The University of Warwick

THEORY OF COMPUTATION

REPORT NO.19

DATA TYPES

BY

DANIEL J. LEHMANN AND MICHAEL B. SMYTH

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

DATA TYPES

by

Daniel J. Lehmann[*] and Michael B. Smyth

Department of Computer Science
University of Warwick
Coventry, West Midlands,
CV4 7AL
Great Britain

[*] Address from September 1977: Department of Mathematics, University of Southern California, University Park, Los Angeles, California, 90007

Abstract

A Mathematical interpretation is given to the notion of a data type. The main novelty is in the generality of the mathematical treatment which allows procedural data types and circularly defined data types. What is meant by data type is pretty close to what any computer scientist would understand by this term or by data structure, type, mode, cluster, class. The mathematical treatment is the conjunction of the ideas of D. Scott on the solution of domain equations (Scott (71), (72) and (76)) and the initiality property noticed by the ADJ group (ADJ (75), ADJ (77)). The present work adds operations to the data types proposed by Scott and generalizes the data types of ADJ to procedural types and arbitrary circular type definitions.

The advantages of a mathematical interpretation of data types are those of mathematical semantics in general : throwing light on some ill-understood constructs in high-level programming languages, easing the task of writing correct programs and making possible proofs of correctness for programs or implementations.

1. Introduction

All programming languages have basic data types, some have many, some have few, some have only one basic data type. The most commonly used are : booleans, integers, reals, procedures, labels, atoms, lists. They generally come equipped with some operations : logical operations for booleans, arithmetical operations for integers and reals, composition, evealuation and abstraction for procedures, a defining facility for labels, and list-manipulation primitives for lists.

In most data types the user has the facility to denote new objects by the use of expressions combining old objects by operations. All these facilities are easy to understand. The first non-trivial fact about data types is that, in certain data types and in certain programming languages, objects can be defined implicitly; they are defined by expressions that contain their own denotation. This is the facility which is generally referred to as "recursive definition" but which we prefer to call "circular definition". This facility is generally offered for procedural data types only.

In the most advanced languages (called extensible) the user may define new data types from old ones by the use of constructors. In ALGOL 68, for example, these constructors are <u>struct</u>, <u>proc</u>, <u>ref</u>, <u>row</u>, <u>union</u>. New operations on data types may also be defined by defining new procedures.

In some languages, like ALGOL 68, new data types may also be defined circularly. The following two definitions are examples of such circular definitions in ALGOL 68:

mode tree = struct (int label, ref tree left, ref tree right)

mode fun = proc (fun) fun

All languages known to the authors that allow such circular definitions of data types put stringent restrictions on the generality of such a facility. Lehmann (77) showed that many circular definitions not allowed in ALGOL 68 are meaningful and very useful.

The question of the mathematical meaning of circular definitions inside procedural data types was first answered, independently, by H. Bekic, D. Park and D. Scott, who noticed that the functions defined were least fixpoints of monotone functionals. The fact that mode-constructors are functors was mentioned in Scott (72), in a remark attributed to Lawvere, and later made explicit by Reynolds and Wand (74). Circular definitions of sets and languages had been known to algebraists for a certain time when, in 1969, Scott gave a precise meaning to circular definitions involving the function-space constructor (his arrow, the ALGOL 68 proc). A general method to solve domain equations, implicit in Scott (72), was made explicit by Reynolds. The categorical nature of this unified construction, only hinted at by Scott, was emphasized by Wand (74). The main idea behind the present work is that the problems involved in defining data types can best be handled by an exact generalization of the well-understood methods used in studying definitions of objects within a data type. This involves generalizing from posets to categories, from monotone functions to functors, and from least fixpoints of continuous functions to initial fixpoints of continuous functors. The ADJ group concentrated on the problem of defining functions on data types and insisted that data types do not consist only of a set of elements, structured in some way (generally a partial order) but consist also of certain functions. They understood the importance of initiality and noticed that certain data types were initial algebras (or initial many-sorted algebras) but were unable to include procedural data types in their treatment and did not see the link with initial fixpoint of continuous functors. The relation between our work and the authors just mentioned can be summarized as follows. We provide a categorical version of Scott's domain constructions that is simpler than Wand's. At the same time, we take full account of the ideas of ADJ, while avoiding the limitations to equationally defined, non-procedural data types which their approach entails. As to the mathematical results in the paper, most of these are fairly obvious - once one has grasped the idea of systematically generalizing from posets to categories. The main purpose of the work is, however, not to present detailed results, but to show that a clear and rigorous basis for the theory and practice of data types can best be provided by the concepts of ω-categories, ω-continuous functors, and initial fixpoints.

2. Mathematics

This section will introduce the basic notions and notations to be used in the sequel.

Definition 1: A (similarity) type T is a ranked alphabet.

The rank of a symbol is called its arity.

A type is a set of symbols (intended to represent functions); to each symbol is attached a natural number (intended to be the number of arguments taken by the function represented), the arity of the symbol. If T is a type $T_n \subseteq T$ is the set of all symbols of rank n.

Definition 2: A (universal) algebra of type T is a set S (called the carrier of the algebra) and for each $n \in N$ a function $\phi_n : T_n \to S^{S^n}$.

ϕ_n associates with each symbol of arity n a function : $S^n \to S$ of the corresponding number of arguments.

The following notions of category theory will be assumed to be known: category, object, arrow, domain, codomain, identity, composition, small categories, limits, colimits, products, coproducts, equalizers, coequalizers, monics, epis, isomorphisms, initial and terminal objects, zero object, functors, coseparators, well powered categories. The reader is referred to MacLane (71) and Herrlich-Strecker (73).

Definition 3 : ω is the category whose objects are the natural numbers : $\{ 0, 1, 2, \ldots, i, \ldots \}$ and the arrows all couples (i,j) of natural numbers such that $i \leq j$, with the obvious identities and composition.

Definition 4 : C is an ω-category iff C has an initial object and all colimits of ω-diagrams.

Definition 5 : A functor $F : A \to B$ is an ω-functor iff F preserves all existing colimits of ω-diagrams.

Definition 6 : If $T : C \to C$ is an endo-functor a T-algebra is an arrow (of C) of the form $\phi : T_c \to c$

A similarity type T', as in definition 1, can be considered as a functor T in the category Set. An example will show this better than a formal definition.

Let $T' = \{ 0, S \}$ with rank $(0) = 0$, rank $(S) = 1$

T would be the functor defined by $TA = 1 + A$ and $Tf = I_1 + f$.

Then a T-algebra would be a function $\phi : 1 + S \to S$ and would correspond uniquely with a set (S) equipped with one constant and one unary operation. The reader will easily see how to generalize the above example to arbitrary similarity types (even infinite ones).

Definition 7 : If $T : C \to C$ is an endo-functor the category of T-algebras is the category whose objects are the T-algebras and whose arrows, from $\phi : Tc \to c$ to $\psi : Td \to d$ are those arrows $\alpha : c \to d$ of C such that $\alpha\phi = \psi T\alpha$.

$$\begin{array}{ccc} c & \xleftarrow{\phi} & Tc \\ \alpha \downarrow & & \downarrow T\alpha \\ d & \xleftarrow[\psi]{} & Td \end{array}$$

If T corresponds to a similarity type T' the arrows of the category of T-algebras are the homorphisms of the universal algebras of type T'.

A word of caution is necessary here to warn the reader that our definition of a T-algebra, though seemingly only an extension of the one found in Mac-Lane (71) where T is always supposed to be a monad, has in fact a different purpose. The functor T that we make to correspond to a similarity type T' is not the one Mac-Lane would consider (the one building the carrier of the free algebra). Our notion of an algebra is identical to what Arbib and Manes (74) called a T-dynamics. We are interested only in the case where T is an ω-functor and C an ω-category.

Theorem 1 : Let C be an ω-category and $T : C \to C$ be an ω-functor, then the category of T-algebras is an ω-category.

Proof : The proof of this theorem is more-or-less routine arrow-chasing. As the existence of ω-colimits will not be used in the sequel its proof will be left to the reader. The existence of an initial T-algebra will be proved in detail.

Let $\perp$ be the initial object of C (its existence is ensured because C is an ω-category) and let $\perp_c$ be the unique arrow : $\perp \to c$.

The following ω-diagram has a co-limit (C is an ω-category)

$$\perp \xrightarrow[\perp_{T\perp}]{} T\perp \xrightarrow[T\perp_{T\perp}]{} T^2\perp \xrightarrow[T^2\perp_{T\perp}]{} T^3\perp \longrightarrow \cdots$$

Let $\mu_i : T^i\perp \to a$ be a colimiting cone.

$T\mu_i : T^{i+1}\perp \to Ta$ is a colimiting cone because T is an ω-functor.

Then there is a unique $\alpha : Ta \to a$ such that $\alpha \circ T\mu_i = \mu_{i+1}$. We claim that α is the initial T-algebra.

Suppose $\beta : Tc \to c$ is a T-algebra.

Define $\nu_o = \perp_c$ and $\nu_{i+1} = \beta \circ T\nu_i$.

$\nu_i \circ \perp_{T\perp} = \perp_c = \nu_o$ and by induction on i :

$\nu_{i+1} \circ T \perp_{T\perp} = \beta \circ T (\nu_i \circ T^{i+1}\perp_{T\perp}) = \beta \circ T\nu_{i-1} = \nu_i$,

and the cone ν_i commutes.

Suppose $\gamma : a \to c$ is an arrow of T-algebras

$$\begin{array}{ccc} a & \xrightarrow{\gamma} & c \\ \alpha \uparrow & & \uparrow \beta \\ Ta & \xrightarrow[T\gamma]{} & Tc \end{array}$$

$\gamma \circ \mu_o = \gamma \circ \perp_a = \perp_c = \gamma_o$ and by induction

$\gamma \circ \mu_{i+1} = \gamma \circ \alpha \circ T\mu_i = \beta \circ T\gamma \circ T\mu_i = \beta \circ T(\gamma \circ \mu_i) = \beta \circ T\nu_i = \nu_{i+1}$

and γ has to be the unique arrow such that $\gamma \circ \mu_i = \nu_i$.

On the other hand, by the universality of μ_i there is such a γ.

$\gamma \circ \alpha \circ T\mu_i = \gamma \circ \mu_{i+1} = \nu_{i+1} = \beta \circ T\nu_i = \beta \circ T\gamma \circ T\mu_i$

By universality of $T\mu_i$: $\gamma \circ \alpha = \beta \circ T\gamma$ and γ is an arrow of T-algebras. □

Remark : From Theorem 1 we shall only use the existence of initial T-algebras and the careful reader may have noticed that we did not prove the most general possible results. Initial algebras may be proved to exist even in categories in which not all ω-diagrams have a colimit; it is enough to suppose that all ω-diagrams in a specified subcategory have colimits (in the large category), that the initial element is in the subcategory and is initial in the subcategory and that the subcategory is closed under T. These results may be of use when studying certain categories which are not ω-categories, but the simple version restricted to ω-categories is adequate for the purposes of this paper.

<u>Theorem 2</u> : Let T be an endo-functor on C. If $\alpha : Ta \to a$ is an initial T-algebra then α is an isomorphism.

<u>Proof</u> :

$$\begin{array}{ccc} a & \overset{\alpha}{\leftarrow} & Ta \\ \beta \downarrow & & \downarrow T\beta \\ Ta & \underset{T\alpha}{\leftarrow} & T^2a \end{array}$$

Initiality of α implies the existence of $\beta : a \to Ta$ such that $\beta \circ \alpha = T\alpha \circ T\beta$

But $\alpha \circ \beta \circ \alpha = \alpha \circ T(\alpha \circ \beta)$

$$\begin{array}{ccc} a & \overset{\alpha}{\leftarrow} & Ta \\ \alpha \circ \beta \downarrow & & \downarrow T(\alpha \circ \beta) \\ a & \underset{\alpha}{\leftarrow} & Ta \end{array}$$

which, by initiality of α implies $\alpha \circ \beta = I_a$

Then $\beta \circ \alpha = T(\alpha \circ \beta) = T(I_a) = I_{Ta}$

α and β are inverse isomorphisms. □

We want now to proceed in giving examples of the application of the above theorems. Our claim is that data types can always be considered to be objects in an appropriate ω-category, such that each data type constructor is an ω-endo-functor of this category.

Example 1 : Set and universal algebras

Set is cocomplete (and also complete) and so is an ω-category.
$\times$: Set$\times$Set$\to$Set is an ω-functor because it is a product and finite limits preserve directed co-limits in Set (see Mac Lane 1971 Theorem 1 p. 211).
$+$: Set$\times$Set$\to$Set is an ω-functor because it is a coproduct, and so has a right adjoint and preserves all colimits. Obviously constant functors are ω-functors, composition of ω-functors is an ω-functor and a bi-functor is an ω-functor iff it is an ω-functor separately in each argument.

Theorem 3 : Let T be a similarity type, then the associated functor T : Set$\to$Set is an ω-functor.

The proof is obvious. Theorem 1 and 2 then imply the existence of initial algebras of any type and the fact that the initial algebra, as a function, is an isomorphism. One knows that, in Set, not only initial algebras but also arbitrary free algebras exist and also arbitrary free algebra in equational classes of algebras, but this is of no interest to us. The existence of initial algebras was known long before the term initial had been coined and we claim no credit for the above theorem. The framework of universal algebras is too restricted for data types and, as noticed by ADJ, many-sorted algebras seem more suited.

Example 2 : Set^n and n-sorted algebras.

$$Set^n = \underbrace{Set \times Set \times \ldots \times Set}_{n \text{ times}}$$

Set^n is obviously cocomplete (and also complete) and so is an ω-category. If $T : Set^n \to Set^n$ the T-algebras are a generalization of what is called in the literature many-sorted algebras, heterogenous algebras or algebras with a scheme of operators. Theorem 2 implies the Proposition 2.1 of ADJ (77). Many sorted algebras are closer than algebras to what one understands data-types should be, nevertheless the problems of circular definition of objects inside a data-type cannot be tackled in Set^n for lack of an order structure on the objects (which are n-tuples of unordered sets). Burge (75) is probably the best, though somewhat informal, account of what can be done with sets.

Example 3 : ω-CPO^* and continuous algebras.

To remedy the absence of order structure on the objects ADJ (77) have proposed to use many-sorted algebras whose carriers are ω-complete partial orders with least element and whose operations are ω-continuous functions. Our objection to this is that the problem of circularly defined data-types whose definition involves the arrow (or the ALGOL 68 proc constuctor) is not solved, simply because the arrow is not a bi-functor in the above category : it is contravariant in the first argument. We shall nevertheless show that our results allow a very simple proof of ADJ (77)'s main technical result : the existence of initial continuous algebras. Let ω-CPO* be the category the objects of which are the ω-complete posets (every ω-diagram $a_0 \subseteq a_1 \subseteq \ldots a_i \subseteq \ldots$ has a l.u.b.) with least elements and the arrows of which are the strict (bottom preserving) ω-continuous functions. Markowsky (74) showed that the full subcategory of ω-CPO* which consists of all chain-complete posets (which he calls CPC*) is complete and cocomplete. We shall briefly pause here to prove this result for ω-CPO*; the method used here is a definite improvement on Markowsky's. Nevertheless these results will not be used in the sequel both because we do not think that ω-CPO* is a good candidate for the category of data types and because, by using the remark after Theorem 1, the existence of an initial T-algebra in ω-CPO* can be proved for all functors T which preserve a special class of monics for which it can be shown that all ω-diagrams (of special arrows) have a co-limit (in ω-CPO*).

<u>Co-completeness Theorem</u> : ω - CPO* is complete and co-complete.

<u>Proof</u> : To prove completeness it is enough to prove the existence of products and equalizers of pairs (MacLane (71) p.109). Products in ω-CPO* are just like in Sets; it is a trivial task to check that the product of ω-complete partial orders is an ω-complete partial order, that the projections are strict continuous functions, that the unique mediating arrow from a cone of continuous functions is continuous and

that the unique mediating arrow from a cone of strict functions is strict. Equalizers of pairs in ω-CPO* are just like in Set.

The equalizer of $A \underset{g}{\overset{f}{\rightrightarrows}} B$ is $h : A' \to A$ where $A' = \{ a \mid a \in A, f(a)=g(a) \}$ with the ordering induced by the one on A and h is the injection. A' is an ω-complete CPO because f and g are strict which implies $\perp \in A'$ and f and g are ω-continuous. h is obviously strict and continuous. Now to prove co-completeness, by Herrlich and Strecker (73) (23. 14 p. 163) it is enough to prove that ω-CPO* is well-powered, and has a co-separator. ω-CPO* is easily seen to be well-powered.

Let $2 = \{ \perp, \top \}$ be the two-points ω-CPO order by $\perp \sqsubseteq \top$.

Lemma a : If $f : A \to B$ is a monic in ω-CPO* then it is one-to-one.

Proof : Suppose $f(a_1) = f(a_2)$ for f monic. Let $h_i : 2 \to A$ be defined for i = 1,2 by $h_i(\perp) = \perp$ and $h_i(\top) = a_i$.

For i = 1,2, h_i is a strict ω-continuous function and

$h_1 \circ f = h_2 \circ f \Rightarrow h_1 = h_2 \Rightarrow a_1 = a_2$

Lemma b : 2 is a co-generator in ω-CPO*.

Proof : Suppose f,g are two different arrows : $A \to B$.

Then $\exists\, a_o \in A$ such that $f(a_o) \neq g(a_o)$ and by symmetry we may suppose $f(a_o) \not\sqsubseteq g(a_o)$.

Let $h : B \to 2$ be defined by $h(b) = \begin{cases} \perp \text{ if } b \sqsubseteq g(a_o) \\ \top \text{ else} \end{cases}$

Clearly h is monotone and continuous.

But $hf(a_o) = \perp$ and $hg(a_o) = \top$ and $hf \neq hg$. □

Our Theorem 1 then implies the existence of an initial Σ-algebra for any ranked alphabet Σ, which is the main result of ADJ (77). Obviously the same holds for many-sorted continuous algebras, which are ω-functors in (ω-CPO^*)^n. Categories slightly different from ω-CPO^*, for example that of ω-complete cpo's and strict Δ-complete cpo's and strict Δ-continuous maps are, by similar proofs, seen to be ω-categories and the many initiality results of ADJ (77) can be obtained in a unified way, if one thinks these are interesting. The category whose objects are countably based algebraic posets with least elements and whose arrows are strict ω-continuous maps which preserve finite elements, inspired from Courcelle and Nivat (78) is also cocomplete (and complete) and hence an ω-category. This last result can be proved either by the method used above or by noticing that the category is equivalent to that of partial orders and strict monotone functions which is very easily studied. In all preceding examples the construction of Adamek (74) ensures the existence of arbitrary free T-algebras over any object. Our insistence that T be an ω-functor guarantees that the free T-algebras are obtained as colimits of ω-diagrams. ADJ noticed that expressions with variables represented objects in such free T-algebras and made totally clear the way such objects yield maps from the environment to the obvious domain. In the sequel we shall admit without further formalities that expressions built out of constants, variables and (continuous) functions yield (continuous) functions or more generally (continuous) functionals.

Example 4 : CPO^a and circularly defined data types.

Let CPO^a (a stands for adjunction) be the category the objects of which are the ω-complete partial orders (the same objects as those of ω-CPO^*) and the arrows of which $f : A \to B$ are the pairs of ω-continuous maps $f = (f^L, f^R)$ $f^L : A \to B$ and $f^R : B \to A$ such that $f^R \circ f^L = I_A$ and $f^L \circ f^R \sqsubseteq I_B$.

The following lemmas prove that CPO^a is indeed a category. The proofs are trivial and left to the readers.

Lemma 1 : $f : A \to A$ defined by $f = (I_A, I_A)$ is an arrow of CPO^a.

Lemma 2 : If $f : A \to B$ and $g : B \to C$ are arrows in CPO^a then

$$h = (g^L \circ f^L,\ f^R \circ g^R) \text{ is an arrow} : A \to C.$$

The arrows of CPO^a are the pairs of projections of Scott (72), the embeddings of Smyth (76). Wand (74) seems to have been the first to state that the full subcategory of CPO^a consisting of complete lattices is an ω-category. Plotkin (76) and the authors noticed that +, × and the function space constructor (→) were ω-functors in CPO^a, so simplifying Wand's (74) treatment which uses a non-standard notion of continuity. Lehmann (76) proves a more general statement from which the fact that CPO^a is an ω-category and ×, +, → are ω-functors are instant corollaries. Certain full subcategories of CPO^a : the category SFP-R of Plotkin, the category of algebraic consistently complete cpo's, that of effectively given algebraic consistently complete cpo's have been shown to be ω-categories themselves by Plotkin and Smyth. They also are closed under ×, + and → . SFP-R is also closed under the power domain constructor $\underline{P}$ of Plotkin (76); the others are closed only under Smyth's P_o (Smyth (76b)).

Smyth (76a) has defined a subcategory of CPO^a containing as objects only the effectively given continuous consistently complete cpo's which is closed under $\times$, $+$, $\rightarrow$ and P_o, and certain ω-colimits. Unlike all previous examples CPO^a does not possess products or coproducts and Adamek's (74) result about free algebras is inapplicable. Yet by Theorem 2 initial T-algebras exist for arbitrary ω-functors T. Now some obvious lemmas will be stated with only hints of proof or without proof.

Lemma 3 : If $f = (f^L, f^R)$ is an arrow in CPO^a, then any one of f^L or f^R uniquely determines the other.

Proof : Suppose (f^L, f_1^R) and (f^L, f_2^R) are arrows in CPO^a then $f_1^R = f_2^R f^L f_1^R \sqsubseteq f_2^R$ and symmetrically.

Lemma 4 : If $f = (f^L, f^R)$ is an arrow in $CPO^a : A \rightarrow B$ then $f^L(\bot_A) = \bot_B$ and $f^R(\bot_B) = \bot_A$.

Lemma 5 : 1, the one-point partial order is initial in CPO^a.

Lemma 6 : If $f^L : A \rightarrow B$ and $f^R : B \rightarrow A$ are monotone functions such that f^R is ω-continuous, $f^R o f^L = I_A$ and $f^L o f^R \sqsubseteq I_B$ then f^L is ω-continuous.

The following lemmas precise the facts about co-limits in CPO^a. The first lemma is a concrete proposition about CPO^a but all following lemmas may be proved in the abstract framework of order-enriched categories and applied directly to other categories, in particular sub-categories of CPO^a.

Lemma 7 : Let $\Gamma : A_0 \xrightarrow{f_0} A_1 \xrightarrow{f_1} A_2 \rightarrow \rightarrow A_i \xrightarrow{f_i} A_{i+1} \rightarrow \ldots$ be an ω-diagram in CPO^a. There exists a commuting cone $\mu : \Gamma \rightarrow A_\infty$ such that for any $i \in N$ $\mu_i^L \mu_i^R \sqsubseteq \mu_{i+1}^L \mu_{i+1}^R$ and $\bigsqcup_i \mu_i^L \mu_i^R = I_{A_\infty}$.

Proof : Let $A_\infty = \{ < a_0, a_1, \ldots a_i \ldots > \mid \forall i \in N \ \ a_i \in A_i$ and $a_i = f_i^R a_{i+1} \}$ ordered by componentwise ordering. A_∞ is an ω-complete partial order by Lemma 4 and because the f_i^R are ω-continuous. The ω-lub's in A_∞ are componentwise. Let $p_i \ A_\infty \rightarrow A_i$ be the i^{th} projection : $p_i (< a_0, \ldots a_i, \ldots >) = a_i$. p_i is ω-continuous because lub's in A_∞ are componentwise. Let $q_i : A_i \rightarrow A_\infty$ be defined by

$q_i a_i = < f_0^R \ f_{i-1}^R \ a_i, \quad , f_{i-1}^Q a_i, \ a_i, \ f_i^L a_i, \ f_i^L \ f_i^L \ a_i, \ldots >$.

Clearly q_i is well-defined, monotone and $\forall i \in N \ q_i = q_{i+1} \circ f_i^L$.

Furthermore $p_i \circ q_i = I_{A_i}$ and $q_i \circ p_i \sqsubseteq I_{A_\infty}$ so that by Lemmas 3 and 6, $\mu = (q_i, p_i)$ defines a commuting cone $\Gamma \rightarrow A_\infty$ in CPO^a.

$q_i \circ p_i = q_{i+1} \circ f_i^L \circ f_i^R \circ p_{i+1} \sqsubseteq q_{i+1} \circ p_{i+1}$.

$\bigsqcup_i (q_i \circ p_i) \sqsubseteq I_{A_\infty}$ because $q_i \circ p_i \sqsubseteq I_{A_\infty}$, $\forall i \in N$

$\forall j \in N$ $p_j \circ (\bigsqcup_i q_i \circ p_i) \sqsupseteq p_j \circ q_j \circ p_j = p_j$ which implies that

$p_j \circ (\bigsqcup_i q_i \circ p_i) < a_0, \ldots, a_i \ldots > = a_j$ and $(\bigsqcup_i q_i \circ p_i) = I_{A_\infty}$. □

In the next lemmas all ω-sequences of functions for which l.u.b's are used can be easily checked to be ascending.

Lemma 8 : Let Γ (as in Lemma 7) be an ω-diagram in CPO^a.

If the cone $\nu : \Gamma \to B$ is a colimiting cone (in CPO^a) then $\bigsqcup_i \nu_i^L \nu_i^R = I_B$

Proof : Let $\mu : \Gamma \to A_\infty$ be the cone defined in Lemma 7. There exists a unique $h : B \to A_\infty$ such that $h \circ \nu_i = \mu_i$.

Then $\mu_i^L = h^L \circ \nu_i^L$ and $\mu_i^R = \nu_i^R \circ h^R$.

$$I_B = h^R \circ h^L = h^R \circ I_{A_\infty} \circ h^L = h^R \circ (\bigsqcup_i \mu_i^L \mu_i^R) \circ h^L = \bigsqcup_i h^R \circ \mu_i^L \circ \mu_i^R \circ h^L$$

$$= \bigsqcup_i h^R \circ h^L \circ \nu_i^L \circ \nu_i^R \circ h^R \circ h^L = \bigsqcup_i \nu_i^L \circ \nu_i^R .$$

Lemma 9 : Let Γ be an ω-diagram in CPO^a and $\mu : \Gamma \to A_\infty$ be a commuting cone such that $\bigsqcup_i \mu_i^L \mu_i^R = I_{A_\infty}$ then μ_i^L is a colimiting cone in $\omega\text{-}CPO^*$ for Γ^L.

Proof : Clearly μ_i^L commutes. Suppose $\alpha : \Gamma \to B$ is a commuting cone in $\omega\text{-}CPO^*$ and h is an arrow : $A_\infty \to B$ in $\omega\text{-}CPO^*$ such that $h \circ \mu_i^L = \alpha_i$ then $h = h \circ I_{A_\infty} = h \circ (\bigsqcup_i \mu_i^L \mu^R) = \bigsqcup_i h \mu_i^L \mu_i^R = \bigsqcup_i \alpha_i \mu_i^R$

which proves unicity.

For existence let $h = \bigsqcup_i \alpha_i \mu_i^R$. Then $h \circ \mu_j^L = \bigsqcup_{i \geq j} \alpha_i \mu_i^R \mu_j^L =$

$\bigsqcup_{i \geq j} \alpha_i \mu_i^R \mu_i^L f_{i-1}^L \circ\circ\circ f_j^L = \bigsqcup_{i \geq j} \alpha_i f_{i-1}^L \circ\circ\circ f_j^L = \bigsqcup_{i \geq j} \alpha_j = \alpha_j$ □

Lemma 10: With the same hypotheses as in Lemma 9, μ_i^R is a limiting cone for the ω^{op}-diagram Γ^R.

Proof : This is the dual of Lemma 9 (reverse the arrows but not the ordering on arrows).

Lemma 11 : Let Γ be an ω-diagram in CPO^a and $\mu : \Gamma \to A_\infty$ be a commuting cone such that μ^L is a colimiting cone in ω-CPO^* for Γ^L then μ is a colimiting cone (in CPO^a).

Proof : Suppose $\nu : \Gamma \to B$ is a commuting cone in CPO^a. Then $\nu^L : \Gamma^L \to B$ is a commuting cone in ω-CPO^* and there exists a unique $h : A_\infty \to B$ such that $h \circ \mu^L = \nu^L_i$. By Lemma 3 this proves unicity. To prove existence it is enough to find a right-adjoint to h.

$$(\bigsqcup_i \mu^L_i \nu^R_i) \circ h \circ \mu^L_j = \bigsqcup_{i \geq j} \mu^L_i \nu^R_i \nu^L_j = \mu^L_j \Rightarrow (\bigsqcup_i \mu^L_i \nu^R_i) \circ h = I_{A_\infty}$$

$$h \circ (\bigsqcup_i \mu^L_i \nu^R_i) = \bigsqcup_i h \mu^L_i \nu^R_i = \bigsqcup_i \nu^L_i \nu^R_i \sqsubseteq I_B$$

$(h, \bigsqcup_i \mu^L_i \nu^R_i)$ is an arrow in CPO^a. □

Lemma 12 : Let Γ be an ω-diagram in CPO^a and $\mu : \Gamma \to A_\infty$ be a commuting cone such that μ^R is a limiting cone for Γ^R in ω-CPO^*, then μ is a colimiting cone (in CPO^a).

Proof : Dual of Lemma 11.

Definition : Let E_L be the covariant embedding of CPO^a in ω-CPO^* which sends each pair of functions to its left part and E_R be the contravariant embedding of CPO^a in ω-CPO^* which sends each pair of functions to its right part. Clearly E_L and E_R are the identities on objects.

Theorem 4 : 1) CPO^a is an ω-category

2) E_L preserves and reflects ω-co-limits

3) E_R transforms ω-co-limits into ω^{op}-limits and reflects ω^{op}-limits into ω-co-limits.

Proof : 1) by Lemmas 5, 7, 9 and 11.

2) by Lemmas 8 and 9 and Lemma 11.

3) by Lemmas 8 and 10 and Lemma 12.

The next Theorem will be used to prove that functors (or bi-functors) in CPO^a are ω-functors.

<u>Theorem 5</u> : If T is an endo-functor on CPO^a (a bi-functor $CPO^a \times CPO^a \to CPO^a$), T preserves ω-colimits if for every sequence $f_o^L f_o^R \sqsubseteq f_1^L f_1^R \sqsubseteq \ldots \sqsubseteq f_i^L f_i^R \sqsubseteq \ldots$ $\bigsqcup_i f_i^L f_i^R = I \Rightarrow \bigsqcup_i (T f_i)^L (T f_i)^R = I.$

<u>Proof</u> : By Lemmas 8, 9 and 11.

CPO^a, can be considered as a (non-full) subcategory of $\omega\text{-}CPO^*$; if $T' : \omega\text{-}CPO^* \to \omega\text{-}CPO^*$ preserves adjunctions its restriction to CPO^a is a functor $T : CPO^a \to CPO^a$. The following theorem shows that the initial T-algebra is also an initial T'-algebra. It is useful to draw more radical initiality properties for the data types circularly defined by definitions involving only + and × (at the exception of →), as those considered in ADJ (75).

<u>Theorem 6</u> : Let $T : CPO^a \to CPO^a$ be a functor and $T' : \omega\text{-}CPO^* \to \omega\text{-}CPO^*$ an ω-functor such that $E_L T = T' E_L$ then T is an ω-functor and if ϕ is the initial T-algebra then $E_L \phi$ is the initial T'-algebra.

<u>Proof</u> : T' is an ω-functor by hypothesis and E_L is an ω-functor by Theorem 4 $\Rightarrow T'E_L = E_L T$ is an ω-functor. But by Theorem 4 E_L reflects ω-colimits and then T is an ω-functor. Theorem 1 asserts the existence of $\phi : TA \to A$ the initial T-algebra. The proof of Theorem 1 shows that ϕ is the unique arrow in CPO^a such that $\phi \circ T\mu_i = \mu_{i+1}$ for $\mu : \Gamma \to A$ the colimiting cone for Γ

$$\Gamma : \bot \underset{\bot_{T\bot}}{\to} T\bot \to \ldots \to T^i\bot \underset{T^i\bot_{T\bot}}{\to} T^{i+1}\bot$$

By Theorem 4, E_L preserves ω-colimits and $E_L\mu : E_L\Gamma \to A$ is a colimiting cone. Because the initial object in CPO^a is also initial in ω-CPO^*, $E_L\Gamma$ is the following diagram in ω-CPO^* :

$$E_L\Gamma : \bot \underset{\bot_{T'L}}{\to} T'\bot \underset{T'\bot_{T'L}}{\to} T'^2\bot \to \dots \to T'^i \underset{T'^i\bot_{T'\bot}}{\to} T'^{i+1} \to \dots$$

The proof of Theorem 1 shows that the (up to isomorphism) initial T'-algebra is the unique arrow $\psi : T'A \to A$ of ω-CPO^* such that

$\forall i \in N \quad \psi \circ T'E_L\mu_i = E_L\mu_{i+1}$

But $E_L\mu_{i+1} = E_L\ \phi \circ E_L\ T\ \mu_i \ = E_L\ \phi \circ T'E_L\mu_i$

and $\psi = E_L\ \phi$.

□

<u>Remark</u> : Markowsky's (74) result on the cocompleteness of CPC^* is not used. Circularly defined data types the definition of which involves + (<u>union</u>), × (<u>struct</u>) and → (<u>proc</u>) can be seen to be initial algebras in CPO^a, as will be explained in the sequel (see Lehmann (77) for a preview oriented towards ALGOL 68). Scott's original (72) solution to domain equations consisted of considering the subcategory of CPO^a whose objects are continuous lattices and whose arrows are pairs of Δ-continuous projections. It can be easily checked that this subcategory is closed under ω-colimits. Scott (76) proposes another definition of data types and reduces the solution of domain equations to least upper bounds. Plotkin and Smyth have recently shown that these l.u.b's are ω-colimits in a suitable category of adjunctions which is equivalent to the subcategory of CPO^a considered originally by Scott (72).

Example 5 : *Dom* and a more general notion of a data-type.
Lehmann (76) defined a category *Dom* the objects of which are ω-categories and the arrows of which are adjunctions with identity unit. *Dom* is an ω-category and $\times$, $+$, $\rightarrow$ and P can be defined to be ω-functors. CPO^a is a full subcategory of *Dom* closed under ω-co-limits, $\times$, $+$, and $\rightarrow$. The correspondents of Theorems 4 and 5 hold and relate *Dom* and ω-Cat, the category of ω-categories and strict ω-functors. *Dom* provides a more general notion of a data type, useful when the powerset constructor, or non-deterministic procedural types are allowed.

We shall now proceed to give a number of examples showing how circular type definitions do indeed define initial algebras. The algebraic aspects will be stressed : a circular type definition does not only define a partially ordered set but also some functions on this set.

Example 6 : The natural numbers.
The natural numbers, or even the integers, are generally thought to be a basic data type. We shall now demonstrate how they can be circularly defined. Our treatment is equivalent to Lawvere's (64), as reported in MacLane-Birkhoff (67) (pp. 67-70).

Let our underlying category be Set, *1* be the one-point set (it is a terminal object in Set and also a generator) , $\perp$ the unique element of *1*, $+$ be the co-product (disjoint union) and let T be the functor defined by :

if e is a set $Te : \mathit{1} + e$

if $f : e_1 \to e_2$ is a function $Tf : Te_1 \to Te_2$

is the function defined by $(Tf)(a) = \begin{matrix} fa & \text{if } a \in e_1 \\ \bot & \text{if } a \in \mathit{1} \end{matrix}$

In category theoretic notation $Tf = I_{\mathit{1}} + f$

Clearly T is a functor : Set → Set. It is indeed the functor associated with the similarity type $T' = \{ 0, S \}$ rank (O) = 0, rank (s) = 1 described after Definition 6. T is an ω-functor, as all functors associated with similarity types, as explained in Example 1. By Theorem 1 there is an initial T-algebra, $\phi : 1 + A \to A$. The initiality property of ϕ characterizes, up to isomorphism, the function $0 + suc : \mathit{1} + \mathit{N} \to \mathit{N}$ which sends $\bot \in \mathit{1}$ to zero and $n \in \mathit{N}$ to $n + 1 = suc(n)$.

<u>Lemma 1</u> : $0 + suc : \mathit{1} + \mathit{N} \to \mathit{N}$ is the initial T-algebra.

<u>Proof</u> : Suppose $\phi : \mathit{1} + B \to B$

Suppose $\alpha : \mathit{N} \to B$.

If (1) $\alpha \circ (0 + suc) = \phi \circ (I_{\mathit{1}} + \alpha)$ then

(2) $\alpha(o) = (\alpha \circ (0 + suc))(\bot) = (\phi \circ (I_{\mathit{1}} + \alpha))(\bot) = \phi(\bot)$

(3) $\alpha(n + 1) = (\alpha \circ (0 + suc))(n) = (\phi \circ (I_{\mathit{1}} + \alpha)(n) = \phi(\alpha(n))$

Conversely if α verifies (2) and (3)

$\alpha \circ (0 + suc)(\bot) = \alpha(o) = \phi(\bot)$

$\alpha \circ (0 + suc)(n) = \alpha(n + 1) = \phi(\alpha(n))$ and

$\alpha \circ (0 + suc) = \phi \circ (I_{\mathit{1}} + \alpha)$.

By the induction rule for the natural numbers there exists exactly one function $\alpha : N \to B$ verifying (2) and (3) and this proves the initiality of $0 + suc : 1 + N \to N$.

The point we want to make here is that the type N can be circularly defined by : $N = 1 + N$ or in ALGOL 68 notation <u>mode</u> <u>natural</u> = <u>union</u> (<u>void</u>, <u>natural</u>). The initial fixpoint involves not only the set N but also the constant 0 and the function successor. The induction rule for natural numbers which is vital for proving properties of programs manipulating natural numbers is nothing else than an initiality property. In other words the fact that the unique $\alpha : N \to B$ implied by Theorem 1 is a total function is the main tool in proving that certain functions are total, in contradistinction with the more general partial functions which may be defined circularly inside $[N \to B]$ by means of arbitrary continuous functionals : $[N \to B] \to [N \to B]$.

For other circularly defined data types the initiality property may often be used directly in place of an induction principle. In other cases it is the main tool in proving the correctness of an induction principle. The question of the exact relation between initiality and induction will be treated in Section 5.

All useful functions on natural numbers may be defined from 0 + suc with the help of Lemma 1. The usual definition of addition for example may be readily transformed to fit this framework.

$$n + m = \begin{array}{ll} n & \text{if } m=0 \\ suc(n + m') & \text{if } m = suc(m') \end{array}$$

may be obtained the following way.

Let $T = \lambda f\ (I_1 + f)$ and $\psi : T([N \to N]) \to [N \to N]$ be defined by : $\psi = I_N + \lambda f\ (\mathrm{suc} \circ f)$.

Then there is a unique $\alpha : N \to [N \to N]$ making the following diagram commute.

$$\begin{array}{ccc} N & \xleftarrow{\mathrm{suc}} & 1 + N \\ \downarrow \alpha & & \downarrow I_1 + \alpha \\ [N \to N] & \xleftarrow{\psi} & 1 + [N \to N] \end{array}$$

The commutation of the above diagram is equivalent to :

$\alpha(0) = I_N$ and $\alpha(\mathrm{suc}\ n) = \mathrm{suc} \circ \alpha(n)$

$\alpha(0)(m) = m$ and $\alpha(\mathrm{suc}\ n)(m) = \mathrm{suc}\ (\alpha(n)(m))$.

α is the addition.

More generally any function defined by a primitive recursive scheme is the unique arrow making a similar diagram commute. Certain such diagrams, however, define functions which are not primitive recursive.

Theorem 2 asserts that $0 + \mathrm{suc}$ is an isomorphism, its inverse is obviously $\mathrm{null} + \mathrm{pred} : N \to 1 + N$ defined by $(\mathrm{null} + \mathrm{pred})\ (0) = \bot$ and

$(\mathrm{null} + \mathrm{pred})\ (n+1) = n$

The proof of Theorem 2 shows how to define pred in terms of suc.

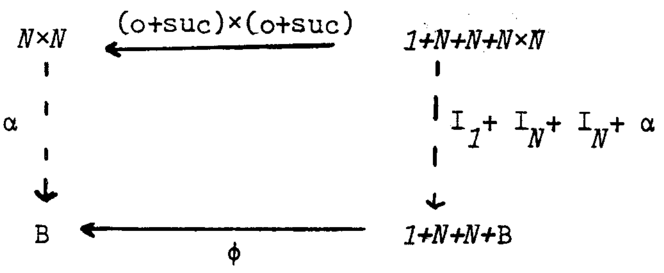
As a further example we shall show that even the equality predicate on N may be defined as the unique arrow from an initial algebra.

Let $TS = 1 + N + N + S$. The initial T-algebra is

$\phi : 1 + N + N + N \times N = (1+N)(1+N) \to N \times N$ defined by $\phi = (o+suc) \times (o+suc)$.

Let $B = \{true, false\}$ and $\psi : 1 + N + N + B \to B$ be defined by

$\phi(\bot) = true$, $\phi(n_1) = \phi(m_2) = false$ and $\phi(b) = b$.



α is the unique arrow : $N \times N \to B$ such that:

$\alpha(0,0) = true$, $\alpha(n+1,0) = \alpha(0,m+1) = false$ and $\alpha(n+1,m+1) = \alpha(n,m)$.

α is the equality predicate.

<u>Example 7</u> : Context-free languages.

All least fixpoints methods previously used in Computer Science are special cases of the more general category-theoretic initial fixpoints presented here. In particular the characterization of context free languages as least solutions of a set of equations can be carried through. Let Σ be an alphabet (not necessarily finite), then $P(\Sigma^*)$, the set of all languages over Σ, ordered by inclusion, is a complete lattice and, by standard methods, an ω-category. Let V_N be an alphabet of non-terminals (not necessarily finite) and p be a function : $V_N \to E$ where E is the set all expressions built from V_N, Σ, concatenation and union. On $P(\Sigma^*)$ concatenation and union are additive (they preserve arbitrary l.u.b's) and so they certainly are ω- functors. p clearly defines an ω-functor : $T : P(\Sigma^*)^{|V_N|} \to P(\Sigma^*)^{|V_N|}$ and the initial T-algebra TA $\subseteq$ A is the least $|V_N|$-uple A of subsets of $P(\Sigma^*)$ such that TA $\subseteq$ A. By theorem 2 TA = A.

Example 8 : Context-free grammars.

A much more interesting example concerns context-free grammars (as opposed to languages). A context-free grammar with n non-terminals can be viewed as an ω-endo-functor $T : Set^n \to Set^n$, in a manner similar to example 7 above, but when U (of subsets of Σ^*) is replaced by + (disjoint union) and . (concatenation of subsets of Σ^*) by × (product). For example the context-free grammar : $S \to a|aSa|aSSa|$ can be looked at as the functor : $T : \lambda S.\ \{a\} + \{a\} \times S \times \{a\} + \{a\} \times S \times S \times \{a\}$

The initial T-algebra $\phi : TA \to A$ consists of a set A isomorphic to $\{a\} + \{a\} \times A \times \{a\} + \{a\} \times A \times A \times \{a\}$, verifying the initiality property. A is isomorphic to the set of all parse trees, ϕ constructs parse trees and ϕ^{-1} decomposes parse trees. Note here that the function frontier : $fr : A \to \Sigma^*$ which assigns to each parse tree the word generated is not one-to-one and for $w \in \Sigma^*$: $|fr^{-1}(w)|$ is the multiplicity of w.

3. Data types as algebras

We now want to make our thesis precise: a data type is an object in a suitable category of domains (CPO^a will do for all applications here but *Dom* or other categories could be considered) equipped with certain operations.

Definition 1: A type t consists of a natural number n>o and n pairs of functors (S_i,T_i), $S_i,T_i : CPO^a \to CPO^a$, $i=1,\ldots,n$

Definition 2: A data type D of type $t=(n,S_i,T_i)$ consists of an ω-CPO D (with light notational ambiguity) and n (ω-continuous) functions $\phi_i : S_iD \to T_iD$.

In practical applications the functors S_i and T_i used are always "polynomial" (they are built from products and sums only) and indeed a data type is a domain equipped with a finite number of functions.

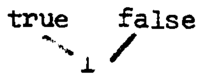
Definition 3: A homomorphism from D_1 to D_2 of type t is a function $f : D_1 \to D_2$ such that the following diagram commute:

$$\begin{array}{ccc} S_iD_1 & \xrightarrow{\phi_i^1} & T_iD_1 \\ {\scriptstyle S_if}\downarrow & & \downarrow{\scriptstyle T_if} \\ S_iD_2 & \xrightarrow[\phi_i^2]{} & T_iD_2 \end{array}$$

Homomorphisms will be used in Section 6 to study implementations. For the moment just notice that we have defined a category of data types of type t.

Basic data types

There is no real problem in defining the basic data types of usual programming languages as objects in CPO^a equipped with certain operations. The data type boolean for example would well be understood as the CPO Bool represented below:



Bool

equipped with the constant true, the unary operation $\sim$ ($\sim$true=false, $\sim$false=true, $\sim\bot=\bot$) and the binary operation $\vee$ ($\bot\vee\bot=\bot\vee$true=true$\vee\bot$=false$\vee\bot=\bot\vee$false=$\bot$, true$\vee$true=true$\vee$false=false$\vee$true=true, false$\vee$false=false).

All other connectives: $\wedge,\rightarrow,\leftrightarrow$ etc... can be defined from the above three.

The question could be raised whether we should not define $\bot\vee$true to be true (instead of $\bot$). This latter proposal is conceivable but would contradict the operational meaning that some want to give to $\bot$ (a non-terminating computation).

Because of the generality of circular definitions of data types and the extended facilities available to define functions on data types we think that it is possible (though we in no way suggest that it should be the case in programming languages) to use only a very few basic data types: *1* the one point domain, and *2* the two points domain for example.

1 *2*

Type-constructors

As explained in Section 3 a type-constructor is an ω-endo-functor in the category of domains, we shall list below some of the most interesting ones.

$\otimes : CPO^a \times CPO^a \to CPO^a$, corresponds to the categorical product in ω-CPO^* ($E_L \circ \otimes = \Pi \circ (E_L \times E_L)$ if Π is the categorical product in ω-CPO^*). If A and B are cpo's $A \otimes B$ consists of all pairs of objects, the first one in A, the second one in B ordered componentwise. If $f:A \to A'$ and $g:B \to B'$ are arrows in CPO^a then $f \otimes g = (f^L \Pi g^L, f^R \Pi g^R)$ for $(a \Pi b)(x,y) = (ax,by)$. That $\otimes$ is an ω-functor is easily checked with the help of Theorem 5, and so the ω-continuity of all functors to be defined now. There are some obvious arrows attached to $\otimes$: $P_1:A \otimes B \to A$, $P_2:A \otimes B \to B$. A slightly different product will be needed for defining stacks and lists.

$\times : CPO^a \times CPO^a \to CPO^a$ corresponds to the categorical product in ω-CPO^{**} (the arrows are very strict ω-continuous functions, those functions which send to bottom only the bottom element). If A and B are cpo's then $A \times B$ consists of only those couples (a,b) with $a \in A$ and $b \in B$ such that $a \neq \perp_A$ and $b \neq \perp_B$ or $a = \perp_A$ and $b = \perp_B$. It is easily seen to be a cpo. If $f:A \to A'$ and $g:b \to B'$ are arrows in CPO^a then $f \times g$ is the restriction of $f \otimes g$ to $A \times B$. This non-standard product has already been used by M. Gordon in his thesis. The obvious arrows $p_1:A \times B \to A$ and $p_2:A \times B \to B$ may be defined.

$+ : CPO^a \times CPO^a \to CPO^a$ corresponds to the categorical sum in ω-CPO^*, and in ω-CPO^{**}. A+B is the coalesced sum of A and B. Arrows $i_1:A \to A+B$, $i_2:B \to A+B$ and $d:A+B \to Bool$ may be defined.

$\oplus$: $CPO^a \times CPO^a \rightarrow CPO^a$ is the separated sum. i_1, i_2 and d may be defined as above.

$\rightarrow$: $CPO^a \times CPO^a \rightarrow CPO^a$ is the functor space functor. $\rightarrow(A,B)$ is $[A\rightarrow B]$ in our notation.

App: $[A\rightarrow B]\otimes A\rightarrow B$, Abst: $[A\times B\rightarrow C]\rightarrow[A\rightarrow[B\rightarrow C]]$ and Y: $[A\rightarrow A]\rightarrow A$ may be defined.

As was mentioned above a power constructor P, or more precisely a number of such constructors have been studied; they will not be used in the present work. The next paragraph exemplifies circular definitions of data types. For example it will be shown how, given a data type A, it is possible to define the data type Stacks of A (StackA). It is only in the next section that it will be shown how these definitions amount to making Stack a type constructor.

Circularly defined data types

This paragraph, and the next one, proceed uniquely by examples. Their purpose is to show that many usual data types are indeed defined circularly, and that circular definitions define not only a certain domain but also certain operations on it.

1) Simple data types

Those are the data types built from basic data types by type constructors and circular definitions.

Natural numbers

In the preceding section the natural numbers were defined circularly as an initial T-algebra for a functor T on Set. For computer science purposes it seems preferable to define them as an initial T-algebra for a functor T on CPO^a. By analogy with the case of Set one could think of using $T' = \mathit{1} + I_{CPO^a}$. This does not give a satisfactory solution: the initial T'-algebra is $\mathit{1}$ (remember that + is the coalesced sum). Scott has proposed, for the natural numbers, what amounts to the initial T-algebra for $T = \mathit{2} + I_{CPO^a}$. one may easily verify that the initial T-algebra η: $\mathit{2}+N_\perp \rightarrow N_\perp$ is such that $N_\perp = \{\perp, o, 1, 2, \ldots, n, \ldots\}$ with $\perp \sqsubseteq n$ and $n \sqsubseteq n$ for any $n \in N_\perp$, those being the only ordered pairs. Pictorially:

o 1 2 ... n ...

⊥

$N_\perp$

⊤ o 1 2 ... n ...

⊥

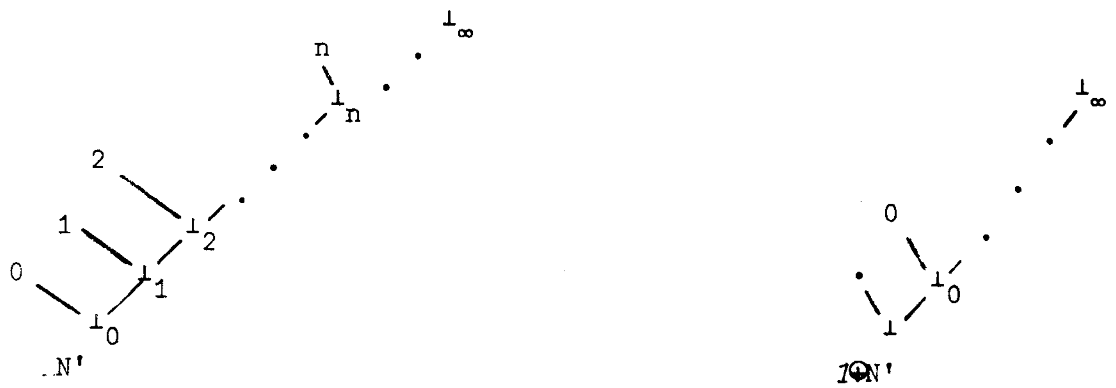
$\mathit{2}+N_\perp$

And η is such that $\eta(\bot)=\bot$, $\eta(\top)=o$ and $\eta(n)= n+1$. By Theorem 2 η is an isomorphism and the data type natural numbers comes equipped with two arrows: η: $2+N_\bot \to N_\bot$ and η^{-1}: $N_\bot \to 2+N_\bot$. To convince the reader that η and η^{-1} are sufficient to define all usual arithmetic functions, just notice that o: $2 \to N_\bot$ may be defined by $o= \eta \circ i_1$, suc: $N_\bot \to N_\bot$ by suc= $\eta \circ i_2$, null: $N_\bot \to$Bool by null= $d \circ \eta^{-1}$, pred: $N_\bot \to N_\bot$ by pred(n)= if $d(\eta^{-1}(n))$ then o else $\eta^{-1}(n)$. All useful arithmetical functions may then be defined by circular definitions. In particular a primitive recursive definition: f(o,n)= h(n), f(m+1 ,n)= g(m,n,f(m,n)) may be translated by:

f(m,n)= if null(m) then h(n) else g(m,n,f(m,n)). It must be noticed that both functions η and η^{-1} must be given; η alone does not allow the definition of null and pred, in the absence of the equality predicate which we have no reason to suppose given or even computable. once pred and null are given, equality may be defined either by the usual primitive recursive definition or by:

eq(m,n)= if null(m) then null(n) else if null(n) then false else eq(pre(m),pred(n)).

In $N_\bot$ the least solution of the equation x= suc(x) is $\bot$. Another candidate for the data type natural numbers is the initial T'-algebra for $T'= 1 \oplus I_{CPO^a}$, using the separated sum. This initial algebra η': T' N'$\to$N' looks the following way:



with $\eta(\bot)= \bot_o$, $\eta(.)= o$, $\eta(\bot_n)= \bot_{n+1}$, $\eta(n)= n+1$, $\eta(\bot_\infty)= \bot_\infty$.

The first version $N_\bot$ we proposed is simpler but the second one N' shows how infinite objects such as $\bot_\infty$ may enter the initial algebra.

Fun = proc(fun) fun

The data type defined by the equation $X = [X \to X]$, corresponding to the ALGOL 68 definition of the title is the initial T-algebra for $T = \to$. It is 1, the one-point domain, the arrow being the identity. This example is given, not because of its usefulness, but because it is one of the circular definitions which could not be understood in the framework set by the ADJ group.

Stacks

The data type Stack A, whenever A is a data type, is the initial T-algebra for $T = 2 + A \times I_{CPO^a}$.

Note that we use the coalesced sums and products. It is easy to check that the initial T-algebra, $cs : 2+A\times S \to S$ is the following.

S is the set of all finite sequences of elements of A different from $\bot_A$ ordered coordinatewise and one bottom element $\bot$.

cs is defined by: $cs(\bot)=\bot$, $cs(\top)=()$ the empty sequence, $cs(a,s)=(a,s)$ the concatenation of a and s, if $s \neq \bot$, and $cs(a,\bot)=\bot$.

Clearly $\Lambda = cs(\top)$ for non-empty s $pop(s) = p_2 \circ cs^{-1}(s)$

$push = cs \circ i_2$ " $top(s) = p_1 \circ cs^{-1}(s)$

$empty = d \circ cs^{-1}$

It does not bother us that pop and top are only partial functions: everything may be done in terms of cs^{-1}. The above way of defining stacks may be compared with ADJ (75) where stacks are defined as initial equational algebras. Our approach answers three criticisms that could be made on ADJ's : it deals with partial orders and not only sets, there is no need to introduce some special object of type A to be $top(\Lambda)$, and most important the equations laid down by ADJ may be justified and shown to be sufficient to characterize stacks. On this third point the reader should notice, as will be explained in section 6, that in most implementations of stacks pop(push(d,s)) differs from s. Our approach is also much closer to what programming languages do allow: circular mode definitions are allowed in ALGOL 68 whereas no programming language allows the definition of an initial equational algebra , obviously many equational specifications do not make any sense for programming and functional data types are not definable equationally.

The use of the coalesced product (×) instead of the usual one (⊗) enables us to avoid the introduction of infinite stacks; a similar equation with × instead of ⊗ defines a data type which contains infinite stacks. This data type has been found useful by some and seems even implementable (see Friedman and Wise (76) and Henderson and Morris (76)). The two data types should be clearly distinguished.

Lists:

The intuition suggests:

$$\text{List } A \cong A + \text{Stack}(\text{List } A)$$

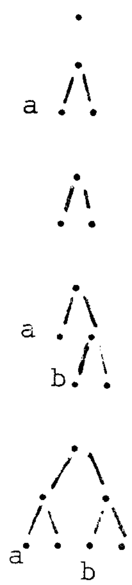
It is only in the next section that it will be shown that Stack is a mode-constructor and that the above definition will be given its strict meaning.

Lisp-lists

Lists in LISP may be informally defined by: a list is either empty or the concatenation of a head and a tail, the head being either an atom or a list, the tail being always a list. This suggests the following definition: $LispA \cong 2+(A+LispA)\times LispA$. Let T be the functor: $\lambda x.(2+(A+x)\times x)$ and c_L: TL→L the initial T-algebra.

LispA does indeed, as the reader may care to check, consist of what one could expect: one bottom element and all finite non-empty binary trees the right leaves of which are unlabelled (they represent empty lists) and the left leaves of which are either unlabelled or labelled by a non-bottom element of A. If the tree consists of only one leaf it has to be considered a right leaf.

Examples:



() the empty list

(a)

(())

(a.(b)) = (a,b)

(a).(b)) = ((a), b)

cl: $2+(A+\text{LispA})\times\text{LispA}\rightarrow\text{LispA}$ is such that:

$$cl(\bot)=\bot \quad , \quad cl(\top)=() \quad , \quad cl(a,\ell)=(a.\ell)$$

$$2+(A+\text{LispA})\times\text{LispA}=2+(A\times\text{LispA}+\text{LispA}\times\text{LispA})=(2+A\times\text{LispA})+\text{LispA}\times\text{LispA}$$

which is obviously desirable.

Binary trees

Labelled binary trees suggest the following definition:

$\text{BtreeA} \cong A+A\times\text{BtreeA}\times\text{BtreeA}$.

The reader may care to check that the initial algebra consists of all finite binary trees all the nodes of which are labelled by non-bottom elements of A, and one bottom element. The arrows obtained construct ($cb : A+A\times\text{BtreeA}\times\text{BtreeA}\rightarrow\text{BtreeA}$) and decompose ($cb^{-1}:\text{BtreeA}\rightarrow A+A\times\text{BtreeA}\times\text{BtreeA}$) trees.

Had we chosen to solve $T \cong A+A\otimes T\otimes T$ we would have defined an initial algebra on a domain containing finite and infinite labelled binary trees.

Trees and Forests

A labelled tree consists either of a single labelled node or of a labelled node and a forest. A forest is either empty or consists of a tree and a forest. Trees and forests are defined by the pair of equations:

$T\cong A+A\times F$

$F\cong 2+T\times F$

It will be shown in Section 4 that forests could equivalently be defined as stacks of trees , after having defined trees as either a single labelled node or a labelled node and a stack of trees. In other words T and F may be defined by: $T\cong A+A\ \mathrm{Stack}(T)$

$F= \mathrm{Stack}(T)$

where the mutually circular definition has been eliminated.

2) Composite data types

Many data types are not simply built up from basic data types with the help of type constructors and circular definitions but defined, somewhat indirectly, by first defining a simple data type and then some new functions on this data type and possibly forgetting some of the old functions. The carrier of a composite data type will be the same as that of the simple data type from which it is built but the operations will be different. Two examples will be analysed here: Queues and Arrays.

Queues

Queues are of the same type as Stacks but the push and pop procedures interact in a different way in the sense that the element popped will be the one which has been pushed on first (and not the last one as in Stacks). Queues are a very useful and interesting data type which seems to have escaped the attention of previous researchers. The obvious idea is to define Queues from Stacks by defining new pop and top functions and then forgetting about the old ones. Stacks of A were defined as two inverse functions:

cs: $2+A\times StackA\rightarrow StackA$ the initial algebra and

cs^{-1}: $StackA\rightarrow 2+A\times StackA$ the empty+top×pop arrow.

We may define dq: $StackA\rightarrow 2+A\times StackA$ by:

$$dq(s)= \text{if } empty(s) \text{ or } empty(pop(s)) \text{ then } cs^{-1}(s) \text{ else } \langle p_1(dq(pop(s))), push(top(s), p_2(dq(pop(s))))\rangle$$

The above definition simply translates the idea that the new top and pop operations operate at the end of the stack. It would also have been possible to leave top and pop unchanged and define a new push operation. Our formalism enables us to prove the equivalence of these two ways of defining queues but we shall not attempt to do that here. The data type Queues then consists of a domain QueueA= StackA and two functions:

cs: $2+A\times QueueA\rightarrow QueueA$

dq: $QueueA\rightarrow 2+A\times QueueA$

cs and dq are not inverses but properties may be proved about them as will be seen in Section 5 when a characterization of Queues as an initial algebra in a certain equational class will be proved. Let us define:

top': $QueueA\rightarrow A$ by top'(s)= $p_1(dq(s))$ for nonempty s and

pop': $QueueA\rightarrow QueueA$ by pop'(s)= $p_2(dq(s))$ for nonempty s.

Arrays

Infinite one-dimensional arrays of A may be defined from the data type $[N_\perp\rightarrow A]$ of functions from the natural numbers to A. Arrays must be equipped with two functions: access: $Arr\times N\rightarrow A$ and update: $Arr\times N\times A\rightarrow Arr$. The function access may easily be seen to be the function eval. The function update is defined by Currying and use of abstraction by:

update(g,n,a)= λm. if m=n then a else eval(g,m)

It is now obvious that: access(update(g,n,a),m)= if m=n then a else access(g,m).

4. The initial fixpoint operator

In this section we will show that the transformation which sends an ω-functor to its initial fixpoint can itself be defined as an ω-functor. This will in turn be used to show that other important constructs, notably data-types with parameters, are ω-functors. As another application, we show that the well-known "reduction of simultaneous recursion to iterated recursion" (de Bakker '71) generalizes to ω-categories. This provides a useful technique for demonstrating the equivalence of data types, as we shall see in Section 6.

We begin with some considerations on functor categories. The following notation is adopted: C is an ω-category, while A,B are arbitrary categories. $[B \to C]$ is the category with objects the ω-functors from A to C and arrows the natural transformations (with "vertical" composition) between such functors. For composition of natural transformations we follow the notation of Herrlich and Strecker: $\circ$ for the vertical, $*$ for the horizontal composition.

Lemma 1. $[A \to C]$ is an ω-category.

Sketch of proof. The initial object of $[A \to C]$ is the constant functor with value $\perp_C$. C^A is an ω-category in which limits and colimits are computed pointwise (MacLane Ch.V,3). We seek to show that, if $T = F_0 \xrightarrow{\tau_0} F_1 \xrightarrow{\tau_1} \cdots$ is an ω-chain in $[A \to C]$, then the colimit $\Phi : T \to F$ of T in C^A is also the colimit in $[A \to C]$; in other words, that F is ω-continuous. To see this, let $\Gamma = A_0 \xrightarrow{g_0} A_1 \longrightarrow \cdots$ be any ω-chain in A, with colimit $\mu : \Gamma \to A_\omega$, and let D be the infinite two-dimensional diagram with rows $F_i \Gamma$ $(i=0,1,\ldots)$ and with vertical arrows $(\tau_i)_{A_j} : F_i A_j \to F_{i+1} A_j$. D commutes by naturality of the τ_i. Taking the colimit of D first by rows and then by columns we find that $F_\omega \mu$ is the colimit of $F_\omega \Gamma$.

A more detailed proof of this lemma may be found in Lehmann (76).

We will define the initial fixpoint operator, $Y:[C\to C]\to C$, in terms of the colimit functor $\underrightarrow{Lim}:C^{\omega}\to C$. We recall (MacLane Ch.IV,2) that $\underrightarrow{Lim}$ may be defined as the left adjoint of the diagonal functor $\Delta:C\to C^{\omega}$. (Strictly speaking, this is not a proper definition, since adjoints are not uniquely determined. We have to suppose that a <u>particular</u> adjunction is chosen once and for all.)

<u>Lemma 2</u>. For any ω-functor $F:C\to C$, let $S(F)$ be the ω-diagram $\perp \xrightarrow{f} F\perp \xrightarrow{Ff} F^2\perp \to \ldots$. For any natural transformation $\tau:F\to G$ (G an ω-functor), let $S(\tau)_n$ (for $n=0,1,\ldots$) be $\tau^n:F^n\perp\to G^n\perp$, where τ^n is the n-fold composition $\tau*\tau*\ldots*\tau$. Then S is a functor from $[C\to C]$ to C^{ω} .

<u>Proof</u>. To show that $S(\tau)$ is a natural transformation, we must show that the diagram (note: $\tau^i_\perp$ means $(\tau^i)_\perp$)

$$(1)\qquad \begin{array}{ccccccc} \perp & \xrightarrow{f} & F\perp & \xrightarrow{Ff} & F^2\perp & \longrightarrow & \\ \downarrow & & \downarrow{\scriptstyle \tau_\perp} & & \downarrow{\scriptstyle \tau^2_\perp} & & \cdots\cdot \\ \perp & \xrightarrow{g} & G\perp & \xrightarrow{Gg} & G^2\perp & \longrightarrow & \end{array}$$

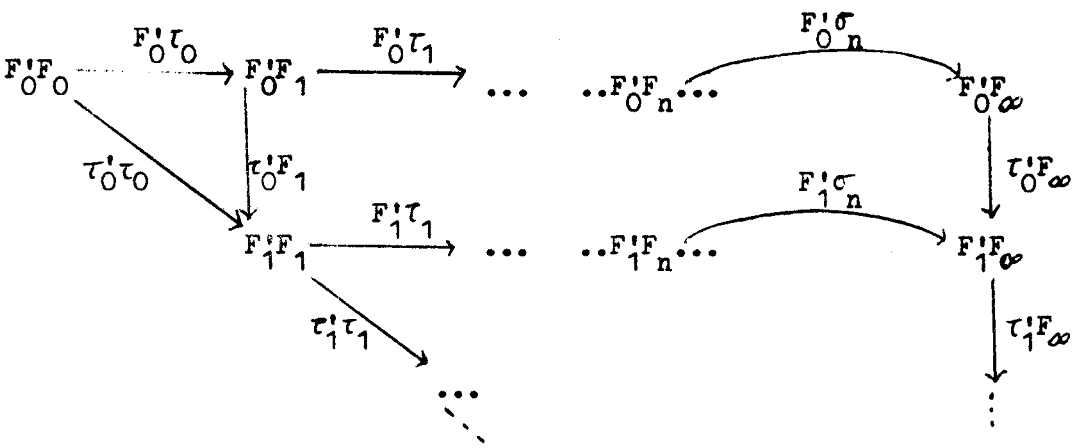
commutes. The 0^{th}(leftmost) square of (1) commutes trivially. Suppose that the n^{th} square commutes. Then

$$\begin{array}{ccc} FF^n\perp & \xrightarrow{FF^nf} & FF^{n+1}\perp \\ {\scriptstyle F\tau^n_\perp}\downarrow & & \downarrow{\scriptstyle F\tau^{n+1}_\perp} \\ FG^n\perp & \xrightarrow{FG^ng} & FG^{n+1}\perp \\ {\scriptstyle \tau_{G^n\perp}}\downarrow & & \downarrow{\scriptstyle \tau_{G^{n+1}\perp}} \\ GG^n\perp & \xrightarrow{GG^ng} & GG^{n+1}\perp \end{array}$$

commutes - the upper half by applying F to the n^{th} square, the lower half since τ is natural. But this means that the $n+1^{th}$ square of (1) commutes, since $\tau_{\perp}^{k+1} = \tau_{G^k\perp} \circ F\tau_{\perp}^{k}$ (k=n,n+1). Thus (1) commutes. That S preserves identities is trivial, and that it preserves composition is an immediate consequence of the interchange law.

Lemma 3. Suppose that $T = F_0 \xrightarrow{\tau_0} F_1 \to \ldots$, $T' = F'_0 \xrightarrow{\tau'_0} F'_1 \to \ldots$ are ω-chains in $[C \to C]$, with colimit cones $\sigma : T \to F_\infty$, $\sigma' : T' \to F'_\infty$. Then the ω-chain $T'' = F'_0F_0 \xrightarrow{\tau'_0 * \tau_0} F'_1F_1 \longrightarrow \cdots$ has the colimit $\sigma'' : T'' \to F'_\infty F_\infty$, where $\sigma''_n = \sigma'_n * \sigma_n$.

Proof. Consider the infinite diagram



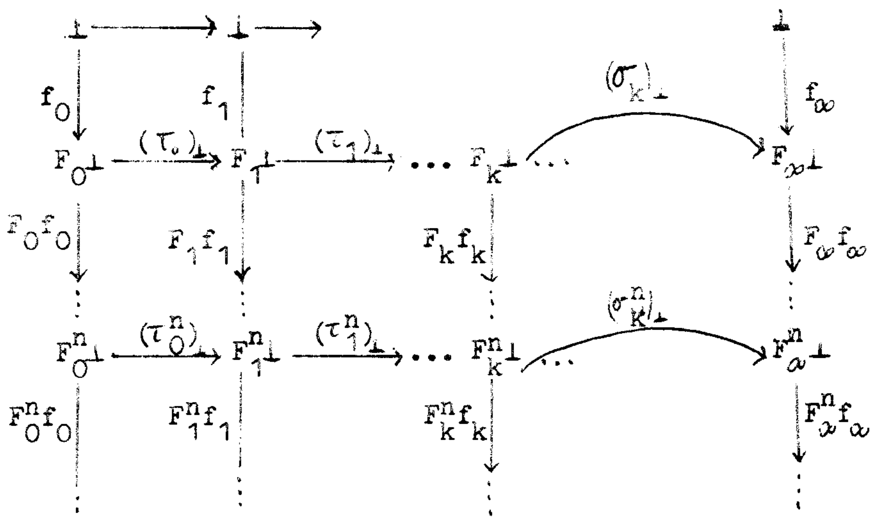
Since the F'_n are ω-continuous, the colimits of the rows are as indicated. The colimit of the right-hand column is constituted by the arrows $\sigma'_nF_\infty : F'_nF_\infty \to F'_\infty F_\infty$, n=0,1,... . Hence the colimit of the diagonal, computed by rows, is constituted by the arrows $(\sigma'_nF_\infty)\circ(F'_n\sigma_n) = \sigma'_n * \sigma_n$.

Definition 1. $Y:[C\to C]\to C$ is $\underrightarrow{\mathrm{Lim}}\circ S$, where S is as defined in Lemma 1.

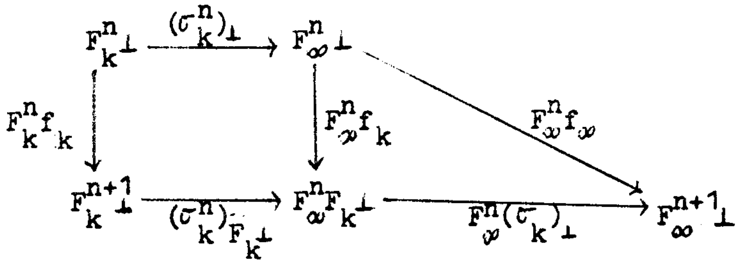
Theorem 1. Y is an ω-functor.

Proof. $\underrightarrow{\mathrm{Lim}}$, as a left adjoint, preserves all colimits. The rest of the proof is concerned with showing that S is ω-continuous. Let $\Phi = F_0 \xrightarrow{\tau_0} F_1 \xrightarrow{\tau_1} \ldots$ be an ω-chain in $[C\to C]$ with colimit $\sigma:\Phi\to F_\infty$. The translation of σ by S is the following infinite diagram:

(2)



By Lemma 3, together with the fact that colimits in $[C\to C]$ are computed pointwise, the colimits of the rows are as indicated in the diagram. Let us prove that the arrows $F^n_\infty f_\infty$ mediate between the successive cones, as indicated. In the first place, f_∞ (the unique arrow from $\perp$ to $F_\infty\perp$) mediates trivially. Now, for any $n \geq 1$, $k \geq 0$, the following diagram commutes:

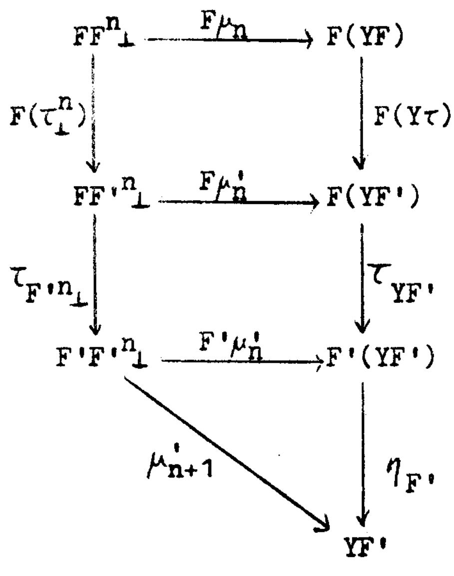


Since $F^n_\infty(\sigma_k)_\perp \circ (\sigma_k^{\ n})_{F_k\perp} = (\sigma_k^{n+1})_\perp$, it follows that $F^n_\infty f_\infty$ mediates, as stated. This shows that (2) is a colimiting diagram for $S\Phi$.

The following lemma gives a characterization of the action of Y on arrows which is often more convenient to work with than Definition 1:

<u>Lemma 4</u>. Suppose that $\tau : F \to F'$ is a natural transformation, where $F, F' \in [C \to C]$. Then $Y\tau$ is the unique arrow from the initial F-algebra η_F to the F-algebra $\eta_{F'} \circ \tau_{YF'}$.

<u>Proof</u>. Let Δ be the chain $\perp \xrightarrow{f} F \xrightarrow{Ff} \ldots$, with colimiting cone $\mu : \Delta \to YF$; and similarly for Δ', μ'. Define the cone $\nu : F\Delta \to YF'$ by: $\nu_n = \mu'_{n+1} \circ \tau_\perp^{n+1}$. We shall prove that $Y\tau \cdot \eta_F$ and $\eta_{F'} \cdot \tau_{YF'} \circ F(Y\tau)$ are equal by showing that each is the mediating arrow from $F\mu$ to ν. For $Y\tau \cdot \eta_F$ this is immediate. For $\eta_{F'} \circ \tau_{YF'} \cdot F(Y\tau)$, consider the commuting diagram (for each $n \geq 0$):



Since $\tau_{F',\perp} \circ F(\tau_{\perp}^{n}) = \tau_{\perp}^{n+1}$, the perimeter of this diagram gives the desired result.

The next lemma introduces an abstraction operator for functors.

Lemma 5. Define $\mathrm{Abst}:[A \times B \to C] \to [A \to [B \to C]]$ on objects (i.e. functors $F:A \times B \to C$) by:

$$\mathrm{Abst}(F)(X) = F(X,-), \quad \text{for } x \in A$$

$$\mathrm{Abst}(F)(\alpha)_Y = F(\alpha, I_Y), \quad \text{for } \alpha: A \to A';$$

and on arrows (natural transformations) by:

$$(\mathrm{Abst}(\tau)_X)_Y = \tau_{\langle X,Y \rangle} .$$

Then Abst is an isomorphism.

- For the proof we refer to Herrlich and Strecker(73),Theorem 15.9; it is routine to check that various functors which appear in the proof are ω-continuous, so that the change in the meaning of [...→...] does not materially affect the proof.

At last we are in a position to understand data type definitions with parameters. A parameterized data type, in our view, is simply an ω-functor (from a category thought of as the parameter category into a data type category). One of the commonest situations is the following: an ω-functor $F:A \times C \to C$ yields the parameterized data type $Y \circ (\mathrm{Abst}(F)):A \to C$.

Example. Stack (Sec.3) must be regarded as a parameterized data type, if we are to make sense of the suggested circular definition

(2a) $\qquad \mathrm{List}A \cong A + \mathrm{Stack}(\mathrm{List}A).$

Here, Stack is an ω-functor, and (2a) defines List as $Y \circ (\mathrm{Abst}(F))$, where F is $\lambda\langle A,C \rangle.A + \mathrm{Stack}C$. It may be asked, what is the exact relation between List and Lisp (as defined in Sec.3)? We will find that such questions can be answered by the help of Theorem 2.

Theorem 2 deals with the reduction of simultaneous recursion to iterated recursion. It says , in effect, that a pair of simultaneous "equations"

$$X \cong F(X,Z)$$

$$Z \cong G(X,Z)$$

can be solved by first solving for Z in terms of X from the second equation, then substituting in the first equation.

<u>Notation</u>. $F,G:C \times C \to C$ are ω-functors. $\langle F,G\rangle : C \times C \to C \times C : V \mapsto \langle FV,GV\rangle$. G_X is $\lambda Z.G(X,Z)$ (i.e. $G(X,-)$). Z_- is the functor $Y \circ (\mathrm{Abst}G)$; thus we write Z_X, Z_θ (where $\theta : X \to X'$), etc. $\bar{F}$ is $\lambda X.F(X,Z_X)$, $\bar{X}$ is $Y(\bar{F})$, and $\bar{Z}$ is $Z_{\bar{X}}$.

<u>Theorem 2</u>. $\langle \eta_{\bar{F}}, \eta_{G_{\bar{X}}}\rangle$ is initial for $\langle F,G\rangle$.

<u>Proof</u>. Note first that, by Lemma 4, Z_- may be characterized as follows. Z_X is $Y(G_X)$; if $\theta : X \to X'$, then Z_θ is determined by

$$(3) \qquad \begin{array}{ccccc} Z_X & \xleftarrow{\eta_{G_X}} & G_X(Z_X) & = & G(X,Z_X) \\ \vdots\, Z_\theta & & \vdots\, G_X(Z_\theta) & = & G(I_X,Z_\theta) \\ Z_{X'} & \xleftarrow[\eta_{G_{X'}} \circ G(\theta, I_{Z_{X'}})]{} & G_X(Z_{X'}) & = & G(X,Z_{X'}) \end{array}$$

Now, suppose that $p:F(X',Z') \to X'$, $q:G(X',Z') \to Z'$, so that $\langle p,q\rangle$ is an $\langle F,G\rangle$-algebra. We will prove that there is at most one arrow from $\langle \eta_{\bar{F}}, \eta_{G_{\bar{X}}}\rangle$ to $\langle p,q\rangle$. Suppose, then, that $\langle \alpha,\beta\rangle$ is such an arrow, i.e.

$$(4)\ \text{a)} \quad \begin{array}{ccc} \bar{X} & \xleftarrow{\eta_{\bar{F}}} & F(\bar{X},\bar{Z}) \\ \alpha \downarrow & & \downarrow F(\alpha,\beta) \\ X' & \xleftarrow[p]{} & F(X',Z') \end{array} \qquad \text{b)} \quad \begin{array}{ccc} \bar{Z} & \xleftarrow{\eta_{G_{\bar{X}}}} & G(\bar{X},\bar{Z}) \\ \beta \downarrow & & \downarrow G(\alpha,\beta) \\ Z' & \xleftarrow[q]{} & G(X',Z') \end{array}$$

Let ζ be the arrow from $\eta_{G_{X'}}$ to the $G_{X'}$-algebra $q: G_{X'}(Z') \to Z'$. Using (3), the following commutes:

$$
(5)\qquad
\begin{array}{ccc}
Z_{\bar{X}} & \xleftarrow{\ \eta_{G_{\bar{X}}}\ } & G_{\bar{X}}(\bar{Z}) \\
{\scriptstyle Z_\alpha}\big\downarrow & & \big\downarrow {\scriptstyle G(\alpha, I_{Z_{X'}}) \circ G(I_{\bar{X}}, Z) \;=\; G(\alpha, Z_\alpha)} \\
Z_{X'} & \xleftarrow{\ \eta_{G_{X'}}\ } & G_{X'}(Z_{X'}) \\
{\scriptstyle \zeta}\big\downarrow & & \big\downarrow {\scriptstyle G_{X'}(\zeta) \;=\; G(I_{X'}, \zeta)} \\
Z' & \xleftarrow{\ q\ } & G_{X'}(Z')
\end{array}
$$

(4)b) and (5) give, respectively:

$$
(6)\qquad
\begin{array}{ccc}
\bar{Z} & \xleftarrow{\ \eta_{G_{\bar{X}}}\ } & G_{\bar{X}}(\bar{Z}) \\
{\scriptstyle \beta}\big\downarrow & & \big\downarrow {\scriptstyle G(I_{\bar{X}}, \beta) \;=\; G_{\bar{X}}(\beta)} \\
Z' & \xleftarrow[\ q \circ G(\alpha, I_{Z'})\ ]{} & G_{\bar{X}}(Z')
\end{array}
\qquad \text{and} \qquad
\begin{array}{ccc}
\bar{Z} & \xleftarrow{\ \eta_{G_{\bar{X}}}\ } & G_{\bar{X}}(\bar{Z}) \\
{\scriptstyle \zeta \circ Z_\alpha}\big\downarrow & & \big\downarrow {\scriptstyle G(I_{\bar{X}}, \zeta \circ Z_\alpha)} \\
Z' & \xleftarrow[\ q \cdot G(\alpha, I_{Z'})\ ]{} & G_{\bar{X}}(Z')
\end{array}
$$

By initiality of $\eta_{G_{\bar{X}}}$, $\beta = \zeta \circ Z_\alpha$.

Next, (4)a) gives

$$
(7)\qquad
\begin{array}{ccc}
\bar{X} & \xleftarrow{\ \bar{F}\ } & \bar{F}(\bar{X}) \;=\; F(\bar{X}, Z_{\bar{X}}) \\
{\scriptstyle \alpha}\big\downarrow & & \big\downarrow {\scriptstyle \bar{F}(\alpha) \;=\; F(\alpha, Z_\alpha)} \\
X' & \xleftarrow[\ p\ F(I_{X'}, \zeta)\ ]{} & \bar{F}(X') \;=\; F(X', Z_{X'})
\end{array}
$$

By initiality of $\eta_{\bar{F}}$, α is uniquely determined; hence β is also.

On the other hand, we see that if we start by defining α as being given by (7), and put $\beta = \zeta \circ Z_\alpha$, then all diagrams (4)-(7) commute, and in particular $\langle \alpha, \beta \rangle$ is an arrow from $\langle \eta_{\bar{F}}, \eta_{G_{\bar{X}}} \rangle$ to $\langle p, q \rangle$.

Corollary 1. For any ω-functor $F:C \times C \to C$,

$$Y(\lambda X.Y(\lambda Z.F(X,Z)) \cong Y(\lambda X.F(X,X)).$$

Proof. Consider the two ways (orders) of solving the pair

$$X \cong Z$$
$$Z \cong F(X,Z)$$

Theorem 2 assures us that the "naive" solutions are correct. In detail: G_X is here $F(X,-)$, Z_X is $Y(\lambda Z.F(X,Z)$, $\overline{F}$ is $\lambda X.Y(\lambda Z.F(X,Z))$, and $\langle\overline{X},\overline{Z}\rangle = \langle Y(\lambda X.Y(\lambda Z.F(X,Z))),Z_{\overline{X}}\rangle$ is the carrier (codomain) of an initial solution. Then, interchanging the roles of X,Z, we have (in a notation symmetrical to that used previously, whose interpretation should be clear): F_Z is $\lambda X.Z$, X_Z is Z, G^* is $\lambda Z.F(Z,Z)$, and $\langle X^*,Z^*\rangle = \langle Y(\lambda Z.F(Z,Z),Y(\lambda Z.F(Z,Z)\rangle$ is the carrier of an initial solution. Since all initial algebras for $\langle F,G\rangle$ are isomorphic, the result follows.

Corollary 2. BtreeA $\cong$ ForA .

Proof. Apply Corollary 1, with F(X,Z) as $2 + A \times X \times Z$.

The mere existence of an isomorphism between BtreeA and ForA is not in itself a very interesting result. However, as we shall see in Section 6, a closer inspection reveals much more than this: it gives us the "representation of forests by binary trees". Likewise for:

Corollary 3. ListA $\cong$ A + LispA.

Proof. Similar to Corollary 2, starting with the equations

$$X \cong A + Z$$
$$Z \cong 2 + X \times Z.$$

5. Methods of definition and proof

Within the framework established in this paper, we have a number of methods available for defining data types, data structures (elements of data types) and functions over data types, and for proving properties of defined entities. Some of these methods are brought together in the present section. The short subsection on definitions is little more than a summary of points made in scattered form elsewhere in the paper. Subsection B is concerned with induction versus initiality as a method of proof. Subsection C takes up the question of how we might prove that our definitions of particular data types yield the "right" properties.

A. Definitions. For definitions of data types we can make a fourfold classification. First, we have the ad hoc definition of some basic data types. Secondly, we can apply a functor - either one of the standard functors or a "parameterized data type" to an already defined type. Thirdly, a data type can be introduced as the initial fixpoint of a functor. Finally, we can define "composite" data types, in which the carrier is taken to be given to be given by one of the preceding three methods, but new defined operations are introduced. Examples of all these methods were provided in Section 3.

Turning to the question of specifying (defining) an element of a given data type, the only point worthy of special comment is that arbitrary recursive specifications are admissible. For procedural data types this is commonplace. For a non-procedural data type, an example was suggested in Sec. 3 . Admittedly, the example is not impressive; the method only becomes really useful when data types which admit infinite structures (say, infinite trees) are studied.

The basic type-constructors have, in view of their categorical definition, certain functions associated with them: projections and injections associated with product and sum, evaluation and abstraction maps associated with the function space, and so on. And, of course, every functor gives us the means to introduce functions: if a function $f:X \to Z$ and functor F have been defined, then we have also the function $Ff:FX \to FZ$. So, for example, functions defined on the natural numbers extend at once to arrays and stacks of numbers. APL is probably the best-known example of a programming language that makes systematic use of this possibility.

Initiality provides yet another way of defining functions. The most important example is the definition of the inverse isomorphism η_F^{-1} for an initial F-algebra $\eta_F:FX_0 \to X_0$ Once the inverse is available, further uses of initiality for defining functions can be eliminated in favour of recursive definitions. In detail: suppose we have a definition of a function $f:X_0 \to X$ by initiality, namely as the unique function satisfying

(1) $$f \circ \eta_F = g \circ Ff$$

where $g:FX \to X$ is defined previously. Then (1) can be rewritten

(2) $$f = g \circ Ff \circ \eta_F^{-1}$$

and this has the form $f = Gf$ for a continuous functional G. Thus f may alternatively be introduced as the (least) solution of (2), construed as a recursive definition.

What we lose by this reduction is the unicity of the solution of (2) - an aspect that may be of great importance in proving properties of f, or of the domains X_0, X. Indeed, it is as a principle of proof, rather than of definition, that initiality is most significant.

B. Initiality and induction. Examples of the use of initiality in proofs are to be found throughout the paper. A very interesting question suggests itself: do we need any principles of proof about data types that cannot be reduced to initiality? One may think, for example, of structural induction. For StackA, this might be formulated as:

$$(3)\qquad \frac{\bot \in S \qquad \Lambda \in S \qquad x \in S \Rightarrow \forall a \in A.\,push(a,x) \in S}{Stack \subseteq S}$$

Can (3) be derived from initiality? At first it may seem as though it can. A set will naturally be construed as a map from StackA into Bool (or perhaps into 2), and thus we reduce (3) to, say:

$$(4)\qquad \frac{f(\bot)=\bot \qquad f(\Lambda)=tt \qquad f(x)=tt \Rightarrow \forall a \in A.\,f(push(a,x))=tt}{f = t}$$

where $t{:}StackA \rightarrow Bool$ is the strict map which sends every non-$\bot$ stack to tt. (4) can indeed be derived from initiality, with a bit of effort (it is not quite trivial). But, of course, the "reduction" of (3) to (4) is faulty. (3) is formulated for arbitrary predicates(=sets), (4) only for continuous predicates. For the usual applications, (4) is insufficient. For example, structural induction is often used to prove statements of the form

$$(5)\qquad \forall x.f(x) = g(x)\ .$$

But equality is not, in general, a continuous predicate.

(3) can indeed be proved quite easily by going back to the explicit construction of StackA as the colimit of a certain ω-chain Γ. The proof goes by way of showing that every element of StackA is already "in" one of the terms of Γ; or, more precisely, that for every $x \in StackA$ there exists n such that $\mu_n^R \mu_n(x) = x$, where $\mu{:}\Gamma \rightarrow StackA$ is the colimiting cone. This shows that every stack is either $\bot$ or Λ or can be obtained from Λ by

a finite number of pushes, which is essentially (3). A similar argument applies in the case of any data type defined as the initial fixpoint of a "polynomial" functor: that is, a functor built up by composition (and transposition of variables) from +, x, constant and identity functors. We will not prove this result here, or even state it precisely. This is partly because we are not satisfied that the result is of the right level of generality (though it certainly covers all the usual cases of structural induction). But the major reason why we do not trouble to make it precise is that we do not, after all, need structural induction.

We should admit at this point that the argument developed so far in this section is not quite satisfactory. We have argued, in effect, that (3) (say) cannot be derived directly from initiality, while it can be obtained indirectly via a particular construction of an initial algebra for $2 + A \times -$. What is unsatisfactory is that we are unable to say exactly what a "direct" derivation is. Despite this, it seems clear that the question whether in proofs about data types we can work just with the abstract characterization by initiality, or must go via the concrete construction, has some methodological significance. At the least, proofs using initiality generalize more readily than proofs using structural induction. For example, where polynomial functors and their initial algebras are concerned, a proof by initiality typically does not make use of the detailed properties of + and x, and will work equally well if $\oplus$ and $\otimes$ are used instead; this, of course, does not hold for structural induction.

In earlier versions of this paper several of the results, particularly in Section 6, were proved by means of structural induction. However, it turned out that the inductions could be eliminated in every case, and that pure initiality arguments were sufficient. One can see how induction might be replaced by initiality, in a simple case, by looking at (5). To prove a statement of this form by initiality, we would try to show that each of f,g is an arrow from A to B, where (for suitable F) A and B are F-algebras, with A initial.

The elimination of induction is not always as straightforward as this, even if it can always be achieved (something about which we are not yet in a position to make any general claims). In the discussion of implementation by isomorphism in Section 6, the effort to remove induction led to a substantial improvement in the results and their proof. A comparatively simple example of the replacement of induction by initiality appears in the discussion of the third topic of this section (below).

Finally, we remark that some uses of ordinary mathematical induction remain in Section 6. These are "harmless", since the predicates involved are in each case continuous (recall from Section 2 that equality is continuous in the case of N), so that a direct reduction to initiality is feasible.

C. Correctness of data type definitions. In the discussion of particular data types in Section 3, no attempt was made to show that our proposed definitions were "correct". But the definitions were supposed to be precise explications of the informal notions current in computer science, and it seems reasonable to demand some demonstration that our data types have the properties usually required.

In trying to make sense of this somewhat vague demand, we may borrow an idea from ADJ. We can try to write down a list of the "usual" properties of the data type in question, and then show that the type we have defined is initial in the class of algebras satisfying those properties; this would surely be convincing evidence.

It happens that this can be done fairly easily for most of data types introduced in Section 3. We will discuss the (not so easy) example of queues in some detail. The example is interesting because the type in question is composite, and because it does not seem to have an accepted algebraic definition.

We do not know of any published list of properties intended to characterize the standard operations on queues. The following list was suggested by D.Park (private communication):

$$\begin{aligned} &\mathrm{pop}(\mathrm{push}(a,s)) = \underline{\mathrm{if}}\ \mathrm{empty}(s)\ \underline{\mathrm{then}}\ \Lambda\ \underline{\mathrm{else}}\ \mathrm{push}(a,\mathrm{pop}(s)) \\ (6)\qquad &\mathrm{top}(\mathrm{push}(a,s)) = \underline{\mathrm{if}}\ \mathrm{empty}(s)\ \underline{\mathrm{then}}\ a\ \underline{\mathrm{else}}\ \mathrm{top}(s) \\ &\mathrm{empty}(\Lambda) = tt \qquad \mathrm{empty}(\mathrm{push}(a,s)) = ff \end{aligned}$$

Here, top and pop should probably be regarded as partial operations. To avoid this difficulty - and also the introduction of terminology relating to many-sorted algebras, which would be needed for the accurate handling of (6) - we can use the approach suggested in Section 3. Top and pop are amalgamated into a map $q:Q \to TQ$ (= $2 + A \times Q$), where

$$q(s) = \begin{cases} \top & \text{if } s = \Lambda \\ \langle top(s), pop(s) \rangle & \text{else} \end{cases}$$

Similarly, push, empty and Λ are replaced by $u: TQ \to Q$, where

$$u(\top) = \Lambda$$

$$u(\langle a,s \rangle) = push(a,s).$$

Park's equations may now be written as follows:

$$(7) \quad \begin{aligned} q \circ u(\top) &= \top \\ q \circ u(\langle a,s\rangle) &= \underline{\text{if}}\ q(s)=\top\ \underline{\text{then}}\ \langle a,u(\top)\rangle\ \underline{\text{else}}\ \langle p_1 \circ q(s), r(a, p_2 \circ q(s))\rangle \end{aligned}$$

(These equations may be made more readable by writing empty(s) instead of $q(s)=\top$, and Λ instead of $u(\top)$.) It will be convenient to have (7) in diagrammatic form. Indeed, (7) is equivalent to the commuting of

$$(8) \quad \begin{array}{ccc} Q & \xleftarrow{\ u\ } & TQ \\ {\scriptstyle q}\downarrow & & \downarrow{\scriptstyle Tq} \\ TQ & \xleftarrow{\ d_u\ } & TTQ \end{array} \quad = 2 + A \times (2 + A \times Q)$$

where d_u is defined by:

$$d_u(\top) = \top$$

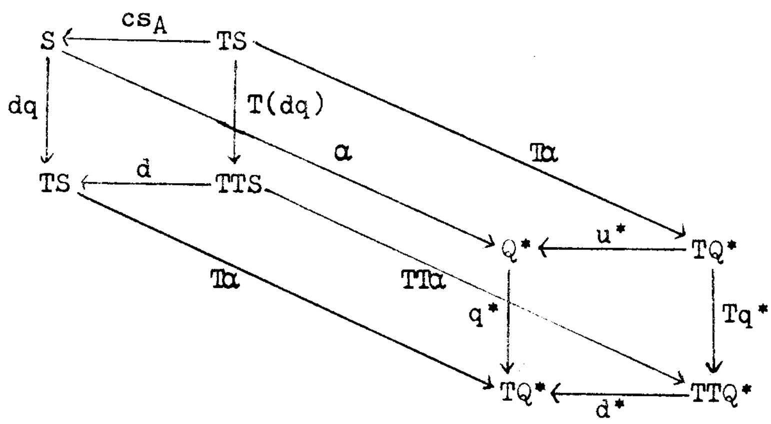
$$d_u(\langle a,\top\rangle) = a$$

$$d_u(\langle a,\langle b,s\rangle\rangle) = \langle b, u(\langle a,s\rangle)\rangle.$$

In the case that Q is StackA and u is cs_A, this just says that q is the operation dq of the data type QueueA (StackA $\xleftarrow{cs_A}$ TStackA $\xleftarrow{dq}$ StackA; see Sec.3), as is easily verified. Thus QueueA certainly satiafies (8). Let $D^* = Q^* \underset{q^*}{\overset{u^*}{\rightleftarrows}} TQ^*$ be any algebra satisfying (8). We want to show that there is a unique arrow from $Queue_A$ to D^*. This means that we must show that the arrow α from the T-algebra cs_A to u^* (given by initiality of cs_A) also satisfies

$$(9) \qquad T\alpha \circ dq = q^* \circ \alpha$$

Consider the following diagram, in which S, d, d^* abbreviate $\mathrm{Stack}A, d_{cs_A}, d_{u^*}$:



The two end squares and the top rectangle commute by definition. One can check that the bottom rectangle commutes; the main case is:

$$\begin{aligned} T\alpha\ d(\langle a,\langle b,s\rangle\rangle) &= \langle b, \alpha\ cs_A(\langle a,s\rangle)\rangle \\ &= \langle b,\ u^*(\langle a,\ \alpha(s)\rangle)\rangle \\ &= d^*(\langle a,\ \langle b,\ \alpha(s)\rangle\rangle) \\ &= d^*\ TT\alpha(\langle a,\ \langle b,s\rangle\rangle). \end{aligned}$$

Since the back square and the bottom rectangle commute, $T\alpha\ dq$ is an arrow from cs_A to d^*. Since the front square and top rectangle commute, $q^*\alpha$ is an arrow from cs_A to d^*. By initiality of cs_A these arrows are equal; the result is proved.

6. Implementation of data types.

Sometimes one wants to be able to prove that certain operations on data type D' enable him to simulate, represent or implement those of a data type D. Typically, D will have been defined "abstractly", say by a circular definition, while D'is more "concrete" (more like what is usually available in programming languages). The representation of stacks and queues by arrays may be cited as examples. An interesting example that does not quite fit this pattern is that of the representation of forests by binary trees. Here the representation involves, as it were, a reduction of "star-height": in the definition of the data type Forests there occurs already a parameterized data type, while Btrees is defined via a purely polynomial functor (see Sec.3, or end of Sec.4).

These examples are well-known, and an elementary treatment may be found in Knuth(69). Our purpose is to give an exact definition of "implementation" which will cover these and other examples, and to exhibit some techniques for proving that proposed implementations are correct. As a preliminary, let us indicate why we are not satisfied with the equational approach of ADJ. Consider the implementation of stacks by arrays. Suppose the operations on arrays which correspond to pop,push are pop',push'. Then it is not true that we have:

$$pop'(push'(a,H)) = H$$

In fact, the "useful" parts of H and pop'(push'(a,H)) will be the same, but the contents of the first free location will, in general, have been changed (by the sequence of operations: push'(a,-),pop'). This situation is quite typical of representations which are not one-one.

The idea behind our definition is very simple. When it is said that data type D' implements data type D, this means, first, that D' and D are of the same type (the operations of D' and D are in one-one correspondence); secondly, that the elements of D' are taken as representing elements of D, and this consistently with the operations of D,D'. Thus:

Definition 1. An *implementation* of D by D', where D and D' are data types, is a homomorphism (of data types, see Sec. 3) $r:D'\to D$.

Remark. As already noted, r need not be one-one. Moreover, r need not be onto: the most one can say (in general) is that every element of D that is definable by means of the operations of D has a representative in D'. Examples would be provided by implementations of the reals, or of procedural data types.

In all the examples to be discussed here, the map r is onto; this is a consequence of

Lemma 1. If D is a data type with carrier A which contains as one of its operations an initial T-algebra $\varphi:TA\to A$, then any implementation $r:D'\to D$ is onto.

Proof. Consider

$$\begin{array}{ccc} A & \xleftarrow{\varphi} & TA \\ \alpha\downarrow & & \downarrow T\alpha \\ A' & \xleftarrow{\varphi'} & TA' \\ r\downarrow & & \downarrow Tr \\ A & \xleftarrow{\varphi} & TA \end{array}$$

By initiality, $r\cdot\alpha = I_A$; hence r is onto.

We continue with a detailed treatment of the examples mentioned in the opening paragraph of this section.

Example 1. Representation of stacks and queues by arrays.

(i) Stacks. Let D be $StackA \xleftarrow{cs_A} T(StackA) \xleftarrow{cs_A^{-1}} StackA$, where T is $2 + A \times -$. Let Z be the set of pairs $\langle H,n\rangle$, where H is an infinite array of A and n is a natural number, with the obvious ordering; more precisely, Z is the coalesced sum of countably many copies of ArrA. Let D' be

$Z \xleftarrow{p} TZ \xleftarrow{q} Z$, where

$$p(\top) = \langle H_0, 0\rangle \qquad (H_0 \text{ an arbitrarily chosen array})$$

$$p(a,\langle H,n\rangle) = \langle upd(H,n,a),n+1\rangle$$

$$q(H,n) = \underline{\text{if}}\ n=0\ \underline{\text{then}}\ \top\ \underline{\text{else}}\ \langle acc(H,n-1),\langle H,n-1\rangle\rangle.$$

Define $r:Z \to StackA$ recursively by:

$$r(H,m) = \underline{\text{if}}\ m=0\ \underline{\text{then}}\ \Lambda\ \underline{\text{else}}\ push(acc(H,m-1),r(H,m-1)).$$

We shall need an elementary property of r which we state as

<u>Lemma 2</u>. For all H,a,m,k, we have:

(1) $$r(upd(H,m+k,a),m) = r(H,m)$$

<u>Proof</u>. By induction on m. The basis m=0 is trivial. Suppose that (1) holds for all H,a,k, with m=n. Then

$$\begin{aligned} r(upd(H,n+1+k,a),n+1) &= push(acc(upd(H,n+k+1,a),n),r(upd(H,n+k+1,a),n)) \\ &= push(acc(H,n),r(H,n) \\ &= r(H,n+1). \end{aligned}$$

We have to show that $r\circ p = cs_A \circ T(r)$. But $r\circ p(\top) = cs_A \circ T(\top)$ trivially, and

$$\begin{aligned} r\circ p(a,\langle H,n\rangle) &= r(upd(H,n,a),n+1) \\ &= push(acc(upd(H,n,a),n),r(upd(H,n,a),n)) \\ &= push(a,r(H,n)) \quad \text{using Lemma 2} \\ &= cs_A \circ T(r)(a,\langle H,n\rangle). \end{aligned}$$

An equally easy verification shows that $T(r)\circ q = cs_A^{-1} \circ r$; thus D' is an implementation of D. Note that in this implementation the functions p,q are not inverse to each other.

(ii) Queues. We represent a queue by an array H together with two indices m,n, such that the items of the queue are (from head to tail) H(m+n-1),...,H(m). Formally, D is $\text{StackA} \xleftarrow{cs_A} T(\text{StackA}) \xleftarrow{dq} \text{StackA}$, Z is the coalesced sum of a double sequence of copies of ArrA, and D' is $Z \xleftarrow{p} TZ \xleftarrow{q} Z$, where

$$p(\top) = \langle H_0,0,0\rangle \qquad (H_0 \text{ arbitrary})$$

$$p(a,\langle H,m,n\rangle) = \langle \text{upd}(H,m+n,a),m,n+1\rangle$$

$$q(H,m,n) = \text{if } n=0 \text{ then } \top \text{ else } \langle \text{acc}(H,m),\langle H,m+1,n-1\rangle\rangle$$

Define $r:Z\to\text{StackA}$ by

$$r(H,m,n) = \underline{\text{if}}\ n=0\ \underline{\text{then}}\ \Lambda\ \underline{\text{else}}\ \text{push}(\text{acc}(H,m+n-1),r(H,m,n-1))$$

Corresponding to Lemma 2, we have: $r(\text{upd}(H,m+n+k,a),m,n) = r(H,m,n)$. Then $r\circ p = cs_A\circ T(r)$ is proved just as before. To complete the proof that D' implements D, we need

Lemma 3. With pop' and top' defined as in Sec.3, we have for $n \geq 1$:

$$\text{top}'\circ r(H,m,n) = \text{acc}(H,m)$$

$$\text{pop}'\circ r(H,m,n) = r(H,m+1,n-1).$$

Proof. By induction on n. For n=1 the proof is trivial in each case. For the induction step we have:

$$\begin{aligned} \text{top}'\circ r(H,m,n+1) &= \text{top}'\circ \text{push}(\text{acc}(H,m+n),r(H,m,n)) \\ &= \text{top}'\circ r(H,m,n) \\ &= \text{acc}(H,m) \\ \text{pop}'\circ r(H,m,n+1) &= \text{pop}'\circ \text{push}(\text{acc}(H,m+n),r(H,m,n)) \\ &= \text{push}(\text{acc}(H,m+n),\text{pop}'\ r(H,m,n)) \\ &= \text{push}(\text{acc}(H,m+n),r(H,m+1,n-1)) \\ &= r(H,m+1,n). \end{aligned}$$

Using this lemma, it is immediate that $T(r)\circ q = dq\circ r$.

Our second main example is the representation of forests by binary trees. As pointed out in Sec.4(Corollary 2), BtreeA $\cong$ ForA. This is not sufficient for the representation(=implementation), since it does not indicate how the operations of ForA are to be implemented. The deficiency can be repaired with the aid of the followinf easy lemma:

<u>Lemma 4</u>. Let $D = Z \xleftarrow{\varphi} FZ \xleftarrow{\varphi^{-1}} Z$ be a simple data type, $r: Z' \to Z$ an isomorphism. Put $\varphi' = r^{-1} \circ \varphi \circ Fr$. Then the data type $D' = Z' \xleftarrow{\varphi'} FZ' \xleftarrow{\varphi'^{-1}} Z'$ is isomorphic to D.

<u>Proof</u>. Obvious.

This lemma shows that there is indeed an (isomorphic) implementation of forests by binary trees (more precisely, by a data type with carrier the binary trees). However, direct application of the lemma does not yield the implementation in a very convenient form. More manageable expressions can be extracted from the proof of Theorem 2 (Sec.4). To see this, we will extend the notation of that theorem in order to handle the solution "in the other order" of the pair

$$(2) \qquad \begin{aligned} X &\cong F(X,Z) \\ Z &\cong G(X,Z) \end{aligned}$$

(Some of this notation was already introduced in Corollary 1 of Sec.4.) Namely, F_Z is $\lambda X.F(X,Z)$, X_- is $Y \circ (\mathrm{Abst}F)$, G^* is $\lambda Z.G(X_Z,Z)$, Z^* is $Y(G^*)$, and X^* is X_{Z^*}. Thus $\langle \eta_{F_{Z^*}}, \eta_{G^*} \rangle$ is initial for $\langle F,G \rangle$. Define ζ as in the proof of the theorem, with X^*,Z^* in place of X',Z' (and $\eta_{F_{Z^*}}, \eta_{G^*}$ in place of p,q):

$$(3) \qquad \begin{array}{ccc} Z_{X^*} & \xrightarrow{\;\eta_{G_{X^*}}\;} & G_{X^*}(Z_{X^*}) \\ {\scriptstyle \zeta}\downarrow & & \downarrow{\scriptstyle G(I_{X^*},\zeta)} \\ Z^* & \xrightarrow{\;\eta_{G^*}\;} & G_{X^*}(Z^*) \end{array}$$

Then diagram (7) becomes

(4)
$$\begin{array}{ccccc} \bar{X} & \xleftarrow{\quad \eta_{\bar F} \quad} & F(\bar X,\bar Z) & = & \bar F(\bar X) \\ \alpha \downarrow & & \downarrow F(\alpha, Z_{\alpha}) & & \\ X^* & \xleftarrow[\eta_{F_{Z^*}} \circ F(I_{X^*},\zeta)]{} & F(X^*, Z_{X^*}) & = & \bar F(X^*) \end{array}$$

α is, of course, an isomorphism. This fits the pattern of Lemma 4, with φ as $\eta_{\bar F}$, r as α^{-1}, and φ' as $\eta_{G^*} \circ F(I_{X^*}, \zeta)$.

<u>Example 2</u>. Representation of forests by binary trees. As in Corollary 2 (Sec.4), we take $F(X,Z)$ as Z, $G(X,Z)$ as $2 + A \times X \times Z$. We find that $\bar X$ = ForA, $\bar Z$ = Stack(A x ForA), $X^* = Z^*$ = BtreeA, Z_{X^*} = Stack(A x BtreeA), $\bar F$ is Stack(A x -), G_X is $2 + A \times X \times -$, and $\eta_{\bar F}, \eta_{G^*}, \eta_{G_{X^*}}, \eta_{F_{Z^*}}$ are cf_A, cbt_A, $cs_{A \times BtreeA}$, I_{BtreeA} respectively. (3) becomes

$$\begin{array}{ccc} \text{Stack}(A \times \text{BtreeA}) & \xleftarrow{\quad cs_{A \times \text{BtreeA}} \quad} & 2 + A \times \text{BtreeA} \times \text{Stack}(A \times \text{BtreeA}) \\ \zeta \,\vdots & & \downarrow I \qquad \downarrow I \qquad \vdots\, \zeta \\ \text{BtreeA} & \xleftarrow{\quad cbt_A \quad} & 2 + A \times \text{BtreeA} \times \text{BtreeA} \end{array}$$

Thus the recursive definition of ζ reads :

$\zeta(s)$ = if empty(s) then NIL else $cbt_A(p_1 \circ hd(s), p_2 \circ hd(s), \zeta \circ tl(s))$,

where NIL is $cbt_A(\top)$. Note that $\eta_{F_{Z^*}} \circ F(I_{X^*}, \zeta) = \zeta$. From (4) we have

$$\begin{array}{ccc} \text{ForA} & \xleftarrow{\quad cf_A \quad} & \text{Stack}(A \times \text{ForA}) \\ r \uparrow & & \uparrow \text{Stack}(I_A \times r) \\ \text{BtreeA} & \xleftarrow{\quad \zeta \quad} & \text{Stack}(A \times \text{BtreeA}) \end{array}$$

in which every arrow is an isomorphism. This yields the recursive definition of the representation function r:

$$r(x) \;=\; \underline{\text{if}}\ \ \text{empty}(x)\ \ \underline{\text{then}}\ \ \Lambda\ \ \underline{\text{else}}\ \ cf^*(\text{root}(x), r\circ lt(x), r\circ rt(x))$$

where $cf^*: A \times ForA \times ForA \to ForA$ is defined by:

$$cf^*(a, f_1, f_2) \;=\; cf_A \circ cs_{A \times ForA}(\langle\langle a, f_1\rangle, cf_A^{-1}(f_2)\rangle).$$

We conclude this section with a remark about the appropriateness of Definition 1. Suppose that we have a program π written in a high-level language using "abstract data types", and that we can prove that π computes a function $f: X \to Z$. Suppose, further, that we have an implementation of the language (a translation into a lower-level language) and, in particular, of the data types used in π. What does it mean to say that the translated program, π', is still correct with respect to f? The answer must, it seems, be as follows. Let π' compute $f': X' \to Z'$, where X', Z' are the implementations of X,Z respectively. Then for any $x' \in X'$, if x' represents $x \in X$, $f'(x')$ represents $f(x)$. This just says that the diagram

$$\begin{array}{ccc} X & \xrightarrow{\;f\;} & Z \\ {\scriptstyle r_X}\uparrow & & \uparrow{\scriptstyle r_Z} \\ X' & \xrightarrow{\;f'\;} & Z' \end{array}$$

commutes. It is at least plausible (we do not have a precise result as yet) that, to ensure that this result holds for every π, f, we will need just the condition laid down in Definition 1.

7. Advice to language designers.

We think there are some practical conclusions to be drawn from our investigation. The fact that the definition of data types has such a simple and clear mathematical semantics does speak in favor of a mathematical definition of programming languages, rather than an operational definition. The main advantage of a mathematical definition is clarity: it is much easier for the user to think in terms of abstract data types, for example initial fixpoints of functors, than in terms of their representations - as he is forced to do if the definition of the language has an operational flavor and stresses representations, in the manner of the ALGOL 68 Report (van Wijngaarden et al(75)).

Quite often a language is difficult to learn not because of what is in the language but because of what is left out; the restrictions, when they are not solidly justified on semantic grounds, are most difficult to memorize. Our second piece of advice would then be to remove all restrictions on the use of circular definitions for specifying data types; again - as with nearly all the points to be made in this section - ALGOL 68 provides an illustration of what needs to be improved upon. Apart from easing the learning of the language, the introduction of arbitrary circular type definitions would allow the programmer to define data types such as lists and trees without having to introduce references (or pointers) explicitly. This would greatly facilitate proofs of correctness. Indeed, proofs of correctness are intractable when sharing of values (sometimes called aliasing) occurs. One can try to exclude aliasing by means of syntactic restrictions on programs; but this does not seem to be possible when references are permitted.

The greatest care should be taken in the design of a programming language to provide the necessary facilities to denote useful basic data types, such as 1 and 2, and useful type constructors, such as sums and products. In particular, the distinctions between x and $\otimes$, + and $\oplus$, should be emphasized. Union should be clearly recognized as being a discriminated union.

The facility for circular definitions of data types should provide a canonical way of defining the functions defined. More precisely, a circular definition $X \cong TX$ should enable the user to give a name to $\eta_T^{-1}:X \quad TX$ and to $\eta_T:TX \quad X$ the initial T-algebra. η_T^{-1} is obviously necessary, but one could question the necessity of η_T and propose to make it an implicit "coercion".

Our last piece of advice is that a language should allow the user to specify abstract data types on one hand and the way he intends his abstract data types to be implemented (by way of a homomorphism of data types) on the other hand. The responsibility for checking that the proposed implementation is indeed a homomorphism could be either left to the user or the compiler could be asked to check a proof of that fact given by the user.

8. Conclusion.

The category-theoretic method enables us to present the semantics of data types as a precise generalization of the usual partial order semantics. This is of great value heuristically in formulating the basic definitions and results of the theory. More specifically, it helps explain the fundamental role of initiality, by showing that this is just (the generalization of) the least fixpoint property.

While we are sure of the solidity of the mathematical foundations, large gaps remain in our treatment of the applications. Implementations of data types need to be investigated in far greater variety than we have attempted in Section 6. And it will no doubt be pointed out that specific design proposals are needed, not just general advice to language designers.

Acknowledgements

Many people, not all of them mentioned in the text of this paper, have played an important role in the genesis of the ideas and techniques used in this paper. We benefited from direct conversations with the theory group at Warwick University and especially D. Park and W. Wadge, and also with D. Scott, G. Plotkin and J. Reynolds. During the elaboration of this work both authors were supported by the Science Research Council grant B/RG 31948 to D. Park and M. Paterson.

References

Adamek, J.

-(1974) Free algebras and automata realizations in the language of categories. Commentationes mathematicae universitatis carolinae. pp. 589-602.

ADJ: Goguen J.A., Thatcher J.W., Wagner E.G., Wright J.B.

-(1975) Abstract data types as initial algebras and the correctness of data representations. Proc. Conference on Computer Graphics, Pattern Recognition and Data Structures.

-(1977) Initial algebra semantics and continuous algebras. Journal of the ACM Vol.24,No.1 pp.68-95.

Arbib, M.A. and Manes, E.G.

-(1974) A categorist's view of automata and systems. Category theory applied to computation and control, Springer Lecture Notes in Computer Science, Vol.25.

de Bakker, J.

-(1974) Recursive procedures. Math. Centrum, Amsterdam.

Burge, W.H.

-(1975) Recursive programming techniques. Addison Wesley.

Courcelle, B. and Nivat, M.

-(1978) Algebraic families of interpretations.

Friedman, D.P. and Wise, D.S.

-(1976) CONS should not evaluate its arguments. Proc. of 3rd Coll. on Automata, Languages and Programming. Edinburgh.

Gordon, M.

-Ph. D. thesis. Edinburgh University.

Henderson, P. and Morris, J. Jr.

-(1976) A lazy evaluator. 3rd ACM Symposium on Principles of Programming Languages. pp.95-103.

Herrlich, H. and Strecker, G.E.

-(1973) Category theory. Allyn and Bacon.

Knuth, D.

-(1969) The art of computer programming. Vol.1. Addison Wesley

Lawvere, F.W.

-(1964) An elementary theory of the category of sets. Proc. National Academy of Sciences Vol.52 pp.1506-1510.

Lehmann, D.J.

-(1976) Categories for fixpoint semantics. Theory of Computation Report no. 15, Department of Computer Science, University of Warwick. See also Proc. 17th Annual Symposium on Foundations of Computer Science. I.E.E.E.

-(1977) Modes in ALGOL Y. Theory of Computation Report no. 17, Department of Computer Science, University of Warwick. See also Proc. 5th Annual I.I.I. Conference.

MacLane, S.

-(1971) Categories for the working mathematician. Springer.

MacLane, S. and Birkhoff, G.

-(1967) Algebra. Macmillan.

Markowsky, G.

-(1974) Categories of chain-complete posets. IBM research Techn. Report RC5100. To appear in Theoretical Computer Science.

Plotkin, G.D.

-(1976) A powerdomain construction. SIAM Journal on Computing Vol.5 No.3 pp.452-487.

Scott, D.S.

-(1971) The lattice of flow diagrams. Semantics of Algorithmic Languages (E. Engeler, ed.) Springer Lecture Notes in Mathematics, Vol.188 pp.311-368.

-(1972) Continuous lattices. Toposes, Algebraic Geometry and Logic (F.W. Lawvere, ed.) Springer Lecture Notes in Mathematics Vol.274 pp.97-136.

-(1976) Data types as lattices. SIAM Journal on Computing Vol.5 No.3 pp.522-587.

Smyth, M.B.

-(1976a) Effectively given domains. Theory of Computation Report no.9 Dep. of Computer Science, Univ. of Warwick. To appear in Theoretical Computer Science.

-(1976b) Powerdomains. Theory of Computation Report no.12, Dep. of Computer Science, Univ. of Warwick. See also Mathematical Foundations of Computer Science, Springer Lecture Notes in Computer Science No.45 pp.537-543.

Wand, M.

-(1974) On the recursive specification of data types. Category theory applied to Computation and Control, Springer Lecture Notes in Computer Science Vol.25.

van Wijngaarden, A.; Mailloux, B.J.; Peck, J.E.L.; Koster, C.H.A.; Sintzoff, M.; Lindsey, C.H.; Meertens, L.G.L.T. and Fisker, R.G.

-(1975) Revised report on the Algorithmic Language Algol 68. Acta Informatica 5 pp.1-236.