ALGEBRAS, THEORIES AND FREENESS: AN INTRODUCTION FOR COMPUTER SCIENTISTS

R.M. Burstall and J.A. Goguen

Edinburgh University/Xerox PARC and SRI International

INTRODUCTION

In the last ten years or so a lot of algebraic ideas have wormed their way into Computer Science, particularly in work connected with correctness of compilers, with abstract data types and with specification. We have been among those responsible [Burstall and Goguen 1980, 1981]. Most papers begin with a compressed section of definitions, but it is difficult for the well-disposed outsider to make much of these. Reference to books for algebraists, such as Graetzer [1979], or even to those angled towards automata theory [Arbib and Manes 1975] may not be encouraging. So it is perhaps worthwhile to present some of the key algebraic ideas in a leisurely and, we hope, intuitive form, emphasising the Computer Science connection. Some water has flowed under the bridge since one of us was last involved in an attempt to do this [Goguen, Thatcher, Wagner and Wright 1975].

In general we would claim that algebraic and categorial ideas can be a help in understanding computational phenomena. There are of course pitfalls, and we would claim to have fallen into most of them. First you can be overly abstract and fail to understand what you are talking about (if anything). Second you can make the obvious seem impressive and waste everybody's time (including your own). You can impose a lot of unfamiliar general concepts on people without giving adequate reward for their labours in the shape of computational insight. *Caveat lector.*

On the other hand there are advantages .....

M. Broy and G. Schmidt (eds.), Theoretical Foundations of Programming Methodology, 329–349.

The same concepts often come up again and again so that it is worth defining them once and for all, for example 'a collection of sets with some functions defined on them'. Such concepts include 'algebra' and 'homomorphism'. We can learn to think in terms of these concepts instead of in terms of their components such as 'set' or 'n-tuple'; our thoughts can then take bigger strides.

Not only does the same concept come up repeatedly but we often see a family of related concepts. For example the same construction may be carried through using 'sets', 'indexed sets', 'partially ordered sets' and 'sets with error elements'. So it is worth while to treat the general case. (It is even more worth while to treat the simplest special case first so that people can understand what you are talking about - *mea culpa*.) In these matters the general case can often be described using the language of category theory, which is particularly well-adapted for talking about the structure of systems without saying exactly what the components are. As a matter of fact we do that all the time in informal Computer Science. We talk happily about while statements of the form '**while** expression do **statement**' without worrying exactly what the expressions and statements are (Do they permit upper and lower case identifiers? Arrays with several subscripts? Mostly we don't care.) But most mathematical treatments of programming languages are very bottom-up and describe these matters in boring detail, or they give a toy language and leave the generalisations to the reader's intuition (not a bad idea, but it is nice to know how to be more precise). So in a way the more abstract categorial concepts correspond more closely to our informal ways of talking.

Although this stuff, particularly the categorial concepts may seem hard to get the hang of, there is hope: we can bring to bear our computing intuition by programming up all the ideas in the mathematics books. One of us has a go at this for some algebraic and categorial concepts [Burstall 1980], with the able assistance of David Rydeheard and Don Sannella. It turns out that many categorial proofs are constructive and hence can give rise to programs; the difficulty in reading the books comes partly from the fact that the resulting programs are very elegant and have little redundancy, they get maximum mileage from a very small number of general procedures.

This paper introduces the ideas of signature, algebra and theory, also initial algebra and free algebra. Our style of treatment motivates, but does not indulge in, the more abstract approach of category theory.

SIGNATURES

We start with the notion of signature. This corresponds in programming terms to a bunch of declarations, declaring types, constants and procedures.

```
integer, bool, tree: type

zero, one: integer
plus, minus, times: function(integer, integer) result integer
true, false: bool
nil: tree
tip: function (integer) result tree
node: function(tree, tree) result tree
isnil, istip: function(tree) result bool
sum: function(tree) result integer
```

Notice that we just name the types; we have not said what they consist of (records, arrays or whatever). Nor do we give the bodies of the procedures. Now we can simplify things a little by regarding the constants such as zero and one as functions with no arguments. Also we will follow mathematical practice and use the word "sort" rather than "type" and "operator" rather than "function name". Thus a signature consists of a set of sorts and a set of operators, each with given input and output sorts, for example

```
< {integer, bool, tree},
{   zero, one: →integer,  plus, minus, times: integer × integer →integer,
    nil: →tree,  tip: integer → tree,  node: tree × tree → tree,
    isnil, istip: tree → bool,  sum: tree → integer } >
```

We may think of the second element of this pair as defining for each possible sequence of input sorts and output sort such as <<tree,tree>,tree>, the set of operators having those sorts. More formally we need the notion of indexed family of sets. By an *I-indexed family of sets* we mean a function from I to sets. Let Nat mean {0,1,2,...}. Then for example the function mapping 0 to {a,b}, 1 to {a,c,d}, 2 to the empty set and so on is a Nat-indexed family of sets of letters, written more briefly "$\{a,b\}_0$, $\{a,c,d\}_1$, $\emptyset_2$, ...".

Now in a signature the operations are a family of sets indexed by their "arity", that is their sequence of input sorts and their output sort.

Defn. A **signature**, Σ, is a pair $\langle S, \Omega \rangle$, where S is a set (of sorts) and Ω is a family of sets (of operators) indexed by $S^* \times S$.

(Above, for example, the element <integer integer,integer> is the index of the set {plus,minus,times} in Ω).
To save writing we will assume Σ is <S,Ω> whenever it is mentioned.

ALGEBRAS

Now signatures are not much fun unless we give some meaning to the names. They are syntax without semantics. If we give these meanings we get an algebra. In a programming language we might say (assuming types integer and bool already defined)

```
representation tree is sexpression

nil: tree = "NIL"
tip: function(n: integer) result tree = cons(n, nil)
node: function(t1,t2: tree) result tree = cons(t1, t2)
isnil: function(t: tree) result bool = null(t)
istip: function(t: tree) result bool = null(cdr(t))
sum: function(t: tree) result integer =  if null(t) then 0 else..
```

This associates some particular set with the sort tree, namely the set of Lisp sexpressions, and some particular function with each operator taking arguments of the appropriate sorts and producing an appropriate result. That is an algebra is a signature together with a function taking sorts to sets and another function taking operators to functions.

A simple example would be the algebra of numbers modulo 4, call it Mod4, with signature

< {number}, {zero,one: → number, plus,times: number × number → number}>

which can be written

number {0, 1, 2, 3}

zero 0

one 1

plus

	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

times

	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

The algebra of trees has infinite sets for the sorts integer and tree and the tables for the operations are correspondingly rather too big for our page, but the principle is the same.

We call the family of sets associated with the sorts the *carrier* of the algebra, writing $|A|$ for the carrier of algebra A.

Defn. Let Σ be a signature. A **Σ-algebra** A is an S-indexed family of sets, $|A|$, called the carrier of A, together with an $S^* \times S$-indexed family of functions $\alpha_{us} : \Omega_{us} \to (|A|_u \to |A|_s)$ where $u \in S^*$, $s \in S$ and $|A|_{u1\ldots un} = |A|_{u1} \times \ldots \times |A|_{un}$.

Notation If $\omega \in \Omega_{us}$ and $\langle a_1,\ldots,a_n \rangle \in A_u$, we write $\omega(a_1,\ldots,a_n)$ for $\alpha_{us}(\omega)(a_1,\ldots,a_n)$ where there is no ambiguity.

HOMOMORPHISMS

Now we can, at least informally, understand the idea of evaluating expressions in an algebra. For example "plus(one,one)" in the above algebra gives 2 and hence "times(plus(one,one),one)" also gives 2. We might have two algebras which are rather similar in that evaluating in one gives analogous results to evaluating in the other. Suppose there is a function f from the carrier of the first algebra to that of the second (or if there are several sorts such a function for each sort) with the property that if we evaluate an expression in the first algebra and get answer a, then evaluating it in the second algebra will always give answer f(a). More generally we might ask that if in the first algebra evaluating an expression like plus(x,times(x,y)) with variable x bound to a and y bound to b gives result c, then evaluating the same expression in the second algebra with the corresponding binding, x to f(a) and y to f(b), should give the corresponding result f(c). In this case we say that f is a *homomorphism* from the first algebra to the second one.

For example if we define another algebra, say Mod2, that of numbers modulo 2, with {0,1} as the set of numbers and the obvious multiplication, then there is a homomorphism f from Mod4 to Mod2, with

f(0) = 0 f(1) = 1 f(2) = 0 f(3) = 1

(Try evaluating some expressions in both these algebras.) We write

f:Mod4 → Mod2

Now consider a "parity" algebra with {even,odd} as the set of numbers and the obvious tables for plus and times, that is plus(even,even) = even and so on. Then there is a homomorphism from the algebra of natural numbers, {0,1,2,3,...} with the usual meaning of plus and times, to Parity.

As a matter of fact all expressions with corresponding bindings give corresponding results if and only if each operator given corresponding arguments gives corresponding results, that is in the example above

f(zero) in Mod4 = zero in Mod2
f(one) in Mod4 = one in Mod2
f(plus(m,n)) in Mod4 = plus(f(m), f(n)) in Mod2
f(times(m,n)) in Mod4 = times(f(m), f(n)) in Mod2

We may as well adopt this property of operators as our official definition of homomorphism, thus

Defn. If $\Sigma = \langle S,\Omega\rangle$ then a Σ-homomorphism from a Σ-algebra $\langle A,\alpha\rangle$ to a Σ-algebra $\langle A',\alpha'\rangle$ is a map $f: |A| \to |A'|$ such that for each $\omega \in \Omega_{us}$ and each $a_1 \in A_{u1},\ldots,a_n \in A_{un}$, $f_s(\omega(a_1,\ldots,a_n)) = \omega(f_{u1}(a_1),\ldots,f_{un}(a_n))$.

The function f from Mod2 to Parity defined by f(0) = even and f(1) = odd is clearly a homomorphism; so is its inverse g, with g(even) = 0 and g(odd) = 1. We say that f and g constitute an *isomorphism* between Mod2 and Parity. Thus an isomorphism between algebras A and B is a pair of inverse homomorphisms, $f: A \to B$ and $g: B \to A$ such that $f \bullet g = 1_A$ and $g \bullet f = 1_B$. This is an important idea, because when we think abstractly about some notion of number we do not care what algebra we are talking about to within an isomorphism. Roman numbers are the same mathematically as Arabic numbers. When we speak of abstract data types we mean that we do not care about the particular set of elements, we are just talking about some algebra to within an isomorphism. We consider two algebras to be "abstractly the same" if they are isomorphic.

WORD ALGEBRAS AND INITIAL ALGEBRAS

But for any signature there is a particularly interesting algebra called the *word algebra*. Its elements for a given sort are all the expressions in the signature denoting elements of that sort. For our number signature the set of "numbers" of

the word algebra would be "zero", "one", "plus(zero,zero)", "plus(zero,one)" etc. What are the operations? What should "times" do when given "zero" and "plus(zero,one)"? It should produce "times(zero,plus(zero,one))" of course. The operations simply build larger expressions from smaller ones. To be more definite we may think of the expressions as character strings and the operations as string manipulators, for example, using <> for string concatenation,

```
times: procedure (s1,s2: string) result string =
                    "times(" <> s1 <> "," <> s2 <> ")"
```

Alternatively we could think of the expressions as written in Reverse Polish or represented by trees, using records, or whatever. It doesn't matter because all these expression algebras will be isomorphic. In fact when we speak of "the" word algebra we mean any of these isomorphic algebras.

Is there any more "abstract" way to characterise this algebra (to within an isomorphism) instead of picking a particular representation? The key idea is that given any interpretation of the operators we get a unique value for each expression. But an interpretation of the operators is just another Σ-algebra, A, and the evaluation of the expressions is just a homomorphism from the word algebra to A. Indeed this is the only homomorphism to A. If we know the interpretation of each operator we can work out inductively a value for each term. We call the Σ-algebra having this *unique homomorphism property* the *initial Σ-algebra*.

More formally

Defn. I is an initial Σ-algebra iff for any Σ-algebra, A, there is a unique homomorphism $f: I \to A$.

Making an abstract definition by using a unique homomorphism property, also called a *universal property*, is an important device (used extensively in category theory). Let us try to see what it means. Consider the signature

```
sorts n
opns z:n      s:n → n
```

What is the initial algebra for this signature? Algebras of the signature include

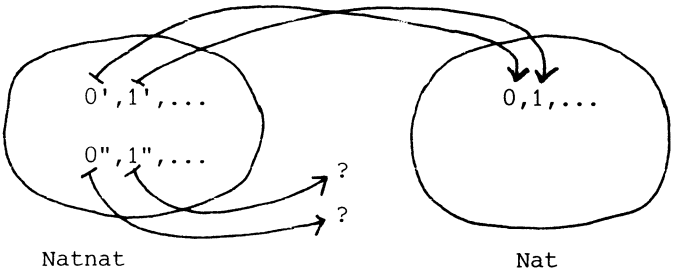
```
Nat    |Nat| = {0,1,2,...}      z = 0
                                 s(x) = x+1

Mod2   |Mod2| = {0,1}           z = 0
                                 s(0) = 1, s(1) = 0
```

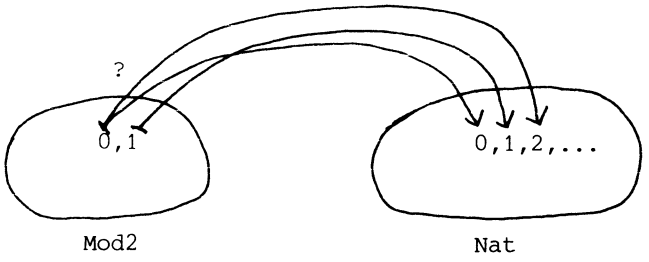
Natnat $|Natnat| = \{0',1',2',\ldots 0'',1'',2'',\ldots\}$

$z = 0'$
$s(0') = 1',\ s(1') = 2',\ldots$
$s(0'') = 1'',\ s(1'') = 2'',\ldots$

Now if we try Natnat as the initial algebra we hit trouble. How many homomorphisms are there from Natnat to, say, Nat? Well, z in Natnat must go to z in Nat so 0' goes to 0, and also by the homomorphism property s(0') goes to (s(0), that is 1' goes to 1. But where does 0" go? It could go to 0 or 1 or 2 or anywhere. 0' is there to give a value to z, 1' to give a value to s(z) and so on. But 0" is just junk, a useless piece of baggage. It need not be there and has no idea what to do with itself in a homomorphism. So there are many homomorphisms from Natnat, and it cannot be the initial algebra. No junk, please.



Let us try Mod2 for initial algebra, looking for homomorphisms to, say, Nat. Since z goes to z,0 in Mod2 goes to 0 in Nat, s(0) goes to s(0) so 1 goes to 1, s(1) goes to s(1) so 0 goes to 2. But we already had 0 goes to 1! 0 is schizophrenic. There is no homomorphism from Mod2. In fact 0 in Mod2 is serving as the value of both z and of s(s(2)), introducing this confusion. Mod2 is not the initial algebra. No confusion, please.



It all ends happily. Nat is the initial algebra because it has

no junk: every element of the carrier is the value of some term

no confusion: different terms get different values.

Thus the unique homomorphism property is a neat way of insisting on these intuitive properties.

Fact. If I and I' are both initial Σ-algebras then they are isomorphic.

Proof. Since I is an initial Σ-algebra we have a homomorphism $f: I \to I'$, and similarly one $f': I' \to I$. Now the composition $f \circ f': I \to I$ is a homomorphism. But so is $1_I: I \to I$, the identity, so by uniqueness $f \circ f' = 1_I$. Similarly $f' \circ f = 1_{I'}$. So f and f' are inverses, and I and I' are isomorphic. □

It is not difficult to show by induction that the word algebra is initial.

The fact that there is just one homomorphism from the word algebra to any other algebra is a way of stating the familiar property of expressions that there is just one way of creating a given expression using the operators described. Thus "times(zero,plus(zero,one))" can only be created by applying the operator times to the two expressions "zero" and "plus(zero,one)". That is the operators all have unique inverses.

An example of the application of initial algebras in describing programming concepts is so-called *Abstract Syntax*, McCarthy [*circa* 1776]. By the abstract syntax of a language we mean a collection of rules showing how to construct all phrases of the language in the form of trees rather than character strings (for example S-expressions are the trees generated by the abstract syntax of LISP). Now the abstract syntax of a language is defined by a signature whose sorts are the various syntax classes, e.g. variable, constant, expression, statement, declaration and block. The operators correspond to the rules for building elements of these classes from their components, thus we might have

```
x,y: variable
0,1: constant
vexpr: variable → expression
cexpr: constant → expression
sumexpr: expression × expression → expression
ifthen: expression × statement × statement → statement
assignment: variable × expression → statement
```

Now the initial algebra is just the set of elements of the syntax classes (think of them as trees) together with the operations for constructing them. Because we have defined

this algebra abstractly, upto isomorphism only, we have not specified any particular concrete syntax. Furthermore a compiler is essentially a homomorphic map from the initial algebra to strings of machine code instructions with suitable combining operations (not quite because this does not take account of environments produced by declarations). For further discussion of this *Initial Algebra Semantics* see Goguen, Thatcher, Wagner and Wright [1977].

In closing we should mention the notion of *final algebra* ('dual' to initial) which has a unique homomorphism from any algebra to it. This is also of use in specification work.

FREE ALGEBRA ON A SET

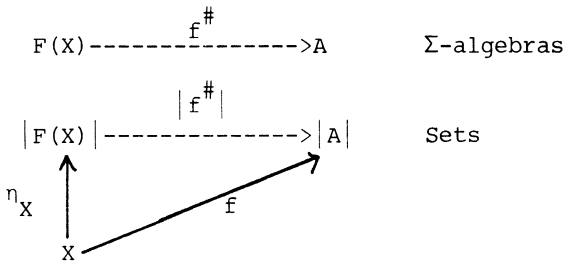
A slightly more general concept is the word algebra on a set X of variables of given sorts. The set of elements of this algebra is the set of expressions using these variables, for example as well as "plus(zero,zero)" we have "plus(x,zero)". The variables and expressions are of course associated with appropriate sorts. The operators are the expression constructors as before. Call this algebra $W_\Sigma(X)$. Again it enjoys a unique homomorphism property, and to avoid talking of particular representations we may use this property to define a more abstract concept, the free algebra on X.

Defn. By a Σ-algebra on an S-indexed set X we mean a Σ-algebra $F(X)$ with a function $\eta_X: X \to |F(X)|$, such that for any Σ-algebra A, any S-indexed family of functions $f: X \to |A|$ extends uniquely to a homomorphism $f^\#: F(X) \to A$. By $f^\#$ extends f we mean that $\eta_X ; |f^\#| = f$.

We will write F_Σ for F when we need to say what the signature is.

Note that we write $|f^\#|$ for the homomorphism $f^\#$ regarded as a function, a subtlety which can be ignored if you wish.

Not surprisingly we can show that the word algebra on X is a free algebra on X. Here η_X is the obvious function taking each variable x in X to the expression "x" in $|F(X)|$. (Compare the LISP function taking x to cons(x,nil)). Now A might be, for example, the usual arithmetic algebra and f is a binding of variables and $f^\#$ is the evaluation function giving a numerical value to each expression. Naturally $f^\#$("x") = f(x), i.e. $\eta_X ; |f^\#| = f$. This is expressed graphically by saying that the following diagram commutes



(We say that a diagram commutes if taking the composition of the functions on arrows along any path round a triangle or square or larger polygon gives the same result. Here the path $\eta_X \circ |f^\#|$ is equal to the path f. Similarly when we have homomorphisms instead of functions. We use dotted lines for arrows whose existence is being asserted.)

This diagram is mysterious; let us make friends with it. It really says that $f^\#$ is an extension of f, i.e. as far as X is concerned $f^\#$ is the same as f. But $f^\#$ is a homomorphism, so we have to regard it as just a function $|f^\#|$ before we can compare it with the function f. Now we would like to say that $|f^\#|$ restricted to X is f. That is all right if X is a subset of $|F(X)|$, but the free algebra F(X) is 'abstract'; we can make it with a lot of different carriers. So η_X just tells us which elements in the carrier stand for elements of X, and $\eta_X \bullet |f^\#|$ is the nearest thing we can truthfully say to '$f^\#$ restricted to X'. This is what we pay for the 'abstractness' of the free algebra notion.

We started with an informal notion of evaluating expressions for motivation. We now have a precise notion of expressions and their evaluation.

A remark on nomenclature: The names "word algebra" and "word algebra on X" apply strictly to the algebras we have described. As we shall see "initial" and "free" refer to more general categorial concepts of which these are particular examples.

THEORIES

Now we may be interested in the collection of algebras with a given signature which satisfy certain laws, such as associativity or commutativity. We can express these laws as equations. An equation in a signature over a set of variables is nothing more than the variables and a pair of expressions in those variables. For example the equation "plus(x,y) = plus(y,x)" over {x,y} can be thought of as the triple <{x,y}, plus(x,y), plus(y,x)>. Formally

Defn. A Σ- equation, e, is a triple <X,t1,t2> where X is an S-indexed set (of variables) and t1,t2 $\in |W_\Sigma(X)|$ are terms on X of the same sort. (The equation would normally be written "for all X, t1 = t2".)

Now we want to discriminate amongst algebras on the basis of some set of equations. What does it mean for an algebra to satisfy an equation, say "e1(x,y) = e2(x,y)". We understand it to mean that, whatever the values of x and y, if we evaluate the left hand expression in the algebra we get the same result as evaluating the right hand expression with those values. So if A is the algebra and f: {x,y} $\rightarrow |A|$ is a binding of the variables then we require that its extension to expresssions, $f^\#$, obey the $f^\#(e1(x,y)) = f^\#(e2(x,y))$. In general we have

Defn. A Σ-algebra A satisfies a Σ-equation <X,t1,t2> iff for all maps f: X $\rightarrow |A|$, $f^\#(t1) = f^\#(t2)$. We write A $\models$ e for A satisfies e.

Now let E be a set of equations. An algebra is a model of E if it satisfies each of the equations of E. We write E* for the set of all models of E.

Now given a set of equations which describes a set of models we may be interested in what else we can say about these models. Hence we need to speak of all the equations which are satisfied by every model in some set M, call this set of equations M*. (We use the same symbol * again in order to emphasise the duality.) Now E** is the set of all equations which describe the models of E; it is called the *closure* of E. Inference systems aim to generate exactly this set of logical consequences (if they do so they are called consistent and complete). For equations there is a very simple such inference system using reflexivity, symmetry and transitivity of equality plus the substitution properties. But we are concerned here with semantics and models rather than proof theory.

Consider as a very simple example the theory with one sort "person", three nullary operators (constants) and two equations

```
sorts   person
opns    holmes, moriarty, babbage: person
eqns    holmes  = moriarty
        morarity = babbage
```

Now we consider algebras with elements p1,p2,p3,..., say, and various values for the constants. Some of these algebras will satisfy the equations, those in which holmes and moriarty get the same value and moriarty and babbage get the same value; others will not. But if any algebra satisfies the given two

equations then it also satisfies the equation "holmes=babbage". So this equation is in the logical closure.

We call a signature with a set of equations a *presentation*. A set of equations is called *closed* if its closure is the set itself. We call a signature with a closed set of equations a *theory*. A theory may well consist of an infinite set of equations.

Defn. A **presentation** is a pair $\langle \Sigma, E \rangle$, where Σ is a signature and E is a set of equations.

Defn. A **theory** is a presentation $\langle \Sigma, E \rangle$, such that E is closed.

When we give a set of equations as above we are usually giving a presentation, but we mean this to denote its closure, a theory. So we will allow ourselves to say "the theory so-and-so" when "so-and-so" is actually a presentation.

Notice that presentations and theories are "syntactic" objects as opposed to algebras which are "semantic" objects. Remember that one theory can have many algebras which satisfy it.

ALGEBRAS FOR THEORIES

Now we can define notions analogous to "Initial Algebra" and "Free Algebra on X" but relative to all the algebras of a given theory. Consider the theory String defined by

```
sorts letter, string
opns  A,B, . . . ,Z: letter
      empty: string
      unit : letter → string
      _ . _: string, string → string
eqns  empty.s = s
      s.empty = s
      s.(t.u) = (s.t).u
```

What is an interesting algebra for string? Well of course A might be the same as C and s.s.s the same as s.s. Also the algebra might have as well as strings like unit(A).unit(B) other strings like πλατο and σοκρατεσ and λιττλεγρεεψμεν, but the theory did not say anything about them and the poetic imagination is discouraged in Computer Science education.

So consider the algebra which has no junk (elements are in the algebra only if they are the values of some term) and no confusion (terms denote equal elements only if the equations

force them to). This is what we call the initial algebra. A rather neater way to define it is to say that it is the algebra which has a unique homomorphism to every other algebra, where algebra now means algebra which is a model of the given theory.

This leads to

Defn. I is an **initial algebra** of a theory iff for any algebra of the theory, A, there is a unique homomorphism f: I → A.

An easy way to get the initial algebra of a theory is to take the word algebra on its signature and then form equivalence classes according to the equations of the theory. We have to ensure that if terms t1 and t2 are in the same equivalence class, then ω(t1) and ω(t2) are in the same class, for any unary operator ω, and analogously for operators of several arguments. Such an equivalence is called a *congruence*.

Now of course we can do the same thing with variables. Given a theory T, the free T-algebra on a set X of variables is the one in which any binding of the variables to elements of another algebra determines a unique homomorphism. It can be obtained by taking equivalence classes of elements in the Σ-word algebra on X where Σ is the signature of T. More formally

Defn. By a **free algebra on X** of a theory we mean an algebra F(X) of the theory with a function η_X: X → |F(X)|, such that for any algebra A of the theory, any function f: X → |A| extends uniquely to a homomorphism $f^\#$: F(X) → A. By $f^\#$ extends f we mean that $\eta_X \bullet |f^\#| = f$.

CHANGING THE SIGNATURE

If we are interested in specifications and their implementations we will be interested in changing signatures, mapping sorts in one signature into sorts in another and operators in one to operators in the other. For example given a signature Integer thus

sorts integer, boolean
opns zero, one: integer
plus, minus: integer, integer → integer
equal: integer, integer → boolean

and a signature Entier thus

```
sorts    entier, booleen
opns     zero, un: entier
         plus, moins: entier, entier → entier
         egal: entier, entier → booleen
```

we can define a function, Translate, thus

```
integer to entier    boolean to booleen
zero to zero         one to un
plus to plus         minus to moins    equal to egal
```

Note that this function preserves argument and result sorts, thus the input sort of the translation of "plus" is the translation of its input sort and so on. We call such a function a signature morphism. In general a signature morphism may go from one signature to another with extra sorts or operators, and the function need not be one to one, two sorts may map to the same sort in the second signature, and similarly for operators.

Signature morphisms arise in describing programming languages. Suppose that we have some notion of parameterised module and we want to talk about binding the formal types and procedures of the module to actual ones in the external environment. This binding is a signature morphism.

It is more elegant to think of a signature morphism as a pair of functions, one for sorts and one for operators. Formally we have

Defn. A **signature morphism** σ from a signature $\Sigma 1$, say $\langle S1,\Omega 1\rangle$, to a signature $\Sigma 2$, say $\langle S2,\Omega 2\rangle$, is a pair $\langle f,g\rangle$ consisting of a function $f: S1 \to S2$ and a family of functions $g_{us}: \Omega 1_{us} \to \Omega 2_{f^*(u)f(s)}$, where $f^*:S1^* \to S2^*$ is the extension of f to strings. We write $\sigma: \Sigma 1 \to \Sigma 2$.

It is convenient to write $\sigma(s)$ for $f(s)$, $\sigma(u)$ for $f^*(u)$ and $\sigma(\omega)$ for $g_{us}(\omega)$.

What happens to algebras when we change the signature? If we go from $\Sigma 1$ to $\Sigma 2$ does a $\Sigma 1$-algebra turn into an $\Sigma 2$-algebra? No. There is a general and mysterious principle that when you go from syntax to semantics things often go backwards; "contravariant" is the posh word for this. (As in "My husband is in a contravariant mood today.") Given a $\Sigma 2$-algebra and a signature morphism $\sigma: \Sigma 1 \to \Sigma 2$, we can create a $\Sigma 1$-algebra. We get the set for a sort s of the $\Sigma 1$-algebra by taking the set for the sort $\sigma(s)$ in the $\Sigma 2$-algebra, and we get the table for an operator ω of the $\Sigma 1$-algebra by taking the table for $\sigma(\omega)$ in the $\Sigma 2$-algebra. In our example above if we are given an algebra for

Entier we can form one for Integer thus: the set for integer is that for entier, the set for boolean is that for booleen, the table for minus is that for moins and so on. Easy. The contravariance is not too surprising if we reflect that given a Pascal machine and a translater from Ada to Pascal we have an Ada machine.

Given the morphism σ: $\Sigma 1 \to \Sigma 2$ and the $\Sigma 2$-algebra A, we call the resulting $\Sigma 1$-algebra $U_\sigma(a)$. That is U_σ is the function from the set of all $\Sigma 2$-algebras to the set of all $\Sigma 1$-algebras corresponding to the signature morphism σ: $\Sigma 1 \to \Sigma 2$.

Defn. If σ: $\Sigma 1 \to \Sigma 2$ is a signature morphism, and A is a $\Sigma 2$-algebra, say $\langle |A|, a \rangle$, then we define the function U_σ : $\Sigma 2$-algebras $\to$ $\Sigma 1$-algebras, by $U_\sigma(A) = A'$, where $|A'|_s = |A|_{\sigma(s)}$, and $\omega(a_1,\ldots,a_n)$ in A' is $\sigma(\omega)(a_1,\ldots,a_n)$ in A.

A useful special case is where σ is an inclusion morphism, that is the sorts of $\Sigma 1$ are a subset of the sorts of $\Sigma 2$, similarly for the operators, and $\sigma(s) = s$, $\sigma(\omega) = \omega$. For example the signature Letters may be included in the signature Strings-of-Letters, so we can get a Letters algebra from a Strings-of-Letters algebra by just forgetting the extra sorts and operators. If we have an inclusion morphism from $\Sigma 1$ to $\Sigma 2$ we call $\Sigma 2$ an *enrichment* of $\Sigma 1$, and talk of the $\Sigma 1$-*reduct* of an $\Sigma 2$-algebra. In this case we write $\Sigma 1 \subseteq \Sigma 2$.

THEORY MORPHISMS

Let us try something a little more ambitious. Given two theories T1 and T2, what would be a reasonable notion of mapping from T1 to T2? Suppose that they have signatures $\Sigma 1$ and $\Sigma 2$ respectively. Then we may consider a signature morphism, σ, from $\Sigma 1$ to $\Sigma 2$. Given an algebra of T2 we can apply U_σ to it. We get a new algebra with signature $\Sigma 1$ but is it a T1 algebra? Not necessarily. In fact we shall say that σ is a theory morphism from T1 to T2 just if U_σ always gives us a T1 algebra. Formally

Defn. If T1 and T2 are theories, say $\langle \Sigma 1, E1 \rangle$ and $\langle \Sigma 2, E2 \rangle$, by a **theory morphism** σ: T1 $\to$ T2 we mean a signature morphism, σ: $\Sigma 1 \to \Sigma 2$, such that for each T2-algebra A, $U_\sigma(A)$ is a T1-algebra.

Now since σ translates $\Sigma 1$ to $\Sigma 2$ we can use it to translate $\Sigma 1$ terms to $\Sigma 2$-terms in the obvious way, so we can translate a T1-equation to an equation in signature $\Sigma 2$. Now an important fact is that if σ is a theory morphism then this translation is a T2-equation. (Indeed we previously took this as the definition of theory morphism [Burstall and Goguen 80], but following Reichel [1980] we have learned to prefer the semantic definition.)

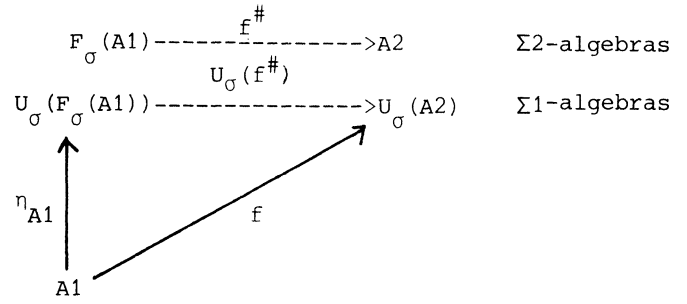
Just as we talked about signature inclusions, so we can talk about theory inclusions, where the signature of the first theory is included in that of the second one and the inclusion is indeed a theory morphism. Note that the equations of the second theory then include those of the first theory. For example the theory of monoids, with identity and associative multiplication is included in the theory of groups, which has an inverse operation also.

Notice that we can not only reduce T2-algebras to T1-algebras using σ, but we can reduce T2-homomorphisms to T1-homomorphisms. We simply relabel the maps using the T1-sorts and forgetting any which are not needed. Thus we can define a function, also called U_σ, from T2-homomorphisms to T1-homomorphisms. If $f: A2 \to B2$ is a T2-homomorphism $U_\sigma(f): U_\sigma(A2) \to U_\sigma(B2)$ is a T1-homomorphism.

ENRICHMENTS AND FREE ALGEBRAS

Suppose that signature $\Sigma 2$ is an enrichment of $\Sigma 1$ via the morphism $\sigma: \Sigma 1 \to \Sigma 2$. Given a $\Sigma 2$-algebra, A2, we can reduce it to a $\Sigma 1$ algebra, A1, namely $U_\sigma(A2)$. But can we do the reverse? Is there a 'best' or 'canonical' way of making a $\Sigma 2$-algebra from a $\Sigma 1$-algebra, A1? If σ is just the inclusion $\Sigma 1 \subseteq \Sigma 2$, then we can make the free algebra $F_{\Sigma 2}(|A1|)$. More generally we have to convert the $\Sigma 1$-carrier $|A1|$ to a $\Sigma 2$-carrier by reindexing according to σ, then apply $F_{\Sigma 2}$, then take a congruence on this determined by the operations of A1. But we prefer to define this $\Sigma 2$-algebra by a unique homomorphism property.

Defn. If $\sigma: \Sigma 1 \to \Sigma 2$ is a signature morphism and A1 is a $\Sigma 1$-algebra, then by the **free $\Sigma 2$-algebra on A1**, with respect to σ, we mean a $\Sigma 2$-algebra $F_\sigma(A1)$ and a $\Sigma 1$-homomorphism $\eta_{A1}: A1 \to U_\sigma(F_\sigma(A1))$ such that, for any $\Sigma 2$-algebra A2 and any $\Sigma 1$-homomorphism $f: A1 \to U_\sigma(A2)$, there exists a unique $\Sigma 2$-homomorphism, $f^\#: F_\sigma(A1) \to A2$, such that $\eta_{A1} \circ U_\sigma(F^\#) = f$; that is the following diagram commutes



Now if $\sigma: T1 \to T2$ is a theory morphism we can define the **free T2-algebra on a T1-algebra A1**, in just the same way,

replacing $\Sigma 1$ by T1 and $\Sigma 2$ by T2 in the above definition. The construction is similar too, forming the free T2 algebra on the re-indexed carrier of A1.

Suppose for example that T1 has the signature zero, successor, plus, times with appropriate equations and T2 has a larger signature which also has the string forming operators empty, unitstring and concatenation with equations for identity and associativity. Then for any T1-algebra, say the numbers modulo 3, there is a free extension to a T2-algebra, strings of numbers modulo 3. Note that even though the T1 algebra satisfies extra equations, such as 0 = succ(succ(succ(0))) the new operators get interpreted freely (no junk, no confusion) as string constructors.

This free extension of a theory, gives us a precise algebraic interpretation of the computer scientist's notion of parameterised data type, "string of X". T1 can be used to tell us the *requirements* on the parameter and is usually a subtheory of T2. For strings T1 could be trivial, the theory with one sort and no operators; for more complex data types it could have operators such as a partial order. For a further explanation see Thatcher, Wagner and Wright [1978] or Burstall and Goguen [1981].

Given a T1-homomorphism $f: X \to U_\sigma(Y)$ we get a T2-homomorphism $f^\#: F(X) \to Y$. Without any extra work we can turn a T1-homomorphism $g: X \to X'$ into a T2-homomorphism $F_\sigma(g): F_\sigma(X) \to F_\sigma(X')$. Just form $g \cdot \eta_{X'}: X \to U_\sigma(F_\sigma(X'))$ and put $F_\sigma(g) = (g \cdot \eta_{X'})^\#$. Thus F like U works on both algebras and homomorphisms.

The following table may help to summarise what we have done

Algebras on a signature	Algebras on a theory
Initial Σ-algebra	Initial T-algebra
Free Σ-algebra on an indexed set	Free T-algebra on an indexed set
Free $\Sigma 2$-algebra on a $\Sigma 1$-algebra, w.r.t. $\sigma: \Sigma 1 \to \Sigma 2$	Free T2-algebra on a T1-algebra, w.r.t. σ: T1 $\to$ T2

FREENESS IN GENERAL: THE CATEGORIAL APPROACH

The reader will have noticed the similarity between our definitions of

Free Σ-algebra on a set
Free T-algebra on a set
Free Σ2-algebra on a Σ1-algebra
Free T2-algebra on a T1-algebra.

They all use the same diagram, although sometimes we write $|...|$ and sometimes U_σ. What is the general definition of freeness? The answer to this and similar questions is the concern of category theory. It deals with arbitrary structures (objects) and structure-preserving functions between them (morphisms). It treats objects (sets, algebras, signatures, theories, etc.) and morphisms (functions, homomorphisms, signature morphisms, theory morphisms, etc.) as 'abstract data types' by concerning itself not with what they are made of ('frogs and snails and puppy dogs' tails'?) but with the operations which may be performed on them, notably composition of morphisms. For a readable introduction to category theory see Arbib and Manes [1975] or the first few chapters of **Goldblatt** [1979].

ACKNOWLEDGEMENTS

RMB would like to thank Peter Landin for educating him in the elements of algebra. We would like to thank the ADJ group (Jim Thatcher, Eric Wagner and Jesse Wright) for much helpful collaboration and our colleagues at Edinburgh and Los Angeles for many algebraic conversations. RMB would like to thank Xerox PARC, where he has been Visiting Scientist, for their support and for their document preparation facilities for a draft (*per ardua ad Alto*), and Eleanor Kerse for typing the final version.

REFERENCES

Arbib, M.A. and Manes, E.G. *Arrows, Structures and Functors: the categorical imperative.* Academic Press: New York, London, 1975.

Burstall, R.M. *Electronic category theory*, in: Proc. Symp. in Math. Foundations of Comp. Science, Rydzyna. Lecture Notes in Computer Science No. 88, 22-40, Springer-Verlag, 1980.

Burstall, R.M. and Goguen, J.A. *The semantics of Clear, a specification language*, in: Proc. of Advanced Course on Software Specifications. Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1980.

Burstall, R.M. and Goguen, J.A. *An informal introduction to specifications using Clear*, in: The Correctness Problem in Computer Science, Academic Press (to appear), 1981.

Goguen, J.A., Thatcher, J., Wagner, E. and Wright, J.B. *An introduction to categories, algebraic theories and algebras.* IBM Technical Report RC 5369, Thos. J. Watson Research Center, Yorktown Heights, N.J., 1975.

Goguen, J.A., Thatcher, J., Wagner, E. and Wright, J.B. *Initial algebra semantics and continuous algebras.* JACM, 24, 1, 68-95, 1977.

Goguen, J.A., Thatcher, J., Wagner, E. and Wright, J.B. *An initial algebra approach to the specification, correctness and implementation of abstract data types*, in: Current Trends in Programming Methodology (ed. R. Yeh), Prentice Hall, N.J., 1978.

Graetzer, G. *Universal Algebra, (Second Edition).* New York: Springer-Verlag, 1979.

Goldblatt, R. *Topoi, the categorial analysis of logic.* Amsterdam: North-Holland, 1979.

Reichel, H. *Initially restricting algebraic theories*, in: Proc. Symp. in Math. Foundations of Comp. Science, Rydzyna. Lecture Notes in Computer Science No. 88, 504-514, Springer-Verlag, 1980.

Thatcher, J.W., Wagner, E.G. and Wright, J. *Data type specification: parameterisation and the power of specification techniques*, in: Proc. of the 10th Annual Symposium on Theory of Computing, ACM, 119-132, 1978.

Theme VIII.

(R.M. Burstall)

A presentation of the Edinburgh approach, from several different angles