Algebraic and Relational Approach to Abstract Data Types and their Representations

Pierre Lescanne

1979

Foreword for the translation

This document is the translation of parts of my *thèse d'État* (currently the first two chapters) which has been defended at the Institut National Polytechnique de Lorraine in Nancy on the 11^{th} September 1979. Except in the thesis document, those chapters have never been published and are interesting because they prefigure the theory of *Evolving Algebra* (later called *Abstract State Machines* or *ASM*). Despite it could be inconvenient for an English reader, I decided to keep in French most of the original key words, except TRUE (VRAI), FALSE (FAUX) and **if ... then ... else...** (**si ... alors ... sinon ...**). Moreover I staid closed to the French original, which might lead to awkward constructions. At some places, the text may look naive when we read it in 2019, but I decided not to change it and to keep the 1979 wording.

The footnotes I wrote during the translation have the prefix *Note at the translation*, the others are footnotes of the original document.

In forty years, much have changed.

- *From the theoretical point of view:* in 1979, the only advanced language with recursive types known in France was Algol 68 (with no implementation by the way). Obviously object oriented programming did not exist. When I implemented algorithms (especially in Chapter V and Chapter VI) I did it in Pascal.

- *From the technical point of view:* in 1979, scientific texts were first handwritten and then typewritten by a secretary (in this case Martine Tesolin, whom I thank again for her beautiful work). Afterwards specific signs (like $\mathfrak{A}$, $\mathfrak{B}$, $\mathfrak{C}$) were handwritten on the typed document by the author, such that further corrections were not possible. Thus in the French text, there are still many remaining mistakes and typos. Trivial errors were corrected at translation, others still remain, for faithfulness.

I would like to thank Claude Pair for everything and especially for giving me access to his personal archives on information systems, Laurent Rollet or organising a workshop on the *History and the Memory of Academic Computer Science in Nancy* and Yuri Gurevich for interesting discussions.

December 2018
Pierre Lescanne

Introduction

1 The problem of programming

The programming activity can be decomposed into two major steps:

1. The first step goes from a problem statement to an algorithm.
2. The second step goes from an *algorithm on abstract objects* to a *program on data* in order to produce an execution on a machine.

In this thesis we speak only of the second step. In the sense given here an *algorithm* is a description of the operations on the inputs of a program to produce the outputs or the results.

The algorithm deals with abstract objects expressed in a mathematical language. If good tools are available the algorithm and the description of the abstract objects do not make reference to an execution or to a computer run. For this reason we can call this description static.

For instance, to express algorithms, we can use applicative languages describing recursive functions or single assignment languages defining sequences [AW76, Bel78]. For abstract objects a similar description is possible. For instance, an algebraic specification allows u to describe abstract data types only by relations on operations.

On the opposite, a *program* is a description of *pieces of computations*, that can be run on a computer. Therefore the data that the program handles shall be run by the computer. The actions which are components of the computation are modifications of the memory state.

If several methods have been proposed for going for an algorithm to a program [Ars78, BD77, Pai79, BW79], going from an abstract data type to a *data structure* was less solved by now. There are three categories of problems:

- the problem of the representations,
- the problem of the assignment,
- the problem of data sharing.

The *problem of representation* can be pictured as follows: an object of an abstract data type is a black box with buttons: in order to get an external value,

we push the adequate button. Some operations combines boxes. The user does not want to known what is inside boxes, but only to know how boxes react to external requests.[1] To represent an abstract data type is to give a possible description of the internal structure of the boxes. We may imagine that building this representation is done more or less automatically.

The problem of assignment occurs as soon as we want to run a sequence of computations and make values to vary. The use of a an abstract data type in a algorithm requires all the objects of the types are available at each time, at leas potentially. This is possible in particular cases. For instance in a computer, natural numbers can be simply denoted and are accessible at each time. However in built-in types like stacks, this is different. At a given time, only a small number of objects are available via identifiers and therefore are present in the store. This identifiers can also share objects. For instance, an identifier can be associated with a sequence of *push*'s and *pop*'s. In a program, at a given time, it refers to the last stack. Therefore it is needless to access all the stacks. Then we raise the following question: how to represent stacks such that *push* is as simple as possible? Since only the last stack is of interest, it can be obtained as a modification of the last stack. Another question is: how to minimise these modifications? The answer to this question leads to an in situ modification of the objects. In other words, a minimum number of modifications are performed on the object without moving it. In short, we must minimise time duplications of objects.

The *problem of data sharing* occurs whenever we want to use at the best the memory space, doing so that different objects might access commons sub-objects, which do not need to be duplicated. In short, we must minimise space duplications of objects.

Addressing these tree problems assumes that we know well the representation of the objects and that we own a good mathematical modelling. Thus, reasoning will be rigorous and a mechanisation can be addressed. It seems that much comes from a good approach of the concept of representation. Therefore, I will address essentially representation, hopping that I will be able to draw consequences for the two other problems.

In this thesis, two approaches are proposed:

- an algebraic approach formalises the concept of a set with operations,
- an relational approach axiomatises the relations with the aim of describing the objects.

A third possible approach is based on the first order predicate calculus. You can find it in Remy's PhD [Ré74] or in the papers of Pair [Pai74] or of Finance [Fin77].

[1] *Note at translation:* This looks like object oriented programming, but this was stated before that paradigm existed.

2 Programming vs representing objects

Representing abstract data types by objects available in a programming language requires an intermediate step, namely a representation in terms of mathematical objects. Those objects are later described by computer tools. Eventually, these three entities are the three edges of the triangle drawn on Figure 1.

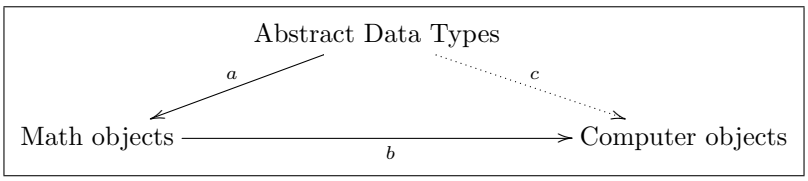


Figure 1: The steps toward the representation

When the function described by the dotted arrow c has been found, the problem of the representation has been solved. Since, we assume that we know b, the central problem is a.

The mathematical objects, called "representations", can be seen another way: they are the first objects which the programmer takes in consideration and which he (she) addresses. Once using these objects, he (she) draws relations among operations, he (she) deduces the abstract data types with which he (she) is going to describe the algorithm. Possibly he (she) adds new operations to the abstract data type. Possibly as well, in the use of the abstract data type in algorithms, he (she) notices that a specific operation occurs more than the others and requires to be implemented more efficiently. This leads the programmer to look carefully at the mathematical objects on which he (she) reasons. This may involve another family of representations, which improves the implementations. See for instance [FVV78].

3 The aims of this thesis

The results of this thesis are essentially the representations of abstract data types by algebraic models. Indeed, we build a model of an abstract data type, aka type algebra (the *mother algebra*) whose objects are also algebras, called daughters and whose operations are operations on those *daughter algebras*. After describing a family of daughter algebras representing the abstract data type "binary tree", the theorem of the algebraic representation of binary trees claims that these daughter algebras are actually a model of the type "binary tree". The theorem of the canonical representation claims that if the abstract data type is "correctly specified", such a model exists always. The proof is an explicit construction of this model. In general, the models are not isomorphic. Actually, several representations are proposed for sets and for graphs.The proposition of complete discrimination characterises the abstract data types whose models are isomorphic.

On another hand, in this thesis, the problem of the non determinism is addressed at two levels:

- *At the level of the operations of the abstract data type*, the theorems of $\mathbb{C}$-stability and of $\mathbb{P}$-stability give conditions on equations in order to make the abstract data type consistent when non deterministic operations are accepted.
- *At the level of the representations*, the theorem of the relational representation of lists is symmetric to the theorem of the algebraic representation of binary trees, when one admits operations of daughter algebras to be relations.

Eventually, addressed for relational algebras, the problem of formal proofs in algebraic systems is considered and experimental results are proposed.

4 Plan of the thesis

This thesis has two parts:

- the *algebras*, chapters I, II,, III, IV and
- the *relations*, chapters V and VI.

Chapter I

An algebraic framework for studying abstract data types

1 Introduction

The aim of this work is double:

- to *set nicely*, around the concept of algebra, several approaches of data structures,
- to propose an *algebraic framework for theories of the representation of Abstract Data Type* in a programming language with assignments, hence with a semantics involving state changes.

We can set the formalism for data structures into two large families [Gut78].

- In a first kind of approach, an object of some structure can possibly evolve in time and be modified; but in order to stay in the structure, it must satisfy along time, either an invariant predicate (this is the approach of Hoare [Hoa72], Wulf [WLS76]), or a family of axioms and its consequences (this is the approach of Pair [Pai74], Remy [Ré74] and Finance [Fin77] see also [GP77]).

  A *construction* of an object from other objects is described by a program in Hoare's approach and by adding modifications to axioms in the Nancy approach. For reasons of harmony and for more mathematical simplicity, which we will justify later on, we propose here to describe an object as an algebra; we speak on "daughter algebras" in opposition to "mother algebras" or *type algebras* which are the mathematical foundation of the description with an abstract data type. The invariance or the membership

to the structure is then translated by the requirement for the daughter algebras to stay in a *class of algebras*. This class of algebras is characterised by properties of first or second order. In particular, this class is not an equational class aka a variety. We build a new daughter algebra by expressing its operations from those of one algebra or of many other algebras. If we do not wish to build a new algebra, but to provide a fixed element of an algebra, we speak of a *selection*. The operation in a daughter algebra which is used to "retrieve" an element in the algebra from other elements is called an *access*. Speaking in term of algebras allows us to characterise them nicely.

In Chapter 5, another approach which we still keep in the same family, is proposed. It is based on Tarski theory of relational algebras and uses ideas proposed by de Bakker, Hitchcock, Park and de Roever ([dBdR72], [dBdR72], [dR74], see also [Liv78]). It recalls Finance, Pair and Remy approach and takes advantage of the algebraic tool to promote a strict rigour, hoping a possible mechanisation (Chapter 6).

- In a second family of approaches, an algebraic description is used and is based on specific operations ruled by axioms on those operations, thus characterising an *abstract data type*, which is the set of objects we are interested in and the set of transformations that are assigned to them [Gut77, GTWW75, LZ77]. This approach is more abstract, since in one sight we look at all the objects and all their constructions.

2 Binary trees

2.1 Algebras "Binary Trees labelled by **Alph**"

We sees immediately that two *sorts* of objects appear, the trees and the values. Let us denote the first Arb and the second Val. Here $\mathsf{Val} = \mathsf{Alph} \cup \mathsf{INDEF}$ where $\mathsf{INDEF} \notin \mathsf{Alph}$. Further, we define operations which allow us to extract sub-trees and to construct new trees. Each operation has a signature, that is a description of its operands and of its result. Thus CONS which builds a binary tree, from two trees and from a value, has signature:

$$(\mathsf{Arb}, \mathsf{Val}, \mathsf{Arb}) \to \mathsf{Arb}$$

which means that its first operand is a tree, that its second operand is a value, that its third operand is a a tree and that its result is a tree. We keep this notation in what follows.

Similarly, the operations GAU, DRO[1] have signature

$$(\mathsf{Arb}) \to \mathsf{Arb}$$

[1] *Note at translation:* Arb stands for *arbre* which means *tree*, GAU stands for *gauche* which means *left*, DRO stands stands for *droite* which means *right*, TET stands for *tête* which means *head*, VID stands for *vide* which means *empty*.

and associate, with a tree x, another tree called the left-hand side or the right-hand side of the tree x. TET has signature:

$$(\mathsf{Arb}) \to \mathsf{Val}$$

and associates, with a tree, a value, namely the label at the head of the tree. Thus, we build an algebraic structure which we call a *heterogeneous algebra* or a *many sorted algebra*. Such an algebra is characterised by a family of sets, namely the sorts[2] and a family of operators of a given signature.

The operations CONS, GAU, DRO and TET satisfy identities which characterise the binary trees and are well known by programmers:

$$\mathsf{GAU}(\mathsf{CONS}(x, v, y)) = x \tag{I.1}$$
$$\mathsf{DRO}(\mathsf{CONS}(x, v, y)) = y \tag{I.2}$$
$$\mathsf{TET}(\mathsf{CONS}(x, v, y)) = v \tag{I.3}$$

These three identities give the properties of the operations on binary trees but do not tell what a binary tree is. Here is a possible answer: a binary tree is an object which can be described by a constant expression. But to speak of a constant expression, we need at least one constant. The simplest one is the one which describes the empty tree, written VID. Moreover it allows by composition with CONS to describe all the binary trees. With no parameter and with a result tree, this is a 0-ary operation. We write its signature:

$() \to \mathsf{Arb}$

Similarly we introduce $\nu + 1$ operations (where ν is the cardinal of Alph) of signature:

$() \to \mathsf{Arb}$

which correspond to the elements of Alph or to an undefined element written INDEF. We get the equation

$$\mathsf{TET}(\mathsf{VID}) = \mathsf{INDEF} \tag{I.4}$$

Every tree can be represented using the operations CONS, GAU, DRO, TET, VID and using the elements of the family Alph $\cup$ {INDEF}. It is interesting to study the algebras that contain only those elements, in other words the algebras that satisfy the equations I.2, I.3, I.3 and I.4 and that contain only the elements built using CONS, GAU, DRO, TET, VID and using the elements of the family Alph $\cup$ {INDEF} . Such an algebra is called a *primary algebra*[3]. Notice that such an algebra may contain infinitely many elements of the form GAU(VID), DRO(VID), GAU(GAU(VID)), GAU(DRO(VID)) etc. In computer science, this makes no sense to consider those elements as different. Two solutions are possible, to create a tree ERR: () $\to$ Arb, which represents an errored tree, resulting from an error in applying GAU or DRO. Then, we state

[2]Some authors especially the logicians, speak, in this case, of types; this word has several meanings for the computer scientists, rarely this of sets of objects [Mor73], this is why we prefer the word "sort".

[3]This naming comes from the fact that $\mathbb{Z}/p\mathbb{Z}$ is a unitary ring without proper sub-ring, if and only if p is a prime number.

$\mathsf{GAU}(\mathsf{VID}) = \mathsf{GAU}(\mathsf{ERR}) = \mathsf{GAU}(\mathsf{ERR}) = \mathsf{DRO}(\mathsf{VID}) = \mathsf{DRO}(\mathsf{ERR}) = \mathsf{ERR}$
together with the axioms which state that the result of an operation on a tuple containing on ERR is ERR. Another solution which avoids introducing ERR, but does not distinguish trees, yielded by errored operations, is to state:

$$\mathsf{GAU}(\mathsf{VID}) = \mathsf{VID} = \mathsf{DRO}(\mathsf{VID}). \tag{I.5}$$

For simplicity, we adopt the second solution.

Notations If $\mathfrak{A}$ is an algebra, then $\mathsf{Arb}_{\mathfrak{A}}$ and $\mathsf{Val}_{\mathfrak{A}}$ denote the sets of sort Arb and Val in the algebra $\mathfrak{A}$.

Then we notice that:

Proposition 1. *Let $\mathfrak{A}$ be a primary algebra,*

$$x \in \mathsf{Arb}_{\mathfrak{A}} \wedge x \neq \mathsf{VID} \Rightarrow (\exists y \in \mathsf{Arb}_{\mathfrak{A}})(\exists z \in \mathsf{Arb}_{\mathfrak{A}})(\exists v \in \mathsf{Val}_{\mathfrak{A}}) \quad x = \mathsf{CONS}(y, v, z)$$

Proof. By induction on the possible representations of x, since all the elements of $\mathsf{Arb}_{\mathfrak{A}}$ admit a representation on basic elements.

First case. $x = \mathsf{GAU}(x')$, if $x' = \mathsf{VID}$ then $x = \mathsf{VID}$ and the result is straightforward, else, by induction, $x = \mathsf{CONS}(y', v', z')$, since clearly x admits a simpler representation on basic elements, hence $x = \mathsf{GAU}(\mathsf{CONS}(y, v', z')) = y'$ where y' has a simpler representation than x, hence still by induction, $y' = \mathsf{CONS}(y'', v'', z'') = x$.

Second case. $x = \mathsf{DRO}(x')$ works like the previous case.

Third case. $x = \mathsf{CONS}(y, v, z)$ involves no proof. □

Proposition 2. $x \neq \mathsf{VID} \Rightarrow x = \mathsf{CONS}(\mathsf{GAU}(x), \mathsf{TET}(x), \mathsf{DRO}(x))$.

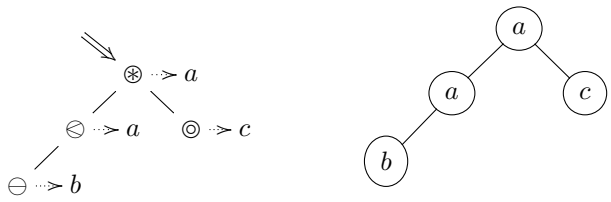
Proof. From the previous result, $x = \mathsf{CONS}(y, v, z)$. Hence

$$\begin{aligned} \mathsf{CONS}(\mathsf{GAU}(x), \mathsf{TET}(x), \mathsf{DRO}(x)) &= \mathsf{CONS}(y, \mathsf{TET}(x), \mathsf{DRO}(x)) \\ &= \mathsf{CONS}(y, v, \mathsf{DRO}(x)) \\ &= \mathsf{CONS}(y, v, z) \\ &= x \end{aligned}$$

by applying the equations I.2, I.3 and I.3. □

We would show easily that all the algebras which satisfy I.2, I.3, I.3, I.4 and I.5[4] are isomorphic, in other words each of them defines the binary trees. One of them is the class of terms without variables modulo the equations.

[4] *Note at translation:* They have to be primary and to satisfy no identity which is not a consequence!



$\mathsf{CONS}(\mathsf{CONS}(\mathsf{CONS}(\mathsf{VID}, b, \mathsf{VID}), a, \mathsf{VID}), a, \mathsf{CONS}(\mathsf{VID}, a, \mathsf{VID}))$

Figure I.1:

2.2 Algebras "One binary tree"

If we looks at a tree, we notice that there are nodes and that, from each of these nodes, we can reach another node, either on the left, or on the right or we can get a value. Therefore we can consider a binary tree as a finite algebra with a set Noe of nodes, a set Val of values (which we suppose equal to Noe $\cup$ {INDEF}) and operations G, D : (Noe) $\to$ Noe, VA: (Noe) $\to$ Val and T: $\to$ Noe. We are not so much interested by a specific algebra, but a family of algebras satisfying properties.

In these algebras, G and D are not defined everywhere. We call them *partial algebras*. Those algebras are the *daughter algebras* of the abstract data type, called *the binary trees*.

Thus the binary tree of Figure I.1 can be represented by a finite algebra where $\mathsf{Noe} = \{\circledast, \circleddash, \circledcirc, \ominus\}$, $\mathsf{Val} = \{a, b, c\}$ and the operations are defined by

	G	D	VA
$\circledast$	$\circleddash$	$\circledcirc$	a
$\circleddash$	$\ominus$		a
$\ominus$			b
$\circledcirc$			c

$\mathsf{T} = \circledast$

Here are the properties that this algebra has to satisfy.

a) *A node is on the left of at most another node and on right of at most another node.* To claim this property we need another set Bool and another total operation EG : (Noe, Noe) $\to$ Bool together with

$$\mathsf{TRUE}, \mathsf{FALSE} : () \to \mathsf{Bool}.$$

We also claim the following inequations:

$$\left.\begin{array}{rcl} \mathsf{EG}(x,y) & \sqsubseteq & \mathsf{EG}(y,x) \\ \mathsf{EG}(\mathsf{G}(x),\mathsf{G}(y)) & \sqsubseteq & \mathsf{EG}(x,y) \\ \mathsf{EG}(\mathsf{D}(x),\mathsf{D}(y)) & \sqsubseteq & \mathsf{EG}(x,y) \\ \mathsf{EG}(\mathsf{G}(x),\mathsf{D}(y)) & \sqsubseteq & \mathsf{FALSE} \\ \mathsf{EG}(\mathsf{G}(x),\mathsf{T}) & \sqsubseteq & \mathsf{FALSE} \\ \mathsf{EG}(\mathsf{D}(x),\mathsf{T}) & \sqsubseteq & \mathsf{FALSE} \\ \mathsf{EG}(\mathsf{T},\mathsf{T}) & \sqsubseteq & \mathsf{TRUE} \end{array}\right\} \tag{I.6}$$

Here we use the sign $\sqsubseteq$ to tell that the left-hand side is less defined than the right-hand side, or, said otherwise, that if the left-hand side is defined then the right-side is defined and both side are equal. Thus the algebra (Figure I.2) such that $\mathsf{Noe} = \{*, \triangle, \square\}$ and such that G, D and VA are defined by

	G	D	VA
$*$	$\triangle$	$\square$	a
$\triangle$		$\square$	b
			c

$\mathsf{T} = *$

is not candidate to represent a tree. Indeed EG does not satisfy the relation

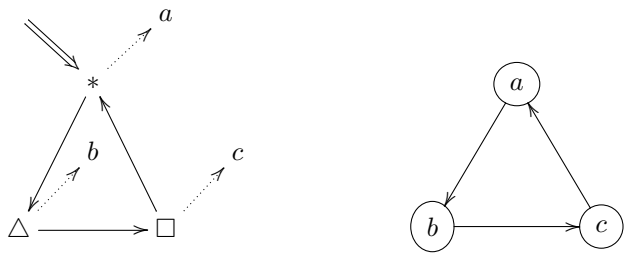


Figure I.2: This is not a tree

$$\mathsf{EG}(\mathsf{D}(x),\mathsf{D}(y)) \sqsubseteq \mathsf{EG}(x,y)$$

since if $x = *$ and $y = \triangle$ then $\mathsf{D}(x) = \mathsf{D}(y) = \square$.

b) *Each node of the algebra is reached.* This is expressed by saying that all the operations of signature ()→Noe, which are defined, represent all the nodes. In term of algebra this means that the algebra is generated by $\emptyset$ (see [Pie68] p. 105) or what is equivalent that the algebra does not contain proper sub-algebras, like previously. We will call such algebras, *primary algebras*. For instance, the algebra such that $\mathsf{Noe} = \{\oplus, \otimes, \ominus, \odot, \oslash\}$,

	G	D	VA
$\oplus$	$\otimes$		a
$\otimes$			b
$\ominus$	$\odot$	$\oslash$	a
$\odot$		$\oslash$	b
$\oslash$		$\odot$	c

$T = \oplus$

EG	$\oplus$	$\otimes$	$\ominus$	$\odot$	$\oslash$
$\oplus$	TRUE	FALSE	FALSE	FALSE	FALSE
$\otimes$	FALSE	TRUE	FALSE	FALSE	FALSE
$\ominus$	FALSE	FALSE	TRUE	TRUE	TRUE
$\odot$	FALSE	FALSE	TRUE	TRUE	TRUE
$\oslash$	FALSE	FALSE	TRUE	TRUE	TRUE

satisfies the equations and the inequations of a) despite this is not a tree (see Figure I.3).

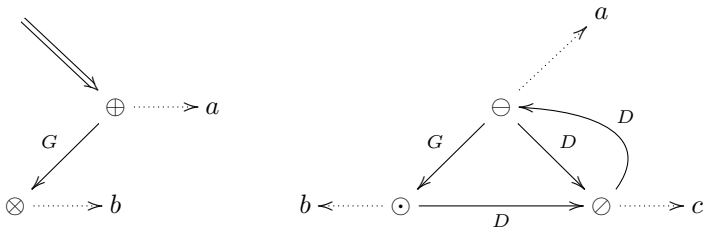


Figure I.3: This is still not a tree

c) *The operations* VA *and* EG *which provide external values are total.*

d) *For each algebra* $\mathfrak{A}$ *the sets* $\mathsf{Val}_{\mathfrak{A}}$ *and* $\mathsf{Bool}_{\mathfrak{A}}$ *are the same.*

We write $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ the class of algebras that satisfies a), b), c) and d).

Let us consider on $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ several operations.

Cons: $\mathcal{C}ons : \underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}] \times \mathsf{Alph} \times \underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}] \to \underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ is defined as follows.

Given two daughter algebras (i.e., two binary trees) $\mathfrak{A}$ and $\mathfrak{B}$ and $v \in \mathsf{V}$, $\mathcal{C}ons(\mathfrak{A}, v, \mathfrak{B})$ is the algebra $\mathfrak{C}$ such that

- $C = * \cup g \times A \cup d \times B$ where $A = \mathsf{Noe}_{\mathfrak{A}}$, $B = \mathsf{Noe}_{\mathfrak{B}}$ and $C = \mathsf{Noe}_{\mathfrak{C}}$[5]
- $\mathsf{T} = *$
- $\mathsf{G}_{\mathfrak{C}}(*) = (g, \mathsf{T}_{\mathfrak{A}})$
- $\mathsf{D}_{\mathfrak{C}}(*) = (d, \mathsf{T}_{\mathfrak{B}})$
- $\mathsf{G}_{\mathfrak{C}}((g, x)) = (g, \mathsf{G}_{\mathfrak{A}}(x))$ $\qquad$ $\mathsf{D}_{\mathfrak{C}}((g, x)) = (g, \mathsf{D}_{\mathfrak{A}}(x))$
- $\mathsf{G}_{\mathfrak{C}}((d, x)) = (d, \mathsf{G}_{\mathfrak{A}}(x))$ $\qquad$ $\mathsf{D}_{\mathfrak{C}}((d, x)) = (d, \mathsf{D}_{\mathfrak{A}}(x))$

and such that

[5] *Note at translation:* $*$, g and d are three fresh values.

- $\mathsf{EG}_{\mathfrak{C}}(*,*) = \mathsf{TRUE}$
- $\mathsf{EG}_{\mathfrak{C}}((g,x),(g,y)) = EG_{\mathfrak{A}}(x,y)$
- $\mathsf{EG}_{\mathfrak{C}}((d,x),(d,y)) = EG_{\mathfrak{B}}(x,y)$
- $\mathsf{EG}_{\mathfrak{C}}(*,(f,x)) = \mathsf{EG}_{\mathfrak{C}}((f,x),*) = \mathsf{FALSE}$ where $f = g$ or $f = d$
- $\mathsf{EG}_{\mathfrak{C}}((g,x),(d,y)) = \mathsf{EG}_{\mathfrak{C}}((d,x'),(g,y')) = \mathsf{FALSE}$

and such that

- $V_{\mathfrak{C}}(*) = v$
- $V_{\mathfrak{C}}((g,x)) = V_{\mathfrak{A}}(x)$
- $V_{\mathfrak{C}}((d,x)) = V_{\mathfrak{B}}(x)$.

We have to prove:

Lemma 1. *$Cons(\mathfrak{A}, v, \mathfrak{B})$ is an algebra of $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$.*

Elements of a proof

a) We check by case that $\mathsf{EG}_{\mathfrak{C}}$ satisfies the inequalities of I.6. For instance,

$$\mathsf{EG}_{\mathfrak{C}}(\mathsf{G}_{\mathfrak{C}}(x), \mathsf{G}_{\mathfrak{C}}(y)) \quad \sqsubseteq \quad \mathsf{EG}_{\mathfrak{C}}(x,y)$$

in the case $x = *$ and $y = (g, y')$

$$\begin{array}{rcl} \mathsf{EG}_{\mathfrak{C}}(\mathsf{G}_{\mathfrak{C}}(*), \mathsf{G}_{\mathfrak{C}}((g,y')) & = & \mathsf{EG}_{\mathfrak{C}}((g, \mathsf{T}_{\mathfrak{A}}), (g, \mathsf{G}_{\mathfrak{A}}(y'))) \\ & = & \mathsf{EG}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}}, \mathsf{G}_{\mathfrak{A}}(y')) \\ & = & \mathsf{FALSE} \\ & = & \mathsf{EG}_{\mathfrak{C}}(*, (g,y')) \end{array}$$

b) $\mathfrak{C}$ is a primary algebra. Indeed each element of A is represented by a composed operation of signature $p_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}}) : () \to \mathsf{Noe}$. Therefore each element of the form (g,x) is reached by a composed operation

$$(g, p_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}})) = p_{\mathfrak{C}}(g, \mathsf{T}_{\mathfrak{A}}) = p_{\mathfrak{C}}(\mathsf{G}_{\mathfrak{C}}(\mathsf{T}_{\mathfrak{C}}))$$

where $p_{\mathfrak{C}}$ corresponds to $p_{\mathfrak{A}}$ in $\mathfrak{C}$. We do likewise for elements of the form $(d, p_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}}))$. For $*$, there is nothing to prove since it is represented by $\mathsf{T}_{\mathfrak{C}}$.

c) The operations $\mathsf{VA}_{\mathfrak{C}}$ and $\mathsf{EG}_{\mathfrak{C}}$ are trivially total.

d) Val and Bool are preserved. □

$\mathcal{G}au$**:** $\mathcal{G}au : \underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}] \to \underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ is defined as follows: let $\mathfrak{A}$ be a binary tree, let $\mathfrak{B}$ be the algebra such that:

$$\begin{aligned} \mathsf{Noe}_{\mathfrak{B}} &= \mathsf{Noe}_{\mathfrak{A}} & \mathsf{T}_{\mathfrak{B}} &= \mathsf{G}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}}) \\ \mathsf{G}_{\mathfrak{B}} &= \mathsf{G}_{\mathfrak{A}} & \mathsf{D}_{\mathfrak{B}} &= \mathsf{D}_{\mathfrak{A}} \\ \mathsf{VA}_{\mathfrak{B}} &= \mathsf{VA}_{\mathfrak{A}} & \mathsf{EG}_{\mathfrak{B}} &= \mathsf{EG}_{\mathfrak{A}} \end{aligned}$$

$\mathfrak{B} = \mathcal{G}au(\mathfrak{A})$ is the smallest sub-algebra (with operations $\mathsf{T}_{\mathfrak{B}}$, $\mathsf{G}_{\mathfrak{B}}$, $\mathsf{D}_{\mathfrak{B}}$, $\mathsf{VA}_{\mathfrak{B}}$, $\mathsf{EQ}_{\mathfrak{B}}$), namely the algebra generated by $\emptyset$.

Lemma 2. *$\mathcal{G}au(\mathfrak{A})$ is an algebra of $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$.*

Proof. b), c), d) are obvious. For a), we have just to prove the equations dealing with $\mathsf{T}_{\mathfrak{B}}$, namely

$$\mathsf{EG}_{\mathfrak{B}}(\mathsf{D}_{\mathfrak{B}}(x), \mathsf{T}_{\mathfrak{B}}) = \mathsf{EG}_{\mathfrak{A}}(\mathsf{D}_{\mathfrak{A}}(x), \mathsf{G}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}})) = \mathsf{FALSE}$$

$$\mathsf{EG}_{\mathfrak{B}}(\mathsf{G}_{\mathfrak{B}}(x), \mathsf{T}_{\mathfrak{B}}) = \mathsf{EG}_{\mathfrak{A}}(\mathsf{G}_{\mathfrak{A}}(x), \mathsf{G}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}})) = \mathsf{EG}_{\mathfrak{A}}(x, \mathsf{T}_{\mathfrak{A}})$$

Since $x \in B$ and $\mathfrak{B}$ is a primary algebra, x is of the form $x = p_{\mathfrak{B}}$ where $p_{\mathfrak{B}}$ is a variable free expression of $\mathfrak{B}$. Therefore there exists $y \in A$ such that $x = \mathsf{G}_{\mathfrak{B}}(y)$ or $x = \mathsf{D}_{\mathfrak{B}}(y)$. Hence

$$\mathsf{EG}_{\mathfrak{B}}(x, \mathsf{T}_{\mathfrak{B}}) = \mathsf{EG}_{\mathfrak{B}}(\mathsf{G}_{\mathfrak{B}}(y), \mathsf{T}_{\mathfrak{B}}) = \mathsf{FALSE}$$

or

$$\mathsf{EG}_{\mathfrak{B}}(x, \mathsf{T}_{\mathfrak{B}}) = \mathsf{EG}_{\mathfrak{B}}(\mathsf{D}_{\mathfrak{B}}(y), \mathsf{T}_{\mathfrak{B}}) = \mathsf{FALSE}$$

Finally

$$\mathsf{EG}_{\mathfrak{B}}(\mathsf{T}\mathfrak{B}, \mathsf{T}_{\mathfrak{B}}) = \mathsf{EG}_{\mathfrak{A}}(\mathsf{G}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}}), \mathsf{G}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}})) = \mathsf{EG}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}}, \mathsf{T}_{\mathfrak{A}}) = \mathsf{TRUE}.$$

□

$\mathcal{D}ro$: $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}] \to \underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ is defined like $\mathcal{G}au$ with the difference that $\mathsf{T}_{\mathfrak{B}} = \mathsf{D}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}})$.

Let us consider now a selection.

$\mathcal{T}et$: $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}] \to \mathsf{Alph}$ defined as $\mathcal{T}et(\mathfrak{A}) = \mathsf{VA}_{\mathfrak{A}}(\mathsf{T}_{\mathfrak{A}})$.

Morphisms In $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ the morphims h are maps $\mathsf{Noe}_{\mathfrak{A}} \to \mathsf{Noe}_{\mathfrak{B}}$ which satisfy:

- $\mathsf{T}_{\mathfrak{B}}$ is defined when it $\mathsf{T}_{\mathfrak{A}}$ is and
$$\mathsf{T}_{\mathfrak{B}} = h(\mathsf{T}_{\mathfrak{A}})$$
- if $\mathsf{G}_{\mathfrak{A}}$ is defined than $h(\mathsf{G}_{\mathfrak{B}})$ is defined and
$$\mathsf{G}_{\mathfrak{A}} = h(\mathsf{G}_{\mathfrak{B}})$$

- if $\mathsf{D}_{\mathfrak{A}}$ is defined than $h(\mathsf{D}_{\mathfrak{B}})$ is defined and

$$\mathsf{D}_{\mathfrak{A}} = h(\mathsf{D}_{\mathfrak{B}})$$

- Moreover,

$$\begin{aligned} \mathsf{VA}_{\mathfrak{B}} &= h(\mathsf{VA}_{\mathfrak{A}}) \\ \mathsf{EG}_{\mathfrak{A}}(x,y) &= \mathsf{EG}_{\mathfrak{B}}(h(x),h(y)) \end{aligned}$$

If h is bijective, we say that h is an isomorphism. Since algebras are partial algebras, there exists a unique algebra, i.e., a binary tree, with an empty carrier Noe. Let us call it $\mathcal{V}id$. In $\mathcal{V}id$, $\mathsf{T}_{\mathcal{V}id}$ is not defined. It is obviously primary. Naturally it represents the empty tree. Notice that if $\mathfrak{A}$ is an algebra of $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph},\mathsf{Bool}]$ which is not $\mathcal{V}id$, $\mathsf{T}_{\mathfrak{A}}$ is defined, otherwise, $\mathfrak{A}$ would have a sub-algebra isomorphic to $\mathcal{V}id$.

Let us focus on the class of algebras in $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph},\mathsf{Bool}]$, in which $\mathsf{Noe}_{\mathfrak{A}}$ is finite and which a subset of a finite enumerable set D. If we want $\mathcal{C}ons$, $\mathcal{G}au$ and $\mathcal{D}ro$ to be stable in this class we must have $* \in D$ and if $A \subseteq D$ then $\{g\} \times D \subseteq D$ and $\{d\} \times D \subseteq D$. Henceforth, we choose for D the smallest set such that

$$D \supseteq \{g,d\} \times D$$

The isomorphism $\simeq$ is a equivalence relation on the algebras of carrier D. We show easily that $\mathcal{C}ons$, $\mathcal{D}ro$, $\mathcal{G}au$ preserve isomorphism. That the value of $\mathcal{T}et$ does not change if we take an isomorphic algebra and that $\mathcal{V}id$ does not admit isomorphic algebras besides itself. Let us call $\underline{\underline{\underline{\mathsf{AAF}}}}[\mathsf{Alph},\mathsf{Bool}]$ the equivalence class of algebras modulo isomorphism with nodes in D. Let us call $\mathcal{C}ons'$, $\mathcal{G}au'$, $\mathcal{D}ro'$, $\mathcal{T}et'$ and $\mathcal{V}id'$ the operations on $\underline{\underline{\underline{\mathsf{AAF}}}}[\mathsf{Alph},\mathsf{Bool}]$ deduced from $\mathcal{C}ons$, $\mathcal{G}au$, $\mathcal{D}ro$, $\mathcal{T}et$ and $\mathcal{V}id$. We get the following theorem.

Theorem 1 (Algebraic representation of binary algebras)**.** *The algebra of* $\underline{\underline{\underline{\mathsf{ARB}}}}[\mathsf{Alph}]$ *where* $\mathsf{Arb} = \underline{\underline{\underline{\mathsf{AAF}}}}[\mathsf{Alph},\mathsf{Bool}]$ *and which has the operations* $\mathcal{C}ons'$, $\mathcal{G}au'$, $\mathcal{D}ro'$, $\mathcal{T}et'$ *and* $\mathcal{V}id'$ *is an algebra of* $\underline{\underline{\underline{\mathsf{ARB}}}}[\mathsf{Alph}]$*. Moreover this is the initial algebra of* $\underline{\underline{\underline{\mathsf{ARB}}}}[\mathsf{Alph}]$*.*

Proof. Show first that $\mathfrak{C} = \mathcal{G}au(\mathcal{C}ons(\mathfrak{A},v,\mathfrak{B})) \simeq \mathfrak{A}$.

Consider the bijection $h : \mathfrak{A} \to \{g\} \times \mathfrak{A}$, such that $h(a) = g \times a$. This is a morphism from $\mathfrak{A}$ to $\mathfrak{C}$. Indeed let us state $\mathfrak{D} = \mathcal{C}ons(\mathfrak{A},v,\mathfrak{B})$.

- $h(\mathsf{T}_{\mathfrak{A}}) = T_{\mathfrak{C}}$ because $\mathsf{T}_{\mathfrak{C}} = \mathsf{G}_{\mathfrak{D}}(\mathsf{T}_{\mathfrak{D}}) = g \times \mathsf{T}_{\mathfrak{A}} = h(\mathsf{T}_{\mathfrak{A}})$.
- $h(\mathsf{G}_{\mathfrak{A}}(x)) = \mathsf{G}_{\mathfrak{C}}(h(x))$ because

$$\mathsf{G}_{\mathfrak{C}}(h(x)) = \mathsf{G}_{\mathfrak{C}}((g,x)) = \mathsf{G}_{\mathfrak{D}}((g,x)) = (g,\mathsf{G}_{\mathfrak{A}}(x)) = h(\mathsf{G}_{\mathfrak{A}}(x))$$

The proof of the equality $h(\mathsf{D}_{\mathfrak{A}}(x)) = \mathsf{D}_{\mathfrak{C}}(h(x))$ is the same as are the proofs of $\mathsf{VA}_{\mathfrak{A}}(x) = \mathsf{VA}_{\mathfrak{C}}(h(x))$ and $\mathsf{EG}_{\mathfrak{A}}(x,y) = \mathsf{EG}_{\mathfrak{C}}(h(x),h(y))$. We also show that $\mathcal{D}ro(\mathcal{C}ons(\mathfrak{A},v,\mathfrak{B})) \simeq \mathfrak{B}$. Moreover we have

$$\mathcal{T}et(\mathcal{C}ons(\mathfrak{A},v,\mathfrak{B})) = v.$$

Indeed

$$\mathcal{T}et(\mathcal{C}ons(\mathfrak{A}, v, \mathfrak{B})) = \mathsf{VA}_{\mathfrak{D}}(\mathsf{T}_{\mathfrak{D}}) = \mathsf{VA}_{\mathfrak{D}}(*) = v$$

To prove that $\mathcal{G}au(\mathcal{V}id) = \mathcal{V}id$, we have just to look at the definition $\mathcal{G}au(\mathcal{V}id)$. Let us consider the algebra $\mathfrak{B}$ such that $\mathsf{Noe}_{\mathfrak{A}} = \mathsf{Noe}_{\mathcal{V}id} = \emptyset$. $\mathsf{T}_{\mathfrak{B}} = \mathsf{G}_{\mathcal{V}id}(\mathsf{T}_{\mathcal{V}id})$ is indeed not defined. This is the same for $\mathsf{G}_{\mathfrak{B}} = \mathsf{G}_{\mathcal{V}id}$, $\mathsf{D}_{\mathfrak{B}} = \mathsf{D}_{\mathcal{V}id}$, $\mathsf{VA}_{\mathfrak{B}} = \mathsf{VA}_{\mathcal{V}id}$, $\mathsf{EQ}_{\mathfrak{B}} = \mathsf{EQ}_{\mathcal{V}id}$. Thus $\mathcal{G}au(\mathcal{V}id)$ which is the smallest sub-algebra of $\mathfrak{B}$ is $\mathcal{V}id$ itself. Therefore $\mathcal{G}au(\mathcal{V}id) = \mathcal{V}id$. We prove as well than $\mathcal{D}ro(\mathcal{V}id) = \mathcal{V}id$. To prove that this is an initial algebra, we need a lemma.

Lemma 3. *If t is a variable-free term built using the operations of the variety* $\underline{\underline{\mathsf{Arb}}}[\mathsf{Alph}]$*, then there exists a variable-free term t', built using only $\mathcal{C}ons$ and $\mathcal{V}id$ such that $t = t'$.*

Proof of the lemma. By induction on the number of $\mathcal{G}au$'s and $\mathcal{D}ro$'s in t. Assume that there exist occurrences of $\mathcal{G}au$ or $\mathcal{D}ro$ in t. Hence there exists at least one occurrence, written f, occurring in $f(\mathcal{V}id)$ or in $f(\mathcal{C}ons(g, v, d))$.

If this is of the first form, the term t'' in which we substitute $\mathcal{V}id$ to $f(\mathcal{V}id)$ contains less occurrences of $\mathcal{G}au$ or $\mathcal{D}ro$ and satisfies $t'' = t$.

If this is of the second form, the term t'' in which, we substitute, in t, g to $f(\mathsf{CONS}(g, v, h))$, if $f = \mathsf{GAU}$ and h to $f(\mathsf{CONS}(g, v, h))$, if $f = \mathsf{DRO}$, contains less occurrences of GAU and DRO and satisfies $t = t''$. □

Proof of the theorem (next)

Each daughter algebra of $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ has only finitely many elements in Noe. Moreover, if $\mathfrak{A}$ is not $\mathcal{V}id$, $\mathcal{G}au(\mathfrak{A})$ and $\mathcal{D}ro(\mathfrak{A})$ are algebras with less elements in Noe, fulfilling the property:

$$\mathfrak{A} = \mathcal{C}ons(\mathcal{G}au(\mathfrak{A}), v, \mathcal{D}ro(\mathfrak{A}))$$

Thus we can show that $\mathfrak{A}$ can be decomposed uniquely, up to an isomorphism. This result and the lemma show that the algebra is initial. Proposition 3, of Section 2 of Chapter II will yield a simpler proof of the initially of $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$. □

2.3 Links with programming

The theorem we have just proved can be formulated as follows: "The algebra $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ is a *representation* of the type 'binary tree' by objects and operations of a lower level of abstraction". Those objects are more concrete, because, among other concepts, they are built using pointers. This modelling is a step toward programming, because daughter algebras, i.e., the algebras of $\underline{\underline{\mathsf{AA}}}[\mathsf{Alph}, \mathsf{Bool}]$ are a good approach for objects represented in the computer, by a programming language. The properties a), b), c) d) are invariants of constructions. For instance, the statement: "the algebra is primary" is for the programmer the natural statement: "at any time, a tree contains only nodes that can be reached from the head".

Translation in a programming language

In ALGOL 68

As an illustration, we show how the objects of AA[Alph, Bool] can be translated into ALGOL 68. Sorts are *modes*[6]. We assume that the mode Alph is known.

mode Alph...

The mode Noe should provide a mean to yield a value by VA and to access two nodes by G and D. It is naturally translated by:

mode Noe struct(*rep Noe* G, *Alph* VA, *rep Noe* D)

As we saw, a tree gives a node, a value and two access functions G and D yielding a node from another node, through a pointer. VA is available at each node. Therefore T is the only node that we must provide when we give a tree. Thus a tree is given by the only natural access T

$$\textit{mode Arb} \quad = \quad \textit{struct}(\textit{rep Noe } \mathsf{T}) \tag{I.7}$$

This definition may look sophisticated to an expert programmer (we discuss further a definition of a tree by *mode Arb rep Noe* and the removal of all the expressions "T of". At first , we keep Definition I.7 to show the reader how this fits with the algebras we described.)

Constructions and selections are described by procedures.

- Thus GAU is described by

  proc gau $=$ $(\textit{Arb } a)$ *Arb* : (G *of* T *of* a)

- CONS is described by

  proc cons $=$ $(\textit{Arb } a, \textit{Alph } v, \textit{Arb } b)$ *Arb* :
  begin heap Noe* := (T *of* $a, v,$ T *of* b)

In a language a la Pascal

We give a translation of the same objects and the same procedures in a Pascal-like language with the difference that procedures are allowed to return results of all the types described in the language:

```
type     alph = ... ; noe = record G : ↑noe ; VA : alph ; D : ↑noe  end ;
          arb = record  T : ↑noe   end  ;
function gau(a : arb) : arb ; begin gau. T : = a. T↑. G  end  ;
function dro(a : arb) : arb ; begin dro. T : = a. T↑. D  end  ;
function tet(a : arb) : alph ;begin  tet : = a. T↑. VA   end  ;
function cons(a : arb ; v : alph ; b : arb) : arb ;
      begin var étoile : ↑noe ; new (étoile) ;
            étoile↑.G : = a. T ; étoile ↑.VA : = v ; étoile↑.D : = b.T
            cons.T : = étoile
      end
```

[6] *Note at translation: modes* are what languages like ML or Haskell call a *types*.

If now in the Algol 68 program we replace the declaration *mode Arb* = *rep struct*(*rep Noe T*) by *mode Arb* = (*rep Noe*) and if we remove expression T *of* we get:

mode Alph = ...; mode Noe = struct (rep Noe G, Alph VA, rep Noe D);
mode Arb = rep Noe;
proc gau = (Arb a) Arb : G of a;
proc dro = (Arb a) Arb : D of a;
proc tet = (Arb a) Arb : nil;
proc vid = (Arb a) Arb : G of a;
proc cons = (Arb a; Alph v; Arb b) Arb : heap Arb := (a, v, b) .

Examining the mode declarations of *Noe* and *Alph*, we notice than we can change them without changing the program by the unique declaration:

mode Arb = rep struct (Arb G, Alph VA, Arb D).

The only change will be on *cons*:

proc cons = (Arb a, Alph v, Arb b) Arb :
heap struct (Arb G; Alph VA; Arb D) Arb := (a, v, b) .

Discussion To state *mode Arb = rep struct (Arb G, Alph VA, Arb D),* is to define the domain *Arb* as a fixed point:

$$Arb = \{VID\} + (Arb \times Alph \times Arb)$$

This is an extensional definition à la Scott (cf [LS77]) where the algebraic aspect disappeared. An approach aiming at narrowing the extensional and the algebraic aspects would be probably extremely fruitfully. However the definition of the type Arb by

mode Arb = rep struct (Arb G, Alph VA, Arb D).

has two risks (see in particular the article of J. H. Morris "Types are not sets" [Mor73]).

- The best known is surely the use of operations not foreseen when describing the type, with risks when running the program, which has much chance to bug. Assignments like

  $$\mathsf{G}\ of\ z := z.$$

  generate monsters which are not trees at least not finite trees. Indeed the object α represented by z satisfies

  $$\mathsf{G}\ of\ \alpha = \alpha.$$

  Notice that programming languages like CLU, ALPHARD, ATM[7] protect the programmer against such risks ([LZ74, WLS76, CCD+79])

[7] *Note at translation:* ATM is a confidential programming language developed in Nancy.

- The second danger is the confusion of two different concepts, this of node and this of tree. Figure I.4 aims at illustrating the difference between those two concepts. A node is an internal object that must be hidden to the user of trees. This principle of hiding information is the warranty of reliable software, because it avoids the modification of values that the user should not access to. This protection could be enforced if we can forbid the user to know about Noe. This could be done by putting the declaration of Noe in a part of the program, not accessible to the user, for instance, in the prologue. Thus the distinction between Noe and Arb and the consecutive distinctions between GAU and G, DRO and D and TET and T do not allow, for instance, the assignment G of z := z, already noticed, since we can only access the nodes through the procedures gau, dro and tet. Let us notice that only one value is attached to a node whereas a

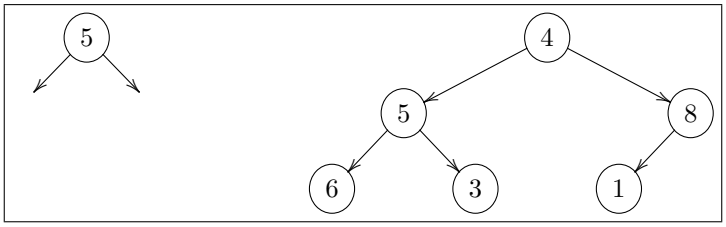


Figure I.4: A node and a tree

family of values is attached to a tree. Among them, one value is put in evidence, namely the head (called tet)

Advantage of an algebraic approach. Let us notice that when programming an abstract data type in a programming language, we have to prove that the procedures represent the operations. Going through daughter algebras is a convenient way to proceed.

Finally let us say that the example of binary trees has been chosen to illustrate type algebras and daughter algebras, because it is simple and because it contains dyadic operations. But this example is less convincing when we go to a representation in ALGOL 68, because this language contains structures closed to those found in trees. Examples of Section 6 and Chapter 4 will provide more convincing examples.

2.4 Binary trees and sharing

In the binary trees that we encountered, no sharing was possible. In the following a restricted form of sharing is allowed. A same node can be both left and right son of another node. In other words, we replace

$$\mathsf{EG}(\mathsf{D}(x), \mathsf{G}(x)) \sqsubseteq \mathsf{FALSE}$$

by

$$(\mathsf{EG}(\mathsf{D}(x), \mathsf{G}(y)) \Rightarrow \mathsf{EG}(x, y)) \sqsubseteq \mathsf{TRUE}$$

where $\Rightarrow$ is a well known Boolean operator.

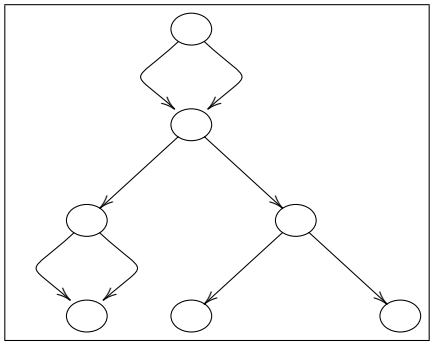


Figure I.5: A tree with sharing

On those trees, the operations $\mathcal{G}au$, $\mathcal{D}ro$, $\mathcal{T}et$ and $\mathcal{V}id$ are not modified. If $\mathfrak{A}$ and $\mathfrak{B}$ are two non isomorphic algebras, we define $\mathcal{C}ons(\mathfrak{A}, a, \mathfrak{B})$ like previously, but if the algebras are isomorphic, we define $\mathfrak{C} = \mathcal{C}ons(\mathfrak{A}, a, \mathfrak{B})$ as follows.

- $C = \{*\} \cup A$[8]
- $T_{\mathfrak{C}} = *$
- $G_{\mathfrak{C}}(*) = D_{\mathfrak{C}}(*) = T_{\mathfrak{A}}$
- if $x \neq *$ then $G_{\mathfrak{C}}(x) = G_{\mathfrak{A}}(x)$ and $D_{\mathfrak{C}}(x) = D_{\mathfrak{A}}(x)$.
- $\mathsf{EG}_{\mathfrak{C}}(*, *) = \mathsf{TRUE}$
- if $x \neq *$ then $\mathsf{EG}_{\mathfrak{C}}(x, *) = \mathsf{EG}_{\mathfrak{C}}(*, x) = \mathsf{FALSE}$
- if $x \neq *$ and $y \neq *$ then $\mathsf{EG}_{\mathfrak{C}}(x, y)) = \mathsf{EG}_{\mathfrak{C}}(y, x) = \mathsf{TRUE}$
- $\mathsf{VA}(*) = a$
- if $x \neq *$ then $\mathsf{VA}_{\mathfrak{C}}(x) = \mathsf{VA}_{\mathfrak{A}}(x)$

It is easy to prove that the algebras built that way fulfil the axioms of Figure I.6.

Let $\mathsf{Arb}_{\mathfrak{S}}$ be the class of algebras that satisfy the axioms of Figure I.6. On $\mathsf{Arb}_{\mathfrak{S}}$ and Alph, we define the operations $\mathcal{C}ons''$, $\mathcal{G}au''$, $\mathcal{D}ro''$, $\mathcal{T}et''$, $\mathcal{V}id''$, by quotienting the operations $\mathcal{C}ons$, $\mathcal{G}au$, $\mathcal{D}ro$, $\mathcal{T}et$, $\mathcal{V}id$. Thus we define an algebra of $\underline{\underline{\mathsf{ARB}}}[\mathsf{Alph}]$ which is an initial algebra in this class.

3 Recall on homogeneous algebras

This section and the two following ones can be dropped in a first reading[9].

Not translated

[8] *Note at translation:* Assuming that $* \notin A$.

[9] *Note at translation:* For this reason, it is not translated.

$$
\begin{aligned}
\mathsf{EG}(x, y) &\sqsubseteq \mathsf{EG}(y, x) \\
\mathsf{EG}(\mathsf{G}(x), \mathsf{G}(y)) &\sqsubseteq \mathsf{EG}(x, y) \\
\mathsf{EG}(\mathsf{D}(x), \mathsf{D}(y)) &\sqsubseteq \mathsf{EG}(x, y) \\
\mathsf{EG}(\mathsf{G}(x), \mathsf{D}(y)) \Rightarrow \mathsf{EG}(x, y) &\sqsubseteq \mathsf{TRUE} \\
\mathsf{EG}(\mathsf{G}(x), \mathsf{T}) &\sqsubseteq \mathsf{FALSE} \\
\mathsf{EG}(\mathsf{D}(x), \mathsf{T}) &\sqsubseteq \mathsf{FALSE} \\
\mathsf{EG}(\mathsf{T}, \mathsf{T}) &\sqsubseteq \mathsf{TRUE}
\end{aligned}
$$

Figure I.6: The axioms of binary trees with sharing

3.1 Heterogeneous algebras

3.2 Construction of algebras

3.3 Free algebras and initial algebras

3.4 Generated algebras and primary algebras

3.5 Polynomial functions

3.6 Identities on algebras

4 Heterogeneous paramaterised algebras

Not translated

4.1 Heterogeneous algebras and partial heterogeneous algebras

4.2 Free heterogeneous algebras and initial free heterogeneous algebras

4.3 Generated heterogeneous algebras and primary heterogeneous algebras

4.4 Final algebras

4.5 Polynomial functions

4.6 Identities on heterogeneous algebras

5 Paramaterised heterogeneous algebras

6 Three examples: File, Set and Circular Lists

The ~~two~~ three[10] following abstract data types, which we are going to study, will contribute with the abstract data type Binary Tree to illustrate the next chapter.

6.1 The Files

We will consider the files, denoted File, on an alphabet Alph. The operators are the following:

FILEVIDE : () → File,

AJ : (File, Alph) → File,

OT : (File) → File,

FR : (File) → Alph ∪ (INDEF),

CONCAT : (File, File) → File ;

They satisfy the identities:

(OT1)	OT(FILEVIDE) = FILEVIDE.
(OT2)	OT(AJ(FILEVIDE, a))) FILEVIDE,
(OT3)	OT(AJ(AJ(f,), a), b)) AJ(OT(AJ(f,a)),b),
(FR1)	FR(FILEVIDE) = INDEF,
(FR2)	FR(AJ(FILEVIDE, a)) = a,
(FR3)	FR(AJ(AJ(f, a), b)) = FR(AJ(f,a))
(CONCAT1)	CONCAT(f, FILEVIDE) = f
(CONCAT2)	CONCAT(f,AJ(f',a)) = AJ(CONCAT(f,f'),a).

[10] *Note at translation:* The example of Circular Lists is Section 4.4 of Chapter II in the original document.

This abstract data type has deep differences with the abstract data type, 'binary trees', which we studied in Section 2. In binary trees GAU and DRO plaid antagonistic roles w.r.t. CONS. Here OT is not antagonistic w.r.t. AJ: we do not "remove" the last element which we "added"; the specification of OT is recursive, thus the left-hand side of the third equation contains OT; we find similar properties with the specification of FR. CONCAT allows us to build a file from two other files, but each file can be expressed uniquely from AJ, so that in CONCAT(f,f'), the operator CONCAT can disappear completely. This is what we call a *secondary operator*. Such an operator does not exist in binary trees.

The class of models FILE[Alph] proposed here is made of primary partial finite algebras with the following operations:

SU : (Place) $\rightarrow$ Place,

PREM : () $\rightarrow$ Place,

VA : (Place) $\rightarrow$ Alph,

EG : (Place, Place) $\rightarrow$ Bool.

It satisfies the following inequations:

EG(PREM, PREM) $\sqsubseteq$ TRUE

EG(PREM, S $\cup$ {x}) $\sqsubseteq$ FALSE

EG(S $\cup$ {x}, PREM) $\sqsubseteq$ FALSE

```
type     alph = ... ; noe = record G : ↑noe ; VA : alph ; D : ↑noe  end ;
          arb = record  T : ↑noe   end  ;
function gau(a : arb) : arb ; begin gau. T : = a. T↑. G  end  ;
function dro(a : arb) : arb ; begin dro. T : = a. T↑. D  end  ;
function tet(a : arb) : alph ;begin  tet : = a. T↑. VA   end  ;
function cons(a : arb ; v : alph ; b : arb) : arb ;
      begin var étoile : ↑noe ; new (étoile) ;
            étoile↑.G : = a. T ; étoile ↑.VA : = v ; étoile↑.D : = b.T
            cons.T : = étoile
      end
```

Figure I.7: Definition of the constructions $\mathcal{O}t$, $\mathcal{A}j$, $\mathcal{C}oncat$

Notations In what follows, we use sometimes the following conventions in order to ease the understanding of formulas:

- FILEVIDE will be written $\wedge$
- AJ(f,a) will be written $f \odot a$

- CONCAT(f,f') will be written $f * f'$
- INDEF will be written ?

Thus
CONCAT(AJ(f, FR(FILEVIDE,a), OT(AJ(AJ(FILEVIDE,a), b))))
will be written

$$[f \odot \mathsf{FR}(\wedge \odot a)] = \mathsf{OT}(\wedge \odot a \odot b)$$

6.2 The sets

For the set[11] we propose here the following description:

VIDE : () → Ens,

AJ : (Ens, Alph) → Ens,

PR : (Ens, Alph) → Bool.

They satisfy the following identities:

PR(AJ(e, a), b) = **if** EG(A, b) **then** TRUE **else** PR(a, b)

PR(VIDE, a) = FALSE

Three families of models will be presented.[12]

The *first* family is made of algebras with essentially an operator ∈: (Alph → Bool. We notice that in these models, no sort, in other words, no part of the object, is hidden from outside. No category of objects has been added and nothings holds for the sort Noe in trees or of the sort Place in files. We define easily an operator.

- $\mathcal{V}ide$ is defined as $\in_{Vide} \equiv$ FALSE
- $\mathcal{A}j(\mathfrak{A}, A)$ is defined as follows: $A \in_{\mathcal{A}j(\mathfrak{A},A)} =$ TRUE and when $B \neq A$, $B \in_{\mathcal{A}j(\mathfrak{A},A)} = B \in_{\mathfrak{A}}$.
- $\mathcal{P}r(\mathfrak{A}, A) = A \in_{\mathfrak{A}}$

The *second* family is made of the same objects as files, but

- $\mathcal{V}ide$ is the algebra such that Place = $\emptyset$,
- $\mathcal{A}j(\mathfrak{A}, a)$ is the algebra $\mathfrak{B}$ such that $\mathsf{Place}_{\mathfrak{B}} = \mathsf{Place}_{\mathfrak{A}} \cup \{\mathsf{Place}_{\mathfrak{A}}\}$. Let us write α the new element:

  $\mathsf{PREM}_{\mathfrak{A}} = \alpha$.

[11] *Note at translation:* "ensemble" means "set" and "vide" means "empty". $\mathcal{P}r$ stands for "present" an alternative to "is in" in order to avoid confusion.

[12] *Note at translation:* Actually four families are presented.

$$
\begin{array}{rll}
\textbf{if } x \in \mathsf{Place}_{\mathfrak{A}} & \textbf{then} & S \cup \{x\} \wedge \mathsf{VA}_{\mathfrak{B}}(x) = \mathsf{VA}_{\mathfrak{A}}(x) \\
& \textbf{else} & S \cup \{x\} = \mathsf{PREM}_{\mathfrak{A}} \wedge \mathsf{VA}_{\mathfrak{A}}(x) = a \\
\textbf{if } x, y \in \mathsf{Place}_{\mathfrak{A}} & \textbf{then} & \mathsf{EQ}_{\mathfrak{B}}(x, y) = \mathsf{EQ}_{\mathfrak{A}}(x, y) \\
\textbf{if } x \in \mathsf{Place}_{\mathfrak{A}} & \textbf{then} & \mathsf{EQ}_{\mathfrak{B}}(x, \alpha) = \mathsf{FALSE} = \mathsf{EQ}_{\mathfrak{B}}(\alpha, x) \\
\textbf{eventually} & & \mathsf{EQ}_{\mathfrak{B}}(\alpha, \alpha) = \mathsf{TRUE}
\end{array}
$$

- $\mathcal{P}r(\mathfrak{A}, a) = (\exists x \in \mathsf{Place}_{\mathfrak{A}})(\mathsf{VA}_{\mathfrak{A}}(x) = a)$

The *third* family is less orthodox, it assumes that Alph contents one specific element which we write O. If Alph would be the set of the non negative numbers, it would be *zero* for instance. In those models, we define a sort Place. There are three operators of the previous model and one operator OE: ()$\rightarrow$ Bool which tells whether O belongs to the set or not:

- $\mathcal{V}ide$ is the algebra such $\mathsf{Place} = \emptyset$ and $\mathsf{OE} = \mathsf{FALSE}$.
- In order to define $\mathcal{A}j(\mathfrak{A}, a)$ two cases are to be considered
  - if $a \neq \mathsf{O}$ then $\mathcal{A}j(\mathfrak{A}, a)$ is defined as in the previous representation and $\mathsf{OE}_{\mathcal{A}j(\mathfrak{A},a)} = \mathsf{OE}_{\mathfrak{A}}$.
  - $\mathcal{A}j(\mathfrak{A}, \mathsf{O})$ is the algebra $\mathfrak{B}$ such that $\mathsf{Place}_{\mathfrak{B}} = \mathsf{Place}_{\mathfrak{A}}$. All the operations are the same on $\mathfrak{A}$ and on $\mathfrak{B}$, except possibly $\mathsf{OE}_{\mathfrak{B}}$ which takes the value TRUE.
  - If $a \neq \mathsf{O}$, $\mathcal{P}r(\mathfrak{A}, a) = (\exists x \in \mathsf{Place}_{\mathfrak{A}})\mathsf{VA}_{\mathfrak{A}} = a$.
    If $a = \mathsf{O}$, $\mathcal{P}r(\mathfrak{A}, \mathsf{O}) = \mathsf{OE}_{\mathfrak{A}}$.

A *fourth* family represents sets by "lists without repetitions". Here again, we get a representation as lists with operators:

- SU : (Place) $\rightarrow$ Place,
- PREM : ( ) $\rightarrow$ Place,
- VA : (Place) $\rightarrow$ Alph,
- EG : (Place, Place) $\rightarrow$ Bool

It satisfies the same equations as the files.

- $\mathcal{V}ide$ is still the algebra such that $\mathsf{Place} = \emptyset$.
- $\mathcal{A}j(\mathfrak{A}, A)$ depends on the existence of a place x, such that $\mathsf{VA}_{\mathfrak{A}}(x) = a$.
  - **if** $(\exists x \in \mathsf{Place})\mathsf{VA}_{\mathfrak{A}}(x) = a)$ **then** $\mathcal{A}j(\mathfrak{A}, a) = \mathfrak{A}$
  - **else** $\mathcal{A}j(\mathfrak{A}, a)$ is defined as in the second model.
- $\mathcal{P}r(\mathfrak{A}, a) = (\exists x \in \mathsf{Place})\mathsf{VA}_{\mathfrak{A}}(x) = a$

Along the third representation, we could build other representations, in particular, not only one element in Alph, but two, three, ..., n etc. In another hand, we could develop other representations based on search binary trees [AHU74]. We will not address them here.

Notation In what follows we will adopt the following conventions:

VIDE will be written $\emptyset$.

AJ(e, a) will be written $a + e$.

6.3 The circular lists

Consider the abstract data type listcirculaire

```
Type   Listecirculaire [Alph]
   Fonctionnalité
       LVIDE :() →  Listecirculaire
       AJ    : (Listecirculaire, Alph) →  Listecirculaire,
       RDT   :(Listecirculaire) →   Listecirculaire
       TET   :(Listecirculaire) →   Alph U {INDEF}

   Axiomatique

       ROT(LVIDE) = LVIDE
       ROT(AJ(LVIDE, a)) = AJ(LVIDE, a)
       ROT(AJ(AJ(a, b), a) = AJ(ROT(AJ(x, a)), b)
       TET(x, b) = b
       TET(LVIDE) = INDEF
```

Figure I.8 represents representations[13] obtained by applying $\mathcal{R}ot$ on the *Listcirculaire* (the *circular list*) $AJ(AJ(AJ(LVIDE, c), b), a)$. We notice that we need four states for a representation which we can imagine to be much simpler, by using primary algebras. For a simple representation, we define two 0-ary operations.

$$\mathsf{T}, \mathsf{Q} : () \to \mathsf{Noe}$$

and one unary operation

$$S : (\mathsf{Noe}) \to \mathsf{Noe}.$$

S is defined everywhere and satisfies

$$\mathsf{S}(\mathsf{Q}) = \mathsf{R}.$$

It is easy to build $\mathcal{R}ot(\mathfrak{A}) = \mathfrak{B}$ as follows (Figure I.9):

$$\begin{aligned} R_{\mathfrak{B}} &= S_{\mathfrak{A}}(R_{\mathfrak{A}}) \\ S_{\mathfrak{B}} &= S_{\mathfrak{A}} \\ Q_{\mathfrak{B}} &= S_{\mathfrak{A}}(Q_{\mathfrak{A}}) \qquad \text{which is therefore } R_{\mathfrak{B}}. \end{aligned}$$

[13] *Note at translation:* This refers to a construction described in Chapter II, which consists in making each term a daughter algebra, in a rather straightforward way.

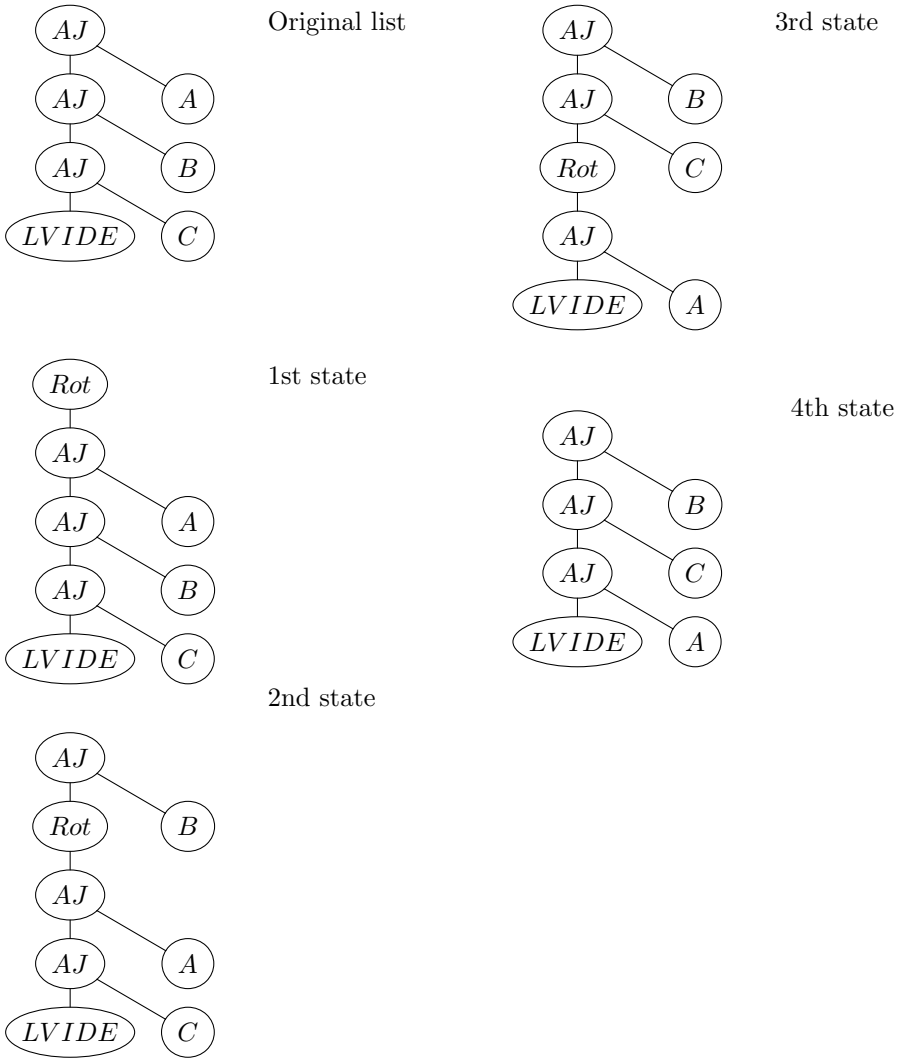


Figure I.8: Canonical representation of $\mathcal{R}ot$

It seems that the second "representation" is the most natural one and it is amazing that the abstract data type suggests the first one. This comes the fact that the algebraic specification, in other words the rewriting mechanism, induces a representation by trees instead of any other representation. This refers to a most general problem well known in data bases. It is difficult to specify simply and efficiently a circular structure and reciprocally to implement efficiently and without error the specification of such a structure.

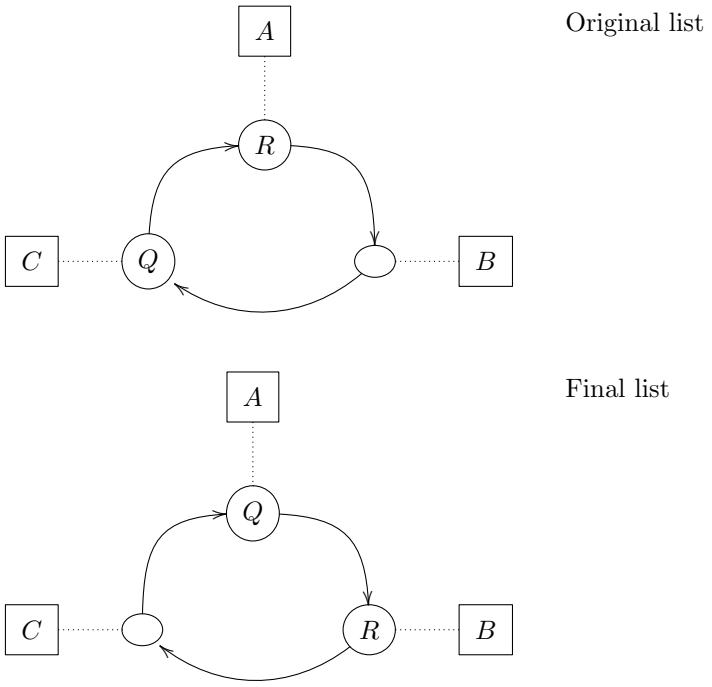


Figure I.9: A more natural representation of circular lists

Chapter II

Algebraic study of the representation of an Abstract Data Type

This chapter has essentially two parts. In the first part, we present several algebraic aspects of abstract data types and their representations: rewriting type algebras, representations by algebras. In the second part we present a canonical representation of an abstract data type.

Convention In this chapter, we take the convention to write Ti the sort associated with the abstract data type which we define, also called the *type of interest* and $Ext_1, ..., Ext_m$ the types which are parts of the definition, also called parameter types.

1 Abstract data types and rewriting

1.1 Rewriting systems [Hue77, RV80]

We can consider abstract data types as rewriting systems, where each identity is oriented, in other words, each identity is a couple (not a pair) $g \to d$ and is called a *rewrite rule*.

We present here relativity informally the concepts of rewriting. In particular, the concept of occurrence of a sub-term is presented intuitively. Huet [Hue77] gives a more complete treatment.

A *substitution* is a family $\sigma = \{x_1 := t_1, ..., x_n = t_n\}$ of couples variable-term. The application of the substitution σ to the term t is the term σt obtained by substituting each occurrence of x_i by t_i.

A term t rewrites in a term t' if

- there exists a sub-term s of t,

- there exists a substitution σ
- there exists a rule $g \to d$

such that $\sigma g = s$ and t' is the term obtained by replacing in t, s by σd.

We write then $t \to t'$. The transitive and reflexive closure is written $\stackrel{*}{\to}$.

A terms is *irreducible* if $t \stackrel{*}{\to} t'$ implies tt', in other words t can no more be reduced. A system is confluent or has the Church Rosser property if $t \stackrel{*}{\to} t_1$ and $t \stackrel{*}{\to} t_2$ imply that there exists t' such that $t_1 \stackrel{*}{\to} t'$ and $t_2 \stackrel{*}{\to} t'$ (Figure II.1). If

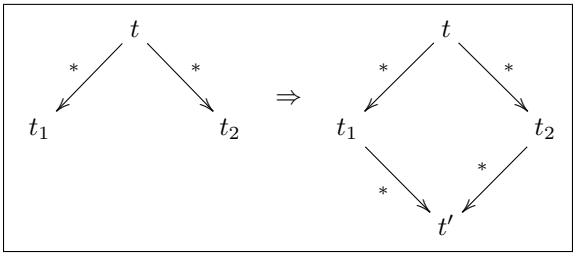


Figure II.1: Confluence or Church-Rosser property

the rewrite system is confluent, each term rewrites in at most one irreducible term $\mu[t]$ which is called its *normal form*. The function μ is, in general a partial function.

A rewrite system is noetherian or has the finite termination if for each term t, there exists no infinite sequence such that $t_0 \to t_1 \to \cdots t_i \to t_{i+1} \to \cdots$

A rewrite system is *complete*, if it is confluent and noetherian. In case of confusion with another concepts of completeness we will say confluent-noetherian. Musser [Mus78] proposes the term *convergent*.

If a system is complete, we can associate one normal form with each term. Hence the function μ is total.

1.2 Confluence in the case of Abstract Data Types

The Knuth-Bendix algorithm [KB70, Hue77] tests the confluence of a rewrite system. It proceeds as follows: g and g' are supposed to have distinct variables, a term g is superposable to g' if there exists a sub-term s of g' (no reduced to variable) and a substitution σ of g such that $\sigma g = \sigma s$. The term $\sigma g'$ is called the result of the superposition.

A rewrite system is *locally confluent* if $t \to t_1$ and $t \to t_2$ imply that there exist t' such that $t_1 \stackrel{*}{\to} t'$ and $t_2 \stackrel{*}{\to} t'$.

Knuth and Bendix show that it is enough in order to test the confluence of a noetherian system to test the local confluence on the left hand sides $\sigma g'$ of rules which result of a superposition, more precisely in the case the substitution σ is "the most general", called *unifier* (cf Chapter VI).

From experience, we noticed the following phenomenon:

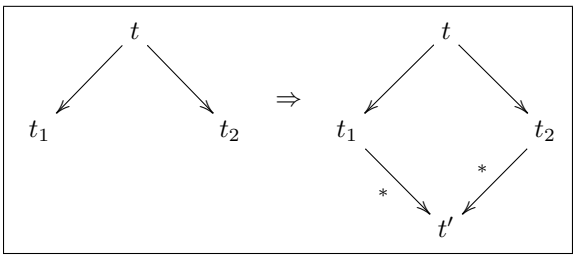


Figure II.2: Local confluence property

In an abstract data type, the left-hand side of rewrite rules are generally not superposable.

Here is the explanation of this phenomenon. The operators of signature $... \to Ti$ can be partitioned into two families.

- the *generators*, i.e., those which appear in normal forms.
- the *secondary operators*, i.e., those which do not appear in normal forms. We call them *destructors*. More precisely a destructor has a signature $(Ti, Ext_{i_1}, ..., Ext_{i_n}) \to Ti$.

and

$$|\mu(f(x, y_1, ..., y_n))| < |\mu(x)| \qquad \text{for every expression } x.$$

$|\_|$ is the length.

A *selector* is an operator of signature $... \to Ext_i$.

Most of the rules of an abstract data type have the form $H(t_1, ..., t_n) \to t$ where $t_1, ..., t_n$ are secondary operators. Thus if there is a superposition of a left-hand side $H'(t'_1, ..., t'_n)$ on $H(t_1, ..., t_n)$, it cannot appear in a strict sub-term since these sub-terms do not contain secondary operators.

Thus $H' = H$ and there exists a substitution σ such that

$$\sigma t_1 = \sigma t'_1, ..., \sigma t_n = \sigma t'_n.$$

Therefore, the term $H(\sigma t_1, ..., \sigma t_n)$ rewrites into two ways σt and $\sigma t'$. This ambiguity is often embarrassing in a specification: this is why we avoid it as most as possible. This leads to avoid all cases of superposition. We could even imagine an automatic system which detects the superpositions, let the specifier know about them and possibly tells him whether there is local confluence[1].

Example 1. *Take again the example of File of Section 5.1.*[2] *Assume the specifier writes the rules:*

(CONCAT1) CONCAT1(f, FILEVIDE) = f

[1] *Note at translation:* Four years later, I created such a system called Reve [Les83].

[2] *Note at translation:* There is a mistake in the original, because this is actually Example 6.1 of Chapter I.

then

(CONCAT2') CONCAT (f, AJ(f', a)) = CONCAT(AJ(f,FR(AJ(f',a))),OT(AJ(f,f')))

then finds out another rule:

(CONCAT2) CONCAT(f, AJ(f',a)) = AJ(CONCAT(f,f'), a)

The second and the third rules superpose trivially, the rule (CONCAT2') which is a little too complicated can be removed. A reason will occur in Example **??**.

Notice that those two rules do no reduce so obviously one to the other, since the proof of their equivalence is not by rewriting, but requires induction.

1.3 Termination in the case of Abstract Data Types

The finite termination, which is subtle to proof in the most general algebraic systems, is often easy to test in "well built" abstract data types. For that, we use a criterion proposed by Musser [Mus78].

Definition 1. *Let F and H be two operators, a rule $H(t_1, ..., t_n) \to t$* reduces *F if it exists $i \in \{1...n\}$ and terms $s_1, ..., s_n$ such that $t_i \in F(s_1, ..., s_p)$ and all occurrences of H in t have the form $H(s_i, ...s_n)$ where s_i is one of the s_j's for $j \in [1..p]$ (Figure II.3).*

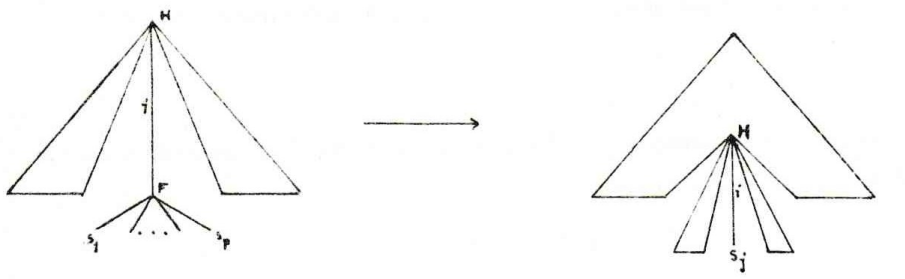


Figure II.3: Reduction scheme of F

Translation under work.

Bibliography

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[Ars78] Jacques Arsac. *La construction des programmes structurés*. Dunod, Paris, 1978.

[AW76] E. A. Ashcroft and W. W. Wadge. Lucid - a formal system for writing and proving programs. *SIAM Journal of Computing*, 5(3), 1976.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[Bel78] Bellegarde et al. MEDEE a type of language for constructing programs. In *Wokshop on reliable software*, Bonn (Germany), 1978.

[BW79] F. H. Bauer and H. Wössner. *Algorithmic language and program development*. Prentice Hall, Inc., London, 1979.

[CCD+79] J. Chabrier, J. J. Chabrier, J. C. Derniame, P Henry, R. Minot, and C. Proch. Le langage ATM: manuel d'utilisation. CRIN (à paraître), 1979.

[dBdR72] J. W. de Bakker and Willem P. de Roever. A calculus for recursive program schemes. In *ICALP*, pages 167–196, 1972.

[dR74] W. P. de Roever. Operation, mathematical and axiomatized semantics for recursive procedures and data structures. Technical report, Mathematical Center, Amsterdam, 1974.

[Fin77] Jean-Pierre Finance. Data structures as a framework to formalize the semantics of a programming language. In B. Robinet, editor, *Comptes-Rendus du 2^e Colloque sur la Programmation*, pages 75–88, Paris, 1977. Dunod.

[FVV78] Jean Françon, Gérard Viennot, and Jean Vuillemin. Description and analysis of an efficient priority queue representation. In *19th*

Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978, pages 1–7. IEEE Computer Society, 1978.

[GP77] Marie-Claude Gaudel and Claude Pair. Les structures de deonnées et leur représentation en mémoire. Technical report, IRIA, 1977.

[GTWW75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B Wright. Abstract data type as initial algebras and correctness of data representations. In *Proceeding of Conference on Computer Grapics, Pattern Recognition and Data Structures*, May 1975.

[Gut77] John V. Guttag. Abstract data type and the development of data structures. *Commun. ACM*, 20(6):396–404, 1977.

[Gut78] John V. Guttag. Notes on type abstraction. In Friedrich L. Bauer and Manfred Broy, editors, *Program Construction, International Summer School, July 26 - August 6, 1978, Marktoberdorf, Germany*, volume 69 of *Lecture Notes in Computer Science*, pages 593–616. Springer, 1978.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[Hue77] Gérard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977* [IEE77], pages 30–45.

[IEE77] IEEE Computer Society. *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, 1977.

[KB70] Donald E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[Les83] P. Lescanne. Computer experiments with the REVE term rewriting systems generator. In *Proceedings of 10th ACM Symposium on Principles of Programming Languages*, pages 99–108. ACM, 1983.

[Liv78] C. Livercy. *Théorie des programmes*. Dunod, Paris, 1978. Livercy stands for Jean-Pierre Finance and Monique Grandbastien and Pierre Lescanne and Pierre Marchand and Roger Mohr and Alain Quéré and Jean-Luc Rémy. Available at `http://perso.ens-lyon.fr/pierre.lescanne/publications.html`.

[LS77] Daniel J. Lehmann and Michael B. Smyth. Data types (extended abstract). In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977* [IEE77], pages 7–12.

[LZ74] Barbara Liskov and Stephen N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[LZ77] B. Liskov and S. Zilles. *Current trends in Programming Methodology*, chapter An Introduction to formal specification of Data Abstraction, pages 1–32. Prentice Hall, 1977. R. Yeh ed.

[Mor73] James H. Morris. Types are not sets. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 120–124. ACM Press, 1973.

[Mus78] D. Musser. Convergent sets of rewrite rules for abstract data types. Technical report, USC Information Sciences Institute, 1978.

[Pai74] Claude Pair. Formalization of the notions of data, information and information structure. In J. W. Klimbie and K. L. Koffeman, editors, *Data Base Management, Proceeding of the IFIP Working Conference Data Base Management, Cargèse, Corsica, France, April 1-5, 1974.*, pages 149–168. North-Holland, 1974.

[Pai79] Claude Pair. La construction des programmes. *Reveu dAutomatique, d'Informatique et de Recherche Opérationnelle*, 2:113–137, 1979.

[Pie68] R. S. Pierce. *Introduction to the theory of Abstract Algebras.* Holt, Rinehart and Holt, 1968.

[RV80] J.-C. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *Journal of the ACM*, 27(4):772–796, 1980.

[Ré74] Jean-Luc Rémy. *Structure d'information et notion d'accès et de modification d'une donnée.* PhD thesis, Université de Nancy I, 1974.

[WLS76] William A. Wulf, Ralph L. London, and Mary Shaw. An introduction to the construction and verification of alphard programs. *IEEE Trans. Software Eng.*, 2(4):253–265, 1976.