for defining functions on the elements of the type. For the benefit of readers who may not be familiar with it we now outline this approach as it pertains to functional programming. Other readers will probably wish to skip the next two paragraphs; all they need to know is that we use the term "catamorphism" to refer to F-homomorphisms whose domain is an initial F-algebra. (We are currently in the process of extending our work to terminal algebras but none of that work is reported here.)

The approach involves several stages building up to the definition of a "universal object" in a category of algebras. First, in place of the introduction rules in Martin-Löf's system the notion of endofunctor is of paramount importance. An endofunctor is (in this context) a pair of functions, one from types to types and the other from functions to functions. Typically, both functions are denoted by the same symbol. Suppose F is an endofunctor, A and B are types and f and g are functions of composable type. Let I_A

denote the identity function on the type A. Then it is required that

$$
\begin{array}{lrcl}
 & F.f \in F.A \longleftarrow F.B & \Leftarrow & f \in A \longleftarrow B \\
 & F.I_A & = & I_{F.A} \\
\text{and} & F.(f \circ g) & = & F.f \circ F.g
\end{array}
$$

Without seeing some examples it is difficult for the uninitiated to envisage the correspondence between a number of introduction rules and an endofunctor. For the moment let us just remark that typically an endofunctor will take the form of a disjoint sum of other more primitive functors, and that each term in such a sum corresponds to one introduction rule. The next step is to define an F-algebra as a pair consisting of a type A and a function $f \in A \longleftarrow F.A$. (Note that if, indeed, the endofunctor F is a disjoint sum of other functors then the function f can be broken down into distinct components each being applicable to elements introduced by one of the corresponding introduction rules.) The data type defined by the endofunctor F is then an F-algebra satisfying a so-called "universal property", namely that there is a unique homomorphism from the data type to each F-algebra. Such homomorphisms take the place of the eliminators in Martin-Löf's theory. To emphasise their special rôle we shall give them the name "catamorphism".

An example would be the data type natural number. Roughly speaking, $\mathbb{N}$ has the property

$$\mathbb{N} \quad = \quad \{0\} + \mathbb{N}$$

where "+" denotes the disjoint sum of two types. (According to this definition the elements of $\mathbb{N}$ are $\hookrightarrow.0$ and $\hookleftarrow.n$ where n ranges over $\mathbb{N}$ and $\hookrightarrow$ and $\hookleftarrow$ denote the injection functions associated with disjoint sum. You should interpret "$\hookrightarrow.0$" as zero and "$\hookleftarrow$" as the successor function. More formally, we recognise in this equation an endofunctor "$\{0\}+$". This is a function that maps the type A to the type $\{0\} + A$. But it may also be extended to map functions to functions by defining $\{0\} + f$ to be that function g such that $g \circ \hookrightarrow$ is the constant function always returning $\hookrightarrow.0$, and $g \circ \hookleftarrow \; = \; \hookleftarrow \circ f$. (Moreover, it satisfies all the properties required of a functor, but that we leave to the reader to verify.) A $\{0\}+$-algebra is a set together with a constant and a unary operator (these being zero and the successor function in the case of the natural numbers), and a $\{0\}+$-homomorphism is just what one would normally understand by a homomorphism of an algebraic structure, in this case a function ϕ, say, from one $\{0\}+$-algebra (A, a, σ), say, to another (B, b, τ), say, that maps the constant of the first to the constant of the second

i.e. $\quad \phi.a \quad = \quad b$

and commutes with the unary operator of the first replacing it with that of the second

i.e. $\quad \phi \circ \sigma \quad = \quad \tau \circ \phi$

That $\mathbb{N}$ is "universal" in the class of $\{0\}+$-algebras just means that for any $\{0\}+$-algebra (A, a, σ), say, there is a unique homomorphism mapping $\mathbb{N}$ to A. With a suitable definition of the operators it is also easily shown that $\{0\} + \mathbb{N}$ is a $\{0\}+$-algebra satisfying the universality property. Thus, $\mathbb{N}$ is a fixed point of the endofunctor $\{0\}+$ in the sense that there are homomorphisms mapping $\mathbb{N}$ to $\{0\} + \mathbb{N}$ and vice-versa which (on account of their uniqueness) are each others' inverses.

To summarise this discussion: in the framework of functional programming datatypes are fixed points of endofunctors on which are defined what we call "catamorphisms", i.e. homomorphisms satisfying a uniqueness and universality property. This is not the place to discuss the practicality of catamorphisms as a program structuring method, that being something that we intend to address in future publications. We hope however that we have provided sufficient background to motivate the calculations that follow in this section. Specifically, we explore the extension of the notion of a (functional) catamorphism to relations. For this we need the notion of endorelator instead of endofunctor. We begin by discussing the least fixed point of an endorelator and then introduce our definition of a (relational) catamorphism.

From now on we assume that F is an endorelator.

4.1 Initial Datatypes

Since endorelators are, by definition, monotonic the Knaster-Tarski theorem asserts the existence of their fixed points, in particular least and greatest. We hope shortly to report on our work on greatest fixed points but in the present paper we restrict our attention to least fixed points. Specifically, the least fixed point of the endorelator F, here denoted by μF, has the defining properties

(12) $\quad \mu F \;=\; F.\mu F$

and, for all X,

(13) $\quad X \;\sqsupseteq\; \mu F \quad \Leftarrow \quad X \;\sqsupseteq\; F.X$

We shall refer to (13) as the *induction principle.*

The following lemma is about all we can say about μF at this stage. Nevertheless, it is a necessary first step.

Lemma 14 μF is a monotype.

Proof

$$\begin{array}{ll}
 & \mu F \text{ is a monotype} \\
\equiv & \quad \{ \text{ definition } \} \\
 & I \;\sqsupseteq\; \mu F \\
\Leftarrow & \quad \{ \text{ induction principle (13) } \} \\
 & I \;\sqsupseteq\; F.I \\
\equiv & \quad \{ \; F \text{ is a relator (10a) } \} \\
 & \textbf{true}
\end{array}$$

□

4.2 Catamorphisms Defined

Definition 15 For endorelator F we define a function, denoted by $(\!|F;\ \_\ |\!)$, by the properties that, for all specs R,

(a) $\quad (\!|F;\ R|\!) \ = \ R \circ F.(\!|F;\ R|\!)$

and for all specs R and X,

(b) $\quad X \ \sqsupseteq \ (\!|F;\ R|\!) \ \Leftarrow \ X \ \sqsupseteq \ R \circ F.X$

□

In other words, $(\!|F;\ R|\!)$ is the smallest solution to the equation

$$X :: \qquad X \ = \ R \circ F.X$$

Its well-definedness is thus guaranteed by the Knaster-Tarski theorem.

We call specs of the form $(\!|F;\ R|\!)$ *catamorphisms* (or *F-catamorphisms* when we particularly wish to be explicit about F) and we verbalise $(\!|F;\ R|\!)$ as "(F-)catamorphism R", omitting the qualification "F" when there is no doubt about the relator in question.

For reasons that will only become clear later, we call (15a) the *computation rule* for catamorphisms.

One may well raise one's eyebrows at the unconventional "banana brackets" we have chosen to denote catamorphisms. The reasoning behind this choice is not evident in this paper but is based on envisaged applications: typically, the relators one encounters in programming problems are formed using the disjoint-sum operator. Consequently, the catamorphism constructor will be applied to a disjoint sum of specs having the same number of components as that of the associated relator. It is our practice, therefore, to separate the components using commas and to delimit the extent of the vector formed in this way by the special brackets. For several examples of such applications see [36] and [38].

The catamorphism $(\!|F;\ I|\!)$ is of particular importance since it is clearly the least fixed point of F. Thus, we have:

(16) $\quad \mu F \ = \ (\!|F;\ I|\!)$

From now on we omit the argument "F" within the catamorphism brackets and write just "$(\!|R|\!)$" instead of "$(\!|F;\ R|\!)$".

4.3 The Unique Extension Property

The definition of a catamorphism is clear enough but with its two distinct parts it is not well-suited to calculational purposes. We proceed now to prove two properties that predict a single-statement definition of catamorphism. The first is simple enough.

Theorem 17

$$(\!|R|\!) \ = \ (\!|R|\!) \circ \mu F$$

Proof

$$\begin{array}{ll}
& (\!|R|\!) \;=\; (\!|R|\!) \circ \mu F \\
\equiv & \quad \{ \text{ lemma 14 } \} \\
& (\!|R|\!) \circ \mu F \;\sqsupseteq\; (\!|R|\!) \\
\Leftarrow & \quad \{ \text{ definition of catamorphism, specifically (15b) } \} \\
& (\!|R|\!) \circ \mu F \;\sqsupseteq\; R \circ F.((\!|R|\!) \circ \mu F) \\
\equiv & \quad \{ \; F \text{ is a relator (10c), } \mu F \text{ is a fixed point of } F \; \} \\
& (\!|R|\!) \circ \mu F \;\sqsupseteq\; R \circ F.(\!|R|\!) \circ \mu F \\
\equiv & \quad \{ \text{ definition of catamorphism, specifically (15a) } \} \\
& \mathbf{true}
\end{array}$$

□

We have now established that $(\!|R|\!)$ satisfies two equations, namely,

$$\begin{array}{llcl}
& X :: \quad & X = R \circ F.X \\
\text{and} & X :: \quad & X = X \circ \mu F
\end{array}$$

Obviously, therefore, it also satisfies the third equation

(18) $\quad X :: \quad X \;=\; R \circ F.X \circ \mu F$

The important insight contained in the next theorem is that the set of specs simultaneously solving the first two equations is identical to the set of solutions of the third equation.

Theorem 19

$$X = R \circ F.X \circ \mu F \quad\equiv\quad X = X \circ \mu F \;\wedge\; X = R \circ F.X$$

Proof

$$\begin{array}{ll}
& X \circ \mu F = X \;\wedge\; X = R \circ F.X \\
\equiv & \quad \{ \text{ substitution } \} \\
& X \circ \mu F = X \;\wedge\; X = R \circ F.(X \circ \mu F) \\
\equiv & \quad \{ \; F \text{ is a relator (10c), } \mu F \text{ is a fixed point of } F \; \} \\
& X \circ \mu F = X \;\wedge\; X = R \circ F.X \circ \mu F \\
\equiv & \quad \{ \Leftarrow \text{ is obvious; } \Rightarrow \text{ by (14) and (3) } \} \\
& X = R \circ F.X \circ \mu F
\end{array}$$

□

More significantly, equation (18) has a *unique* solution, which we shall now prove. It will come as no surprise that a goal in our proof is to invoke the induction principle (13). How we do so is, in our view, particularly elegant and offers an excellent illustration of the benefits to be gained from a systematic development of a theory taking account of clearly stated calculational rules, in this case the Galois connection between factors and composition.

Suppose P and Q are two solutions to (18). I.e.

$$
\begin{array}{llll}
(20) & P & = & R \circ F.P \circ \mu F \\
(21) & Q & = & R \circ F.Q \circ \mu F
\end{array}
$$

Since P and Q are completely symmetrical our task reduces to showing that $Q \sqsupseteq P$. We use factor theory and the induction principle to prove this property as follows.

$$
\begin{array}{ll}
 & Q \sqsupseteq P \\
\equiv & \quad \{ P = \{ (20), \text{theorem } 19 \} P \circ \mu F \} \\
 & Q \sqsupseteq P \circ \mu F \\
\equiv & \quad \{ (1a) \} \\
 & P \backslash Q \sqsupseteq \mu F \\
\Leftarrow & \quad \{ \text{induction principle } (13) \} \\
 & P \backslash Q \sqsupseteq F.(P \backslash Q) \\
\equiv & \quad \{ (1a) \} \\
 & Q \sqsupseteq P \circ F.(P \backslash Q) \\
\equiv & \quad \{ (20), (21), \text{theorem } 19 \} \\
 & R \circ F.Q \sqsupseteq R \circ F.P \circ F.(P \backslash Q) \\
\Leftarrow & \quad \{ F \text{ is a relator, monotonicity of composition} \} \\
 & F.Q \sqsupseteq F.(P \circ P \backslash Q) \\
\equiv & \quad \{ (2a), \text{monotonicity of relators} \} \\
 & \textbf{true}
\end{array}
$$

In conclusion we have:

Corollary 22 (Unique Extension Property)
For all specs X and R,

$$X = (\!|R|\!) \equiv X = R \circ F.X \circ \mu F$$

□

4.4 Consequences of the UEP

Were we obliged to refer to one theorem in the paper that is the most important of all then it would be the above unique extension property. It will be used so often below that we will refer to it within proof hints simply as "uep". A first example is the simple but nevertheless useful identity rule:

Theorem 23 (Identity Rule)

$$\mu F = (\!|\mu F|\!)$$

Proof

$$
\begin{array}{ll}
 & \mu F \;=\; (\!|\mu F|\!) \\
\equiv & \quad \{\text{ uep }\} \\
 & \mu F \;=\; \mu F \circ F.\mu F \circ \mu F \\
\equiv & \quad \{\text{ (12), (14) and (3) }\} \\
 & \textbf{true}
\end{array}
$$

□

Also, the coincidence in $(\!|R|\!)$ of the least and greatest solutions of (18) together with the Knaster-Tarski theorem gives:

Theorem 24

(a) $X = (\!|R|\!) \;\Leftarrow\; X = R \circ F.X \circ \mu F$

(b) $X \sqsupseteq (\!|R|\!) \;\Leftarrow\; X \sqsupseteq R \circ F.X \circ \mu F$

(c) $X \sqsubseteq (\!|R|\!) \;\Leftarrow\; X \sqsubseteq R \circ F.X \circ \mu F$

□

A corollary of the above that figures very prominently in program calculations is

Corollary 25 (Fusion Properties)

(a) $U \circ (\!|V|\!) = (\!|R|\!) \;\Leftarrow\; U \circ V = R \circ F.U$

(b) $U \circ (\!|V|\!) \sqsupseteq (\!|R|\!) \;\Leftarrow\; U \circ V \sqsupseteq R \circ F.U$

(c) $U \circ (\!|V|\!) \sqsubseteq (\!|R|\!) \;\Leftarrow\; U \circ V \sqsubseteq R \circ F.U$

Proof Let $\trianglelefteq \in \{=, \sqsupseteq, \sqsubseteq\}$. Then

$$
\begin{array}{ll}
 & U \circ (\!|V|\!) \;\trianglelefteq\; (\!|R|\!) \\
\Leftarrow & \quad \{\text{ (24) }\} \\
 & U \circ (\!|V|\!) \;\trianglelefteq\; R \circ F.(U \circ (\!|V|\!)) \circ \mu F \\
\equiv & \quad \{\text{ uep; (10c) }\} \\
 & U \circ V \circ F.(\!|V|\!) \circ \mu F \;\trianglelefteq\; R \circ F.U \circ F.(\!|V|\!) \circ \mu F \\
\Leftarrow & \quad \{\text{ invariance of } \trianglelefteq \text{ under composition }\} \\
 & U \circ V \;\trianglelefteq\; R \circ F.U
\end{array}
$$

□

We call the above properties "fusion properties" because during their use as left-to-right rewrite rules a spec is "fused" with a catamorphism to form a catamorphism. (Analogous to our fusion properties are so-called "loop fusion" properties.) Note, however, that we do not always use the rules to "fuse" specs; just as often we use them to "defuse" a spec into component parts. The reader should not allow the one-way character of the name to prejudice their use of such rules.
Note: Previously (e.g. [5] and [37]) we used instead the term "promotion property" in order to establish the connection with a technique used by Bird [11] for improving the efficiency of functional programs.

Theorem 26 (Monotonicity)

$$(\!|R|\!) \sqsupseteq (\!|S|\!) \quad \Leftarrow \quad R \sqsupseteq S$$

Proof

$$\begin{array}{ll}
 & (\!|R|\!) \sqsupseteq (\!|S|\!) \\
\equiv & \quad \{ \text{ fusion (25b), } U := I \ \} \\
 & I \circ R \sqsupseteq S \circ F.I \\
\Leftarrow & \quad \{ \text{ (10a) } \} \\
 & R \sqsupseteq S
\end{array}$$

□

Theorem 27 (Imp and co-imp preservation)
The function $(\!|\_|\!)$ respects (a) imps and (b) co-imps

Proof

(a)
$$\begin{array}{ll}
 & I \ \sqsupseteq \ (\!|f|\!) \circ (\!|f|\!)^{\cup} \\
\equiv & \quad \{ \text{ left factors (1b) } \} \\
 & I/(\!|f|\!)^{\cup} \ \sqsupseteq \ (\!|f|\!) \\
\Leftarrow & \quad \{ \text{ (24b) } \} \\
 & I/(\!|f|\!)^{\cup} \ \sqsupseteq \ f \circ F.(I/(\!|f|\!)^{\cup}) \circ \mu F \\
\equiv & \quad \{ \text{ left factors (1b) } \} \\
 & I \ \sqsupseteq \ f \circ F.(I/(\!|f|\!)^{\cup}) \circ \mu F \circ (\!|f|\!)^{\cup} \\
\equiv & \quad \{ \text{ (3), reverse } \} \\
 & I \ \sqsupseteq \ f \circ F.(I/(\!|f|\!)^{\cup}) \circ ((\!|f|\!) \circ \mu F)^{\cup} \\
\equiv & \quad \{ \text{ (17) } \} \\
 & I \ \sqsupseteq \ f \circ F.(I/(\!|f|\!)^{\cup}) \circ (\!|f|\!)^{\cup} \\
\equiv & \quad \{ \text{ computation rule (15a), reverse } \} \\
 & I \ \sqsupseteq \ f \circ F.(I/(\!|f|\!)^{\cup}) \circ F.(\!|f|\!)^{\cup} \circ f^{\cup} \\
\equiv & \quad \{ \text{ relators (10c) } \} \\
 & I \ \sqsupseteq \ f \circ F.(I/(\!|f|\!)^{\cup} \circ (\!|f|\!)^{\cup}) \circ f^{\cup} \\
\Leftarrow & \quad \{ \text{ left factors (2b), monotonicity } \} \\
 & I \ \sqsupseteq \ f \circ F.I \circ f^{\cup} \\
\Leftarrow & \quad \{ \text{ (2a), monotonicity } \} \\
 & I \ \sqsupseteq \ f \circ f^{\cup}
\end{array}$$

The proof of (b), i.e.

$$I \ \sqsupseteq \ (\!|f|\!)^{\cup} \circ (\!|f|\!) \quad \Leftarrow \quad I \ \sqsupseteq \ f^{\cup} \circ f$$

is left as an entertaining exercise for the reader. One proof proceeds in the same way except that right factors are used instead of left factors. There is, however, a much shorter proof making use of the fusion properties. Part (c) is, of course, just the conjunction of (a) and (b).
□

5 Parameterised Types

5.1 New relators from old

The theorems in the earlier sections are all well and good but a major concern is to build new parameterised type constructors from existing ones. Within the world of functions the technique is now well-understood: consider a binary functor and abstract over one of its arguments. That is, if $\otimes$ is a binary functor and A is a monotype we consider the unary functor $A\otimes$ where for type B,

$$(A\otimes).B \quad = \quad A \otimes B$$

and, for function f,

$$(A\otimes).f \quad = \quad I_A \otimes f$$

This defines, as above, a monotype $\mu(A\otimes)$ that is parameterised by A. In other words, a function ϖ has been identified from monotypes to monotypes, where $\varpi A \ = \ \mu(A\otimes)$. It remains to show that ϖ can be extended to map functions to functions in such a way that it is a functor. To see how this is done we refer the reader to [38].

Our concern here is, however, not limited to functions. We wish to prove that ϖ is a relator: i.e. it can be extended to map specs to specs in such a way that it has all the properties required of a relator. The achievement of this goal is delightfully simple. Within the current algebraic setting the technique parallels that outlined above, but involves just the one definition. Specifically, the following:

Definition 28

Suppose $\otimes$ is a binary relator. It is easy to verify that $I\otimes$ is a relator (where $(I\otimes).R \ = \ I \otimes R$). Its catamorphisms therefore exist and we may define:

$$\varpi R \quad \hat{=} \quad (\!| I\otimes ; \ R \otimes I |\!)$$

□

Note that ϖR is the least solution of the equation $X :: \ X \ = \ R \otimes X$. (See the remarks following definition 15).

In the following calculations we adopt the convention that composition has lower precedence than "$\otimes$". We also drop the argument "$I\otimes$" within the catamorphism brackets since our discussion will be confined to just this one relator.

For easy reference it is useful to instantiate the unique extension property, computation rule and fusion properties of section 4.4 with $F := I\otimes$ and the definition of ϖR. After some simplification, using in particular the assumed compositionality of $\otimes$, these become:

(Unique extension property)

(29) $\quad X \ = \ \varpi R \ \equiv \ X \ = \ R \otimes X \circ \mu(I\otimes)$

(Computation rules)

(30) $\quad ([R]) \;=\; R \circ I \otimes ([R]) \;=\; ([R]) \circ \mu(I\otimes)$

(31) $\quad \varpi R \;=\; R \otimes \varpi R \;=\; \varpi R \circ \mu(I\otimes)$

(Fusion laws)

(32) $\quad U \circ \varpi V \trianglelefteq \varpi R \;\Leftarrow\; U \circ V \otimes I \trianglelefteq R \otimes U$

where "$\trianglelefteq$" is any of "$=$", "$\sqsupseteq$", "$\sqsubseteq$".
Recall also the defining property of $\mu(I\otimes)$:

(33) $\quad \mu(I\otimes) \;=\; I \otimes \mu(I\otimes)$

(34) $\quad X \sqsupseteq \mu(I\otimes) \;\Leftarrow\; X \sqsupseteq I \otimes X$

As would be expected, the fixed-point operator is monotonic in the following sense:

Lemma 35

$$\mu(R\otimes) \sqsupseteq \mu(S\otimes) \;\Leftarrow\; R \sqsupseteq S$$

(Note: Up till now we have used the operator "μ" only in the context of a relator. In general $R\otimes$ is not a relator (although $I\otimes$ is), but it is monotonic and so the existence of a least fixed point is guaranteed by the Knaster-Tarski theorem.)
Proof

$$\begin{array}{ll} & \mu(R\otimes) \;\sqsupseteq\; \mu(S\otimes) \\ \Leftarrow & \quad \{ \text{ definition of } \mu(S\otimes) \text{ } \} \\ & \mu(R\otimes) \;\sqsupseteq\; S \otimes \mu(R\otimes) \\ \equiv & \quad \{ \text{ definition of } \mu(R\otimes) \text{ } \} \\ & R \otimes \mu(R\otimes) \;\sqsupseteq\; S \otimes \mu(R\otimes) \\ \Leftarrow & \quad \{ \text{ monotonicity of relators } \} \\ & R \sqsupseteq S \end{array}$$

□

By theorems 11(a) and 14, $\mu(I\otimes)$ is a monotype. Applying lemma 35 we thus get the slightly more general:

Corollary 36

$$I \sqsupseteq \mu(A\otimes) \;\Leftarrow\; I \sqsupseteq A$$

□

which is the crucial ingredient in verifying the following:

Theorem 37 For all monotypes A

$$\varpi A \;=\; \mu(A\otimes)$$

Proof Assume A is a monotype. Then,

$$
\begin{array}{ll}
 & \mu(A\otimes) \;=\; \varpi A \\
\equiv & \qquad \{\ (29)\ \} \\
 & \mu(A\otimes) \;=\; A \otimes \mu(A\otimes) \circ \mu(I\otimes) \\
\equiv & \qquad \{\ (33)\ \} \\
 & \mu(A\otimes) \;=\; \mu(A\otimes) \circ \mu(I\otimes) \\
\equiv & \qquad \{\ \text{properties of monotypes, specifically (4)}\ \} \\
 & \mu(A\otimes) \;=\; \mu(A\otimes) \sqcap \mu(I\otimes) \\
\equiv & \qquad \{\ \text{corollary 36}\ \} \\
 & \textbf{true}
\end{array}
$$

□

It is a straightforward matter to verify that ϖ is a relator. Here are the proofs of the four properties.

Lemma 38

$$I \;\sqsupseteq\; \varpi I$$

Proof Immediate from the conjunction of theorem 37 and corollary 36.
□

We interpose a small lemma which we have found is often useful in its own right.

Lemma 39

$$\llparenthesis R \rrparenthesis \circ \varpi S \;=\; \llparenthesis R \circ S \otimes I \rrparenthesis$$

Proof

$$
\begin{array}{ll}
 & \llparenthesis R \rrparenthesis \circ \varpi S \;=\; \llparenthesis R \circ S \otimes I \rrparenthesis \\
\Leftarrow & \qquad \{\ \text{fusion — (25a)}\ \} \\
 & \llparenthesis R \rrparenthesis \circ S \otimes I \;=\; R \circ S \otimes I \circ I \otimes \llparenthesis R \rrparenthesis \\
\equiv & \qquad \{\ \text{compositionality of relators}\ \} \\
 & \llparenthesis R \rrparenthesis \circ S \otimes I \;=\; R \circ I \otimes \llparenthesis R \rrparenthesis \circ S \otimes I \\
\equiv & \qquad \{\ \text{computation rule, (30)}\ \} \\
 & \textbf{true}
\end{array}
$$

□

Lemma 40

$$\varpi R \circ \varpi S = \varpi(R \circ S)$$

Proof

$$\begin{array}{cl}
 & \varpi R \circ \varpi S \\
= & \quad \{ \text{ defn. of } \varpi \ \} \\
 & (\!|R \otimes I|\!) \circ \varpi S \\
= & \quad \{ \text{ lemma 39 } \} \\
 & (\!|R \otimes I \circ S \otimes I|\!) \\
= & \quad \{ \text{ compositionality of relators } \} \\
 & (\!|(R \circ S) \otimes I|\!) \\
= & \quad \{ \text{ defn. of } \varpi \ \} \\
 & \varpi(R \circ S)
\end{array}$$

□

Lemma 41 (Monotonicity)

$$\varpi R \sqsupseteq \varpi S \quad \Leftarrow \quad R \sqsupseteq S$$

Proof
Immediate from the definition of ϖ and the monotonicity of catamorphisms and relators.
□

Lemma 42 (Revertability)

$$(\varpi R)^\cup \quad = \quad \varpi(R^\cup)$$

Proof

$$\begin{array}{cl}
 & (\varpi R)^\cup \quad = \quad \varpi(R^\cup) \\
\equiv & \quad \{ \text{ uep (29) } \} \\
 & (\varpi R)^\cup \quad = \quad R^\cup \otimes (\varpi R)^\cup \circ \mu(I\otimes) \\
\equiv & \quad \{ \text{ properties of reverse, } \otimes \text{ is a relator } \} \\
 & \varpi R \quad = \quad \mu(I\otimes) \circ R \otimes \varpi R \\
\equiv & \quad \{ \text{ theorem 36; computation rule (31) } \} \\
 & \varpi R \quad = \quad \varpi I \circ \varpi R \\
\equiv & \quad \{ \text{ lemma 40, } I \text{ is the unit of composition } \} \\
 & \textbf{true}
\end{array}$$

□

Theorem 43 ϖ is a relator.

Proof Lemmas 38, 40, 41 and 42.
□

6 Conclusions

Although the relational calculus has now been around for quite a number of years it has been disparaged as unworkable (in comparison to existing programming calculi). In this paper we have provided elements of an argument to the contrary: that a useful calculus can be built around a relational theory of datatypes. To this end we have been particular conscious of the brevity and clarity of our definitions and calculations. No doubt improvements can be made, but at the time of writing we feel that we have an elegant and convincing theory that is indeed worth exploring further.

Of course, what we have presented here is only a beginning. The immediate cry will undoubtedly be "let us see how you use the theory to develop some programs". Some examples of program derivation that are problematic in the Bird-Meertens formalism but are not so in the current relational framework are discussed in [38]. However, we would be the first to admit that, as yet, we have little experience with the practical application of the theory to program development.

In terms of theory development there also remains much to be done. The present theory encompasses only the very simplest sorts of datatypes commonly occurring in computer programs. We have not yet developed the tools to handle infinite data structures (such as streams or game trees), types with laws (e.g. finite bags) or subtypes (e.g. height-balanced binary trees). Some of this work is underway but it would be premature to hazard a judgement on its chances of success. We hope, however, that some others may have been sufficiently stimulated to join us in our endeavour.

Acknowledgement

In its later stages preparation of this paper was expedited by the use of the proof editor developed by Paul Chisholm [20].

References

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

[2] R.C. Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS8810, Department of Mathematics and Computing Science, University of Groningen, 1988.

[3] R.C. Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, June 1989.

[4] R.C. Backhouse, P. de Bruin, G. Malcolm, E. Voermans, and J. van der Woude. A relational theory of datatypes. Eindhoven University of Technology and University of Groningen, September 1990.

[5] R.C. Backhouse, P. Chisholm, and G. Malcolm. Do-it-yourself type theory, part 1. *EATCS Bulletin*, 34, February 1988.

[6] R.C. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[7] R.C. Backhouse, Bruin P. de, P. Hoogendijk, G. Malcolm, Voermans T.S., and J. van der Woude. Polynomial relators. To appear: 2nd Conference on Algebraic Methodology and Software Technology, May 22-25, 1991.

[8] R.C. Backhouse and R.K. Lutz. Factor graphs, failure functions and bi–trees. In A. Salomaa and M. Steinby, editors, *Fourth Colloquium on Automata, Languages and Programming*, pages 61–75. Springer-Verlag, LNCS 52, July 1977.

[9] J.W. de Bakker and W.P. de Roever. A calculus for recursive program schemes. In M. Nivat, editor, *Proc. IRIA Symp. on Automata, Formal Languages and Programming*. North-Holland, Amsterdam, 1972.

[10] R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and nondeterministic programs. *Theoretical Computer Science*, 43:123–147, 1986.

[11] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM. Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[12] R.S. Bird. Transformational programming and the paragraph problem. *Science of Computing Programming*, 6:159–189, 1986.

[13] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987. NATO ASI Series, vol. F36.

[14] R.S. Bird. A calculus of functions for program derivation. Technical report, Programming Research Group, Oxford University, 11, Keble Road, Oxford, OX1 3QD, U.K., 1988.

[15] R.S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. Technical report, Programming Research Group, Oxford University, 11, Keble Road, Oxford, OX1 3QD, U.K., 1988.

[16] R.S. Bird and L. Meertens. Two exercises found in a book on algorithmics. In L.G.L.T. Meertens, editor, *Program Specification and Transformations*, pages 451–457. Elsevier Science Publishers B.V., North Holland, 1987.

[17] Garrett Birkhoff. *Lattice Theory*, volume 25 of *American Mathematical Society Colloquium Publications*. American Mathematical Society, Providence, Rhode Island, 3rd edition, 1948.

[18] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[19] P.J. de Bruin. Naturalness of polymorphism. Technical Report CS8916, Department of Mathematics and Computing Science, University of Groningen, 1989.

[20] P. Chisholm. Calculation by computer: Overview. Technical Report CS 9007, Department of Computing Science, University of Groningen, 1990.

[21] R.L. Constable, et al. *Implementing Mathematics in the Nuprl Proof Development System.* Prentice-Hall, 1986.

[22] J.H. Conway. *Regular algebra and finite machines.* Chapman and Hall, London, 1971.

[23] J. Desharnais. *Abstract Relational Semantics.* PhD thesis, School of Computer Science, McGill University, July 1989.

[24] E.W. Dijkstra and W.H.J. Feijen. *Een Methode van Programmeren.* Academic Service, Den Haag, 1984. Also available as *A Method of Programming*, Addison-Wesley, Reading, Mass., 1988.

[25] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, Berlin, 1990.

[26] R.P. Dilworth. Non-commutative residuated lattices. *Transactions of the American Mathematical Society*, 46:426–444, 1939.

[27] P. Dybjer. An inversion principle for Martin-Löf's type theory. Dept. Comp. Sci., Univ. Göteborg, 1989.

[28] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology, Volume 4: Data Structuring*, pages 80–149. Prentice-Hall, 1978.

[29] T. Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A. Poigne, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–57. Springer-Verlag Lecture Notes in Computer Science 283, 1988.

[30] C.A.R. Hoare. Notes on data structuring. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

[31] C.A.R. Hoare. A couple of novelties in the propositional calculus. *Zeitschr. fuer Math. Logik und Grundlagen der Math.*, 31(2):173–178, 1985.

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

[33] G. Jones. Constructing the fast Fourier transform. Lecture notes, International Summer School on Constructive Algorithmics, vol. 3, 1989.

[34] J. Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65:154–170, 1958.

[35] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[36] G. Malcolm. Data structures and program transformation. To appear, *Science of Computer Programming*, 1990.

[37] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Conference on the Mathematics of Program Construction*, pages 335–347. Springer-Verlag LNCS 375, 1989.

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

[39] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1986.

[40] P. Martin Löf. Constructive mathematics and computer programming. In L.J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *Logic, Methodology and Philosophy of Science, IV*, pages 153–175. North-Holland, 1982.

[41] L. Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[42] L. Meertens. Constructing a calculus of programs. In J.L.A. van de Snepscheut, editor, *Conference on the Mathematics of Program Construction*, pages 66–90. Springer-Verlag LNCS 375, 1989.

[43] J. Meseguer and J.A. Goguen. Initiality, induction and computability. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–542. Cambridge University Press, 1985.

[44] O. de Moor. Indeterminacy in optimization problems. Lecture Notes, International Summer School on Constructive Algorithmics, vol. 2, 1989.

[45] O. de Moor. Inverses in program synthesis. Lecture Notes, International Summer School on Constructive Algorithmics, vol. 2, 1989.

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

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

[48] G. Schmidt and T. Ströhlein. Relation algebras: Concept of points and representability. *Discrete Mathematics*, 54:83–92, 1985.

[49] G. Schmidt and T. Ströhlein. *Relationen und Grafen.* Springer-Verlag, 1988.

[50] M. Sheeran. Describing hardware algorithms in Ruby. In David et al., editor, *Proc. IFIP TC10/WG10.1 Workshop on Concepts and Characteristics of Declarative Systems.* North-Holland, 1989.

[51] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.

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

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.