APPLICATIONS OF CATEGORY THEORY TO PROGRAMMING AND PROGRAM SPECIFICATION

DAVID ERIC RYDEHEARD

Ph.D. Thesis.

University of Edinburgh.

1981.

ABSTRACT

Category theory is proving a useful tool in programming and program specification - not only as a descriptive language but as directly applicable to programming and specification tasks.

Category theory achieves a level of generality of description at which computation is still possible. We show that theorems from category theory often have constructive proofs in the sense that they may be encoded as programs. In particular we look at the computation of colimits in categories showing that general theorems give rise to routines which considerably simplify the rather awkward computation of colimits.

The general routines arising from categorical constructions can be used to build programs in the 'combinatorial' style of programming. We show this with an example - a program to implement the semantics of a specification language. More importantly, the intimate relationship between these routines and algebraic specifications allows us to develop programs from certain forms of specifications.

Later we turn to algebraic specifications themselves and look at properties of "monadic theories". We establish that, under suitable conditions:

1. Signatures and presentations may be defined for monadic theories and free theories on a signature may be constructed.

2. Theory morphisms give rise to adjunctions between categories of algebras and moreover a collection of algebras of a theory give rise to a new theory with certain properties.

3. Finite colimits and certain factorisations exist in categories of monadic theories.

4. Many-sorted, order-sorted and even category-sorted theories may be handled by somewhat extending the notion of monadic theories.

These results show that monadic theories are sufficiently well-behaved to be used in the semantics of algebraic specification languages. Some of the constructions can be encoded as programs by the techniques mentioned above.

CONTENTS.

INTRODUCTION

Programming languages and specification languages together with programs and specifications are formal objects which have a logical structure. It is therefore no surprise that our attempt to understand and handle these things should make recourse to mathematics to provide a formal framework. What may be surprising to many who are familiar with computers and computer programming is that a branch of mathematical thought as abstract (and apparently as abstruse) as that known as the theory of categories can have any relevance to our understanding of programming.

Pulling a book on category theory from a library shelf and browsing through its pages may well make one baulk at not only its highly abstract contents but also the type of mathematics involved. Category theory grew out of investigations in topology and yet is essentially just a theory of functions. We hope that this work, if nothing more, will provide a programmers' eye view of categories and point to their relevance for our understanding of programming. Some part of this thesis is devoted to translating the mathematics to be found in such a category theory book into computer programs. This in itself is an interesting and novel exercise - to translate a mathematical textbook into instructions for a machine - to make electrons dance to a categorical tune, as remarked in [Burstall 1980]. Some basic knowledge of category theory will be assumed but our presentation of this program will follow the presentation normally to be found in textbooks, so the two may be read in parallel.

On the one hand, categories provide some framework for our understanding of programming and the specification of programs whilst, on the other hand, we have the requisite theory running on a machine. We shall see that this programming of category theory - this interplay of programs and programming theory - sheds light on some fundamental problems in the understanding of programming.

Our interest is largely in the relation of specifications of programs to programs themselves. How do we get from specifications to programs? How do we bridge what we may call the "computation gap"? Several approaches to this problem are known. For instance, starting with a specification it may be possible by some sort of successive refinement of specifications to get to one that is runnable i.e. a program. Or we may try a more direct approach, synthesising a program directly from the text of a specification. The latter cannot, in general, be automated even if we limit ourselves to equational specifications. However, this should not preclude attempts at understanding the process of program synthesis nor prevent us enlarging the class of specifications from which programs can, by some means, be directly synthesised. We will describe some experiments in the synthesis of programs from certain forms of specifications. This will make extensive use of the programming of category theory.

Another approach to the same problem starts at the programming end and tries to make available to the programmer more expressive program features and more powerful general routines. In this way programs may gain the simplicity and succinctness of specifications. In this context, the work of Backus [1978] has been a major inspiration. Backus advocates a 'combinatorial programming' in which high-level general functionals are provided with which to construct programs. The idea is that programs so constructed should contain little complex code - all the complexity is transferred to the general routines. In particular, few, if any, explicit iterations or recursion should be needed by the programmer. To construct programs in this way, we must both view the problem to be programmed in the right light and have routines which have the 'correct' degree of generality. The routines in which we encode category theoretic constructions are certainly 'general' - they have the generality of category theory itself. We will show how these routines can be used to produce programs with an example - a program which 'implements' the semantics of a specification language. The semantics of this language is 'categorical' in the sense that the syntactic operations are interpreted as operations in a category

Later we investigate specifications themselves using elements of category theory. We are interested in the properties of theories as denotations for algebraic specifications. The so-called monadic theories are a categorical means of handling these denotations. We look at several properties of these monadic theories and establish some results of mathematical interest, showing that these theories are sufficiently well-behaved to be used in program specification. More than this, our techniques of implementing category theory on computers can be used to encode some of our work on monadic theories. This intimate relation between programs and specifications gives us further insight into the process of synthesising programs.

The Contents

The first three chapters describe the programming of category theory. Starting with the basic ideas of a category and a functor we describe certain ways of putting categories together to get new categories. This provides a means of building complex data types from simpler types. Then we look at the implementation of colimits. In particular we are interested in making tractable the, often awkward, computation of colimits in categories. We show that there are ways of building these colimit programs using theorems from category theory.

Having established a basic repertoire of routines implementing certain aspects of category theory, we put them to use. In chapter four we describe an implementation of the semantics of a specification language based upon these programs. This provides an example, for a large and complex program, of the program development technique described above. The program itself is of some interest as it can be seen as a small part of what would be needed for an automatic (or semi-automatic) synthesis of programs from specifications.

In chapter five we investigate a systematic means of developing programs from certain types of specifications.

We then turn to specifications themselves and investigate monadic theories, which are a functorial formulation of the notion of a "theory". We establish that, under suitable conditions:

1. Signatures and presentations may be defined for monadic theories and free theories on a signature may be constructed.

2. Every theory morphism gives rise to an adjunction between categories of algebras. Moreover, a collection of algebras of a theory give rise to a new theory with certain properties.

3. Finite colimits and certain factorisations exist in categories of monadic theories.

4. Many-sorted, order-sorted and even category-sorted theories may be handled by extending somewhat the notion of monadic theories.

These results show that monadic theories are sufficiently well-behaved to be used in the semantics of algebraic specification languages. Some of the constructions used to establish these results can be encoded using the techniques in the first three chapters. We thus have programs which implement theory constructions. We look at applications of these programs.

Related Work

The programming of category theoretic results was an experiment - try it and see. We were not, of course, without motivation and it may be of use to place this work in the context of current research. The following have been of some influence:

(1) It has been observed that programs often have an algebraic (or categorical) interpretation and that this leads to more general and better understood programs. For instance it sometimes happens that a collection of functions in a program turn out to be an adjunction - the two functors and the unit and counit - or that a program not obviously connected with category theory can be viewed as the computation of colimits. The papers [Burstall and Landin 1969] and [Burstall 1980] give examples of the connection between programs and algebraic (and categorical) concepts. More examples will be found in

the following chapters.

(2) As mentioned above Backus' work on 'functional programming' [Backus 1978] led us to look for more general functionals and thus to employ the generality achieved by category theory. Part of the inspiration for this comes from Landin's desire for "language-free programming" [Landin 1969]. A comparison of the approach taken by Backus with our programming of category theory can be found on page 12.

(3) Category theory not only achieves a generality at which programming is still possible but also allows us to formalise concepts from programming and program specification. The application of universal concepts from category theory has proved especially fruitful. Members of the ADJ group (see for instance [Goguen, Thatcher and Wagner 1978]) clearly state the need for universal constraints on specifications (initial and free algebras of abstract data types). Colimits have been used to build specifications [Ehrich 1978] and [Burstall and Goguen 1980b], in graph grammars [Ehrig, Pfender and Schneider 1973], in record handling [Ehrig et al. 1980] and in the theory of data types [Lehmann and Smyth 1977]. We are to use colimits as a means of developing recursive programs from specifications. A direct link between the notion of programs and that of categories was noticed in [Burstall and Thatcher 1974].

We are impressed by the succinctness and applicability of algebraic specifications and that problems often seem to fit naturally into this framework (this has long since been known to mathematicians - see [Birkhoff 1938] for instance). The connection between algebraic specifications and programs to implement them seems quite tantalising.

The work of Guttag, Horowitz and Musser (1978), Liskov and Zilles (1974) and Ehrich (1978) as well as many others has been influential. An up to date bibliography of work on abstract data types can be found in [Dungan 1979]. We are indebted to members of the ADJ group for formalising algebraic specification techniques and noticing that universal constraints (initiality of algebras for instance) are involved in their interpretation. The extended exercise in using algebraic methods to handle programming languages and prove the correctness of compilers to be found in [Morris F.L. 1973] and [Thatcher, Wagner and Wright 1981] is of interest. We are to use the CLEAR specification language [Burstall and Goguen 1980b] (described briefly in Appendix Four) to describe programming problems which arise. CLEAR gives us a means of building algebraic specifications from theories in a modular manner. Ideas on how CLEAR may lead to a program development system are given in [Burstall and Goguen 1980c].

Monadic theories, which we investigate in later chapters, were introduced by Godement (1958). Much work has since been done on the applications and properties of these theories (see for instance [Manes 1976]). Particularly close to our own work is that of Adamek (1979) and Adamek and Koubek (1980).

Acknowledgements

Formal acknowledgements will be found in notes at the end of each chapter. However I would like to mention here my indebtedness to some who have helped me. Without the work, encouragement and guidance of Prof. Rod Burstall this thesis could never have come into existence. Some of the contents of this thesis are directly due to Prof. Rod Burstall (e.g. much of the programming of category theory [Burstall 1980]) whilst other parts arose in collaboration with him. The work here is a small part of a large project concerned with the program specification. Don Sannella here at Edinburgh, with whom I collaborated on parts of this thesis, gladly made available his experience in programming and specification. Both Don and Rod scrutinised carefully a draft of this thesis for which I am grateful. I would like to express my appreciation of the work and advice of Prof. Joe Goguen and Dr. Gordon Plotkin.

The research was carried out at the University of Edinburgh under a grant from the Science and Engineering Research Council.

CHAPTER ONE
COMPUTING WITH CATEGORIES

In this chapter and the two following, we look at an experiment in which we translate theorems of mathematics (or, more precisely, the constructive part of the proofs of theorems) into programs. The branch of mathematics with which we are concerned is, as discussed above, the theory of categories.

Categories consist of a collection of objects together with morphisms between these objects. The morphisms have a composition upon them and to each object there is an identity morphism on the object. Formally, we define a category as follows:

<u>Definition</u>

A <u>category</u> is a class O (of objects) and M (of morphisms) such that M can be expressed as a disjoint union:

$$M = \bigcup_{a,b \in O} [a,b], \text{ where } [a,b] \text{ are sets.}$$

There are the following operations:

1. For each triple of objects in O, (a,b,c), there is a composition:

   $[a,b] \times [b,c] \to [a,c]$

   If f is in [a,b] and g in [b,c] the image in [a,c] is denoted fg.

2. For each object c there is an (identity) morphism in [c,c], denoted i_c.

The following axioms are satisfied:

1. Associativity of composition: (fg)h = f(gh) whenever both sides are defined.

2. Left and right identity: If f is in [a,b], $i_a f = f$ and $f i_b = f$.

Sets together with set functions form a category as do partial-orders and monotonic functions. The simple step of introducing morphisms between objects, that is, the transition from a set (or class) of objects to a category of these objects gives us an amazingly powerful theory. Since category theory is no more than an abstraction of the theory of functions - their composition and identities - it may be wondered how category theory gives us the ability to construct objects (which is its role in programming theory). The key to this lies in the fact that we can formalise "specification by universal properties" in category theoretic terms - i.e. in terms of the existence and properties of morphisms.

We are already familiar with this type of specification by a universal property in the case of initial algebras of equational data types. Initiality is an example of a universal property. The importance of this means of specification is that the universality is not explicit in the specification. No need to declare the existence or uniqueness of the object specified nor indeed many of its properties which arise directly from the universality. This considerably simplifies specifications of problems. More than this, we often have a means of constructing the universal object in question both in a mathematical sense and computationally. We will be showing how this may be accomplished computationally in this paper. We shall also turn our attention to the specification of algorithms and how universal properties encode the behaviour required.

The work of the first part of this paper may be seen to be giving some of the power of mathematics to the programmer. For instance, how are we to handle rational numbers computationally? One may suggest that rational numbers should be represented as equivalence classes of certain pairs of integers. Available to the mathematician, though not so far to the programmer, is the more elegant and, at the same time, more general description of the rationals as the minimum field containing the integral domain of the integers. This is a typical "universal specification". We shall be looking at specifications of this form and showing how they can give rise to programs.

Part of the power of mathematics lies in its ability to build its analysis hierarchically by proving theorems and then using these theorems as elementary deductive rules for further analysis. It is a facility that programmers may well envy. Something of this is available to programmers in the various forms of data abstraction that have become current. However, it should be understood that it is not so much the ability to 'abstract' as to choose the right things to elevate to theorems which gives us the power in mathematics. Our long experience with mathematics has taught us what we should look for in a theorem - what sort of generality and what sort of simplicity can be expected of a theorem.

Can we do a similar thing in programming by a careful choice of 'general' routines so that these routines may be used as basic building blocks for programs? Several approaches to this are already known. We have already mentioned the work of Backus on functional programming. This can be viewed as an extension of such things as the APL '/' operator and iterators available in CLU and ALPHARD.

Our approach to constructing general routines which, it is hoped, will do much of the work of programming for us, is to cull the experience gained in mathematics by directly translating powerful general theorems into computer programs. Later we will show how this enables us to build programs in a modular manner by using these routines to handle a large part of the complex code required. We will exhibit a programming example - that of an 'implementation' of the semantics of a language, in our case, of a specification language.

Comparing this categorical programming with the approach taken by Backus brings several points to light:

(1) By introducing categories as basic objects in our programming we have solved the problem of abstracting on data types. That is our programming is not limited to any specific data types. For example graphs may be handled just as well as simpler types e.g. sets. A functional is specialised to the data type in question by applying it to the category of objects of the type. Increasing the complexity of the types in question should not appreciably increase the complexity

of the programming.

(2) The level of generality of our approach may be questioned. As mentioned above programs specifically concerned with category theory (e.g. the program which implements a categorical semantics) may be constructed using the general routines provided. The question arises - what about general programming tasks - tasks that at first sight have little or nothing to do with category theory? Do the routines that we provide give a means of constructing programs which implement any given program specification? At the moment we do not know. What we can say is that, unlike Backus' work, there is a mathematical connection between the routines we derive from category theory and algebraic specifications of programs. This gives us some hope that we are working at the 'correct' level of generality. Later (chapter five) we show that the connection between specifications and our categorical programming gives (for certain types of specifications) a systematic means of program development.

(3) Backus' 'functional programming systems' have associated laws which provide an algebra of programs. In our case the laws are derived from category theory. A typical example, concerning colimit programs, would be the commutativity (to within an isomorphism) of binary coproduct: "coproduct(a,b) = coproduct(b,a)".

In constructing programs from a collection of general routines it is hoped that the iterative and recursive features will be encapsulated in these routines and therefore eliminated from the task of programming. We will see that certain parts of our programming of category theory do indeed encapsulate iterative and recursive constructs and therefore allow us to program in a 'combinatorial' style. To emphasise this we will try to point out where iteration or recursion occurs in the programming.

To handle general routines which take other routines as parameters, we need a language which is either typeless (e.g. LISP) or else typed but allowing the passing of functions as arguments. Any such language would suffice. We choose a strongly typed applicative language, called HOPE (after Thomas Hope of Rankeillor - active c.1740), developed here at Edinburgh. HOPE is fully higher order - it allows all data, including functions, to be arguments. We choose HOPE partly to gather some experience of programming in this experimental language but also because, being an ISWIM-like applicative language, our HOPE programs look somewhat like the mathematical texts which they are intended to encode. For those unfamiliar with languages of this sort there is a summary of the main features of HOPE in Appendix Three. A detailed exposition of the language may be found in [Burstall,MacQueen,Sannella 1979].

CATEGORIES AND FUNCTORS

Our development starts with the basic concept of a category. In programming terms, what is a category? It is a data type parameterised on the types of the objects and morphisms in the category and having the following components: two functions from morphisms to objects giving the source and target of the morphisms, a function from objects to morphisms giving the identity morphism on each object and a composition function taking pairs of morphisms to morphisms. In HOPE this is declared by:

```
    data Cat(o,m) == cat((m->o),(m->o),(o->m),(m#m->m))
```

In this declaration, "Cat" is a type constructor taking the type variables "o" and "m" (object and morphism types respectively) whilst "cat" is a constructor of data objects. We will, throughout our programming, use the following convention:

Convention

The initial letters of type constructors will be capitals, whilst constructors of data objects will begin with small letters.

For those unfamiliar both with categories and with this style of data declaration it may be helpful to compare the above declaration with that of a more familiar data type. Consider the data type "list" which is parameterised on the type of the objects appearing in the list. In HOPE it can be declared as a recursively defined type by:

```
    data List(alpha) == nil ++ cons(alpha,List(alpha))
```

Here the "++" is the disjunction of type instances.

A function on lists can be defined as follows:

```
dec length : List(alpha) -> num

--- length(nil) <= 0
--- length(cons(a,l)) <= length(l) + 1
```

The keyword "dec" introduces a declaration of the type of a function whilst "---" introduces a function definition.

Before we move on to implementing further concepts from the theory of categories, let us examine more closely what we have done in the above declaration of a category. Firstly, notice that we have not included the axioms which a category is required to satisfy (for instance, the associativity of composition). In most programming languages, we do not have the ability to declare axiomatic data types. Inclusion of such axioms would destroy or seriously limit any type checking that could be done on programs. However, these constraints on data types can be considered as correctness criteria for programs rather than included as part of type declarations. To gain some control over these verification details, we may impose upon the program a discipline in which categories are either basic categories for which the the axioms are known to hold (for instance, the category of sets) or are built from other categories by category constructors which are known to preserve the axioms (for instance, the comma category construction). In this way the correctness of the implementation is assured. We could in fact use the type discipline of HOPE to enforce such a scheme but this makes the type of a category unnecessarily obscure. What we are really saying is that we are to consider categories as implementations of a data type whose operations are category constructors. This is an interesting view of categories particularly so because colimits in categories "commute" with these category constructors.

These comments upon the correctness criteria of our programs are important since, in the following pages, we are going to translate proofs in category theory into programs. In these programs we abstract only the constructive part of the proofs omitting the checking of properties which turn a construction into a proof. That we have some control over these properties is therefore of importance.

In this declaration of a category we do not include a set of objects or morphisms - in general the collection of objects and morphisms is "larger" than a set. If the objects and morphisms do form sets then we have a new type to be called "Small_Category", which will be used later in our work. Notice further that the composition of morphisms takes any pair of morphisms as argument whereas, of course, it should take only composable pairs - those for which the target of the first is the source of the second. We handle this error condition by introducing, for each category, a morphism "undef". We can then set the composition of non-composable pairs equal to this "undef".

To access the operations of a category we introduce the following "projection" functions. These are not strictly necessary, but help program readability.

```
    dec source :   Cat(o,m) -> (m->o)
    dec target :   Cat(o,m) -> (m->o)
    dec identity : Cat(o,m) -> (o->m)
    dec compose :  Cat(o,m) -> ((m#m) -> m)

    --- source(cat(s,_,_,_))   <= s
    --- target(cat(_,t,_,_))   <= t
    --- identity(cat(_,_,i,_)) <= i
    --- compose(cat(_,_,_,c))  <= c
```

The underbar denotes a missing (unnecessary) variable.

What is an instance of this type going to look like? The basic category which we consider is the category of sets. With this category we are able to build many familiar data types.

The objects in the category of sets are to be, of course, sets. We shall parameterise our sets on the type of elements they contain. Morphisms in the category of sets are to be set functions. However, to ensure that we can extract the source set and the target set from a morphism, we introduce a new data type:

```
    data Set_Mor(alpha) ==
        mor(Set(alpha),(alpha->alpha),Set(alpha)) ++ undef
```

where 'alpha' is a type variable and '++' is the 'sum' of type cases (allowing case analysis). We can then define the functions required to form the category:

```
dec s : Set_Mor(alpha) -> Set(alpha)
dec t : Set_Mor(alpha) -> Set(alpha)
dec id : Set(alpha) -> Set_Mor(alpha)
dec comp: Set_Mor(alpha)#Set_Mor(alpha) -> Set_Mor(alpha)

--- s(mor(x,_,_))                <= x
--- t(mor(_,_,y))                <= y
--- id(x)                        <= mor(x,ident,x)
--- comp(mor(x,f,y),mor(u,g,v)) <= mor(x,f.g,v) if u=y
                                          else undef
```

The category of sets then looks like this:

```
dec cat_of_sets : Cat(Set(alpha),Set_Mor(alpha))
--- cat_of_sets <= cat(s,t,id,comp)
```

In this rather simple way we can define categories. In the next section we shall be looking at ways of constructing new categories from old which will give us powerful techniques for building categories of large and complex data types from simple types - in fact, all the data types to be used arise via these category constructors from the category of sets alone.

Before turning to this, we introduce a further concept from category theory. Maps between categories are functors. A functor is a pair of maps, on objects and morphisms, which preserve the source, target, identity and composition. As a data type, however, it is just a pair of maps:

```
data Functor(o,m,o1,m1) == functor((o->o1),(m->m1))
```

Functors can be applied to objects and to morphisms

```
    dec ofo : Functor(o,m,o1,m1) # o -> o1
    infix ofo : 5
    --- functor(Fo,_) ofo o <= Fo(o)

    dec ofm : Functor(o,m,o1,m1) # m -> m1
    infix ofm : 5
    --- functor(_,Fm) ofm m <= Fm(m)
```

Functors can also be composed:

```
dec . : Functor(o,m,o1,m1) # Functor(o1,m1,o2,m2) ->
                Functor(o,m,o2,m2)
infix . : 3
--- functor(fo,fm) . functor(go,gm) <= functor(fo.go, fm.gm)
```

where the dot on the right-hand side is function composition. The keyword "infix" introduces infix operations and gives their precedence.

For future reference, notation for the application of a functor F to an object x will be xF, so that application is 'diagrammatic'. The notation for the composition of functors is consistent with this: If F and G are functors, x(FG) = (xF)G.

Notes

The genesis of the programming of category theory is described in [Burstall 1980]. Prof. Rod Burstall, in order to gain insight into the computational aspects of colimits, coded up the colimits of sets in a language called NPL, a precursor of HOPE. The present development of the programming of category theory (up to page 52) owes much to this early version. Rod also outlined some of the code here including that arising from the colimit existence theorem (page 41) and suggested the lifting of colimits. Otherwise it was a joint project between Don Sannella and myself. Don's experience of programming and knowledge of the HOPE system together with my knowledge of categorical matters meant that the work was shared roughly equally.

The HOPE programming language is described in [Burstall, MacQueen and Sannella 1979].

CHAPTER TWO

CONSTRUCTING DATA TYPES - A CATEGORICAL VIEW

One way of developing large programs reliably is to 'structure' them in some manner and build programs hierarchically by fitting together component programs. This hierarchical and modular means of building the complex from the simple is of very general application in programming. For instance it may be used in the building of program specifications [Burstall,Goguen 1980b].

In this section we look at a similar approach to the building of data types from component types. Our type building operations are taken from category theory in which they appear as category constructors. Constructors of categories can be considered to be constructors of data types, if objects of the types can be considered to be objects in categories. Later, we look briefly at the formal relationship between data types and categories (page 96).

The category constructors we are to use have an important computational property. In the next chapter our attention will turn to colimits and means for computing them. It turns out that we can "lift" computations of colimits through these category building operations. That is, at the same time as we build up new types, we can build colimit operations on these types. This greatly simplifies the computation of colimits. As we shall see colimits prove useful in investigating the structure of data objects and allow us to develop programs from specifications.

We now look at some of these category building operations. We show their use for constructing familiar types and how we can implement them. Our first such operation is the construction of comma categories, due to Lawvere [1963b]. In this case we build new data types as morphisms between objects of old types.

THE COMMA CATEGORY

Many data types familiar in programming can be considered to be constructed as comma categories. We first define comma categories and then illustrate this with some examples.

Categories contain both objects and morphisms. Thus when considering any objects in programming, we should ask whether there are morphisms between objects so that a category can be defined. That is, we want to elevate morphisms so that they share an equal status with the objects. How about a category then whose objects are these morphisms? It seems only fair to the morphisms. This is the idea behind the comma category construction. The general definition goes as follows.

Definition

Consider the following picture of categories and functors:

$$\begin{tikzcd} \underline{A} \arrow[r, "L"] & \underline{B} & \underline{C} \arrow[l, "R"'] \end{tikzcd}$$

We construct a category, the comma category, denoted (L,R) as follows:

The objects are triples of the form (a, f: aL -> cR, c) with a an object of A and c an object of C.

A morphism in (L,R) from (a, f: aL -> cR, c) to (a', f': a'L -> c'R, c') is a pair of morphisms one from A the other from C, (g: a -> a', h : c -> c'), such that the following square commutes.

$$\begin{tikzcd} aL \arrow[r, "gL"] \arrow[d, "f"'] & a'L \arrow[d, "f'"] \\ cR \arrow[r, "hR"'] & c'R \end{tikzcd}$$

□

Special cases of this construction arise from restricting either or both of the functors L and R to be constant or identity functors. A case that we are to use arises if A = B and L is the identity. Then objects are of the form (b, f: b -> cR, c). Such a category we shall call a right comma category and denote it by ($\underline{B}$,R). If, on the other hand, $\underline{A}$ is a category with one object and the image of this object in L is b, the we call the category a right object comma category and denote it by (b,R). Similar cases arise from restricting the functor R to be an identity or constant functor.

For further details of this construction consult [Mac Lane 1971].

Examples of Comma Categories

Consider graphs, by which we mean directed multigraphs possibly with loops and cycles. It is with graphs of this sort that we shall be working for most of the time so we will reserve the word "graph" for these graphs alone. These graphs, then, may be described by a set of nodes and a set of edges and two maps, called source and target, which take edges to nodes. We thus have the picture:

    source, target : E -> N

How is this to be an object in a comma category? We need to combine the two set morphisms into one morphism. This can be done by the isomorphism (in cartesian closed categories, in this case that of sets):

$$(X \to Y) \times (X \to Z) \cong (X \to (Y \times Z))$$

The morphism is then,

    f : E -> N x N,

such that f(e) = (source(e),target(e)). The crossproduct of sets can be extended to a functor, X, by defining an obvious morphism part.

Morphisms of graphs are pairs of functions, one mapping edges to edges the other mapping nodes to nodes, such that 'source' and 'target' are preserved. If f : E -> N x N and f' : E' -> N' x N' are two graphs and (p: E -> E', q: N -> N') is a graph morphism, the preservation of 'source' and 'target' is equivalent to the commutation of the following square:

$$\begin{tikzcd}
E \arrow[r, "p"] \arrow[d, "f"'] & E' \arrow[d, "f'"] \\
N \times N \arrow[r, "q \times q"'] & N' \times N'
\end{tikzcd}$$

Thus, if Set is the category of sets and set functions, we see that graphs can be described as objects in the comma category (Set,X) and then morphisms of graphs are morphisms in this category. Thus the category of graphs is (isomorphic to) the comma category, (Set,X).

If we consider instead undirected graphs then we merely need to change the crossproduct functor to the functor which gives unordered pairs of nodes, which we may denote by $\mathfrak{X}$. Thus the category of undirected graphs is (isomorphic to) (Set,$\mathfrak{X}$).

A relation on a set N - by which we mean a subset of NxN - can also be expressed as an object in a comma category. For this we need to revise our picture a little. The observation required is that a relation can be defined by a map of the form

f : N x N -> 2

where 2 is any two element set (say, {true,false}). Then, if m and n are related f(m,n) = true, otherwise f(m,n) = false. Again this is an object in a comma category, this time the category, (X,2), where X is the crossproduct functor. Notice that morphisms in this category preserve the unrelatedness of elements as well as their relatedness.

Notice how these categories have been built from the category of sets using suitable functors. We may continue this to more elaborate data types. Consider, for instance, signatures. A signature is a set of sorts, S, and a set of operations, O, together with an arity for each operation. An arity is a string of sorts, the last sort in the string being the output sort and the remainder being the input functionality. Now if "strings" is the functor taking a set, S, onto the set of non-empty strings on S, S^+, (with morphism part as string substitution) then a signature is a pair of sets, S and O together with a set morphism:

$$O \rightarrow S^+$$

This is an object in the category (<u>Set</u>,strings). Noting that signature morphisms are correctly described by morphisms in this category, we see that this comma category is (isomorphic to) the category of signatures.

We now turn to the implementation of comma categories and show how these various data types arise.

<u>Implementation of Comma Categories</u>

We will be dealing with the following picture of categories and functors:

$$\begin{tikzcd} \underline{A} \arrow[r, "L"] & \underline{B} & \underline{C} \arrow[l, "R"'] \end{tikzcd}$$

Let the three categories <u>A</u>, <u>B</u> and <u>C</u> have types Cat(o,m), Cat(o1,m1) and Cat(o2,m2) respectively. An object of the comma category (L,R) is a triple of type:

```
o # m1 # o2.
```

A morphism in this comma category, in our treatment, contains not only an <u>A</u>-morphism and a <u>C</u>-morphism but also the source and target comma-objects. We therefore declare the type:

```
data Comma_Mor(o,m,o1,m1,o2,m2) ==
        comma_mor((o#m1#o2),(m#m2),(o#m1#o2))
```

To define the comma category, (L,R), we need the four functions: source, target, identity and composition. These are defined by:

```
    dec s : Comma_Mor(o,m,o1,m1,o2,m2) -> o#m1#o2
    dec t : Comma_Mor(o,m,o1,m1,o2,m2) -> o#m1#o2
    dec id : Cat(o,m) # Cat(o2,m2) ->
                    ((o#m1#o2) -> Comma_Mor(o,m,o1,m1,o2,m2)
    dec comp : Cat(o,m) # Cat(o2,m2) ->
       (Comma_Mor(o,m,o1,m1,o2,m2) # Comma_Mor(o,m,o1,m1,o2,m2)
            -> Comma_Mor(o,m,o1,m1,o2,m1))

    --- s(comma_mor(a,_,_)) <= a
    --- t(comma_mor(_,_,c) <= c
    --- id(A,C) <= (lambda obj & (a,_,c) =>
         comma_mor(obj,(identity(A)(a),identity(C)(c)),obj) )
    --- comp(A,C) <=
         (lambda comma_mor(Y,(f,g),_),comma_mor(_,(h,j),Z) =>
              comma_mor(Y,(compose(A)(f,h),compose(C)(g,j)),Z) )
```

Thus, for any pair of functors, the comma category is given by:

```
    dec comma_cat : Cat(o,m) #
                    Functor(o,m,o1,m1) #
                    Cat(o1,m1) #
                    Functor(o2,m2,o1,m1) #
                    Cat(o2,m2) ->
                        Cat((o#m1#o2),Comma_Mor(o,m,o1,m1,o2,m2))

    --- comma_cat(A, L, B, R, C) <= cat(s,t,id(A,C),comp(A,C))
```

The various restricted types of comma categories (in which either or both of the functors L and R are set to the identity functor or to constant functors) are obtained as instantiations of the general comma category as follows.

The identity functor on a category is defined by,

```
    dec I : Cat(o,m) -> Functor(o,m,o,m)
    --- I(_) <= functor(ident,ident)
```

where "ident" is the identity function, ident(x) = x. Constant functors are given by,

```
    dec K : Cat(o,m) -> (o -> Functor(Num,Num,o,m))
    --- K(cat(_,_,id,_)) <=
            (lambda a =>
               functor((lambda 1 => a),(lambda 1 => id(a))) )
```

where the source of the functor is the unit category consisting of one object and its identity both of which are the number '1'.

By setting L to be the identity we get the right comma category:

```
    type Right_Comma_Mor(o1,m1,o2,m2) ==
                   Comma_Mor(o1,m1,o1,m1,o2,m2)

    dec right_comma_cat :
          Cat(o1,m1)#Functor(o2,m2,o1,m1)#Cat(o2,m2)
               -> Cat((o1#m1#o2),Right_Comma_Mor(o1,m1,o2,m2))
    --- right_comma_cat(B,R,C) <= comma_cat(B,I(B),B,R,C)
```

Notice that we can define types in terms of other types and type-operators, introducing such definitions with the keyword "type".

Likewise by setting R to be the identity we get the left comma category. If L is a constant functor, we call the category a right object comma category. These are given, in a like manner, by,

```
    type Right_Obj_Comma_Mor(o1,m1) ==
                   Comma_Mor(o1,m1,o1,m1,Num,Num)

    dec right_obj_comma_cat : Cat(o1,m1)#o1 ->
          Cat((o1#m1#Num),Right_Obj_Comma_Mor(o1,m1))
    --- right_obj_comma_cat(B,b) <=
          comma_cat(B,I(B),B,K(B)(b),unit_category)
```

where "unit-category" is defined above. Similarly we can define the left object comma category.

There are projection functors,

    left: (L,R) -> A

    right: (L,R) -> C

defined by:

```
    dec left : Functor((o#m1#o2),Comma_Mor(o,m,o1,m1,o2,m2),o,m)
    --- left <= functor( (lambda (a,_,_) => a),
                         (lambda comma_mor(_,(f,_),_) => f) )

    dec right :
          Functor((o#m1#o2),Comma_Mor(o,m,o1,m1,o2,m2),o2,m2)
    --- right <= functor( (lambda (_,_,c) => c),
                          (lambda comma_mor(_,(_,g),_) => g) )
```

How do we construct the category of graphs? Graphs, as comma objects, are given by the type definition:

```
    type Graph(alpha) == Set(alpha) # Set_Mor(alpha) # Set(alpha)
```

where 'alpha' is a type variable, that of the type of the labels in the graph. Graph morphisms are:

```
    type Graph_Mor(alpha) ==
          Right_Comma_Mor(Set(alpha), Set_Mor(alpha),
                          Set(alpha), Set_Mor(alpha))
```

To construct the category of graphs we need the crossproduct functor. Before defining this functor we need to explain a tagging mechanism which we use. Because of the strong type discipline in HOPE, we cannot describe graphs as morphisms from sets of type Set(alpha) to sets of type Set(alpha x alpha). We need to coerce pairs of type "alpha x alpha" to objects of type "alpha". This may be done by a explicit coercion:

```
    data Tag(alpha) == just(alpha) ++ pair(alpha,alpha) ++ ...
```

Thus objects of type Tag(alpha) are either objects of type "alpha" labelled with "just" or pairs of objects of type "alpha" labelled with "pair" (the continuation "..." indicates that extra tagging will be needed later).

The crossproduct functor is then given by,

```
    dec ocrossprod: Set(Tag(alpha))->Set(Tag(alpha))
    --- ocrossprod(S) <= pair * (S X S)

    dec mcrossprod: Set_Mor(Tag(alpha)) -> Set_Mor(Tag(alpha))
    --- mcrossprod(mor(s,f,t)) <=
            mor( ocrossprod(s),
                 (lambda pair(a,b) => pair(f(a),f(b))),
                 ocrossprod(t))

    dec cross_product:
            Functor( Set(Tag(alpha)),Set_Mor(Tag(alpha)),
                     Set(Tag(alpha)),Set_Mor(Tag(alpha)))
    --- cross_product <= functor(ocrossprod,mcrossprod)
```

where X is the cartesian product of two sets and the set f*S, for a function f and a set S, is {f(s) | s in S}. With this functor we describe the category of graphs by:

```
    dec cat_of_graphs: Cat(Graph(Tag alpha),Graph_Mor(Tag alpha))
    --- cat_of_graphs <=
           right_comma_cat(cat_of_sets,cross_product,cat_of_sets)
```

Notice how, using a familiar set functor, the comma category construction above immediately gives us the data type of graphs.

We have chosen to represent graphs as comma objects. This makes their formal behavior amenable but as a representation it is rather clumsy and includes duplication of information. We thus introduce a function for the construction of these comma objects from the relevant data:

```
    dec graph :·
         Set(Tag alpha)#(Tag(alpha)->Tag(alpha))#Set(Tag alpha)
                        -> Graph(Tag alpha)
    --- graph(E,f,N) <= (E,mor(E,f,cross_product ofo N),N)
```

We can also introduce decomposition functions. Strictly these are not necessary as they can be obtained by "pattern matching" but they will make subsequent programs more readable.

```
    dec edges : Graph(Tag(alpha)) -> Set(Tag(alpha))
    dec nodes : Graph(Tag(alpha)) -> Set(Tag(alpha))
    dec graph_map : Graph(Tag(alpha)) ->
                        (Tag(alpha) -> Tag(alpha))
    dec dom :   Graph(Tag alpha) -> (Tag(alpha) -> Tag(alpha))
    dec range : Graph(Tag alpha) -> (Tag(alpha) -> Tag(alpha))
    --- edges( (E,_,_) ) <= E
    --- nodes( (_,_,N) ) <= N
    --- graph_map( (_,mor(_,f,_),_) ) <= f
    --- dom( (_,mor(_,f,_),_) )  <=
                  lambda e => n1 where pair(n1,_) == f(e)
    --- range( (_,mor(_,f,_),_) ) <=
                  lambda e => n2 where pair(_,n2) == f(e)
```

Signatures as comma objects can be handled similarly. There is a difficulty here in that the "strings" functor takes finite sets to infinite sets. All infinite sets in our program are represented by the constant constructor for sets "bigset".

Again we need a coercion of types. This time it is of 'strings of alpha' to 'alpha':

```
    data Tag(alpha) ==
            just(alpha) ++ ... ++ string(List Tag(alpha)) ...
```

where 'string' is another constructor for the type 'Tag(alpha)'. The "strings" functor is given by

```
    dec ostrings : Set(Tag alpha) -> Set(Tag alpha)
    dec mstrings : Set_Mor(Tag alpha) -> Set_Mor(Tag alpha)

    --- ostrings(S) <= nil_set if S=nil_set else bigset
    --- mstrings(mor(s,f,t)) <=
                  mor( ostrings(s),
                       (lambda string(l) => string(f*l)),
                       ostrings(t) )

    dec strings : Functor( Set(Tag alpha),Set_Mor(Tag alpha),
                           Set(Tag alpha),Set_Mor(Tag alpha))
    --- strings <= functor(ostrings,mstrings)
```

Its only non-trivial part is its action on morphisms which is a string substitution and uses the predefined recursive 'maplist' function, "*".· Signatures and their morphisms as objects and morphisms of a comma category are:

```
    type Signature(alpha) ==
            Set(alpha) # Set_Mor(alpha) # Set(alpha)

    type Signature_Mor(alpha) ==
            Right_Comma_Mor(Set(alpha), Set_Mor(alpha),
                            Set(alpha), Set_Mor(alpha) )
```

The category of signatures is then:

```
    dec cat_of_signatures :
            Cat(Signature(Tag alpha),Signature_Mor(Tag alpha))
    --- cat_of_signatures <=
            right_comma_cat( cat_of_sets,
                             strings,
                             cat_of_sets )
```

We shall be looking at colimits in these comma categories in the next chapter but now we turn to another method of combining two data types to give a new type. This time we are given two categories and produce the category of all functors from one category to the other. Using this, we can construct diagrams in categories. Moreover these functor categories are closely associated with theories which occur in program specification (page 101).

FUNCTOR CATEGORIES

A functor is to be, as before, a pair of maps on objects and morphisms.

```
    data Functor(o,m,o1,m1) == functor((o->o1),(m->m1))
```

To make a category whose objects are functors we need to introduce morphisms between functors. Suppose that F,G : $\underline{A}$ -> $\underline{B}$ are two functors, then a morphism from F to G is a natural transformation α : F $\dot{\rightarrow}$ G. It is defined to be a map taking an object, a, of $\underline{A}$ to a morphism, f : aF -> aG, of $\underline{B}$ such that certain diagrams commute. In terms of a data type, it is simply a map from objects to morphisms together with the source and target functors:

```
    data Nat_transform(o,m,o1,m1) ==
            nat_transform( Functor(o,m,o1,m1),
                           (o->m1),
                           Functor(o,m,o1,m1))
```

There are two compositions of these natural transformations - the vertical or "dot" composition and the horizontal or "ring" composition. These can both be encoded:

```
    dec dotcomp : Cat(o,m)#Cat(o1,m1) ->
         (Nat_transform(o,m,o1,m1) # Nat_transform(o,m,o1,m1)
                   -> Nat_transform(o,m,o1,m1))
    dec ringcomp : Cat(o2,m2) -> (Nat_transform(o,m,o1,m1)#
         Nat_transform(o1,m1,o2,m2) -> Nat_transform(o,m,o2,m2))

    --- dotcomp(A,cat(_,_,_,comp)) <=
       lambda  nat_transform(F,alpha,_), nat_transform(_,beta,H)
         => nat_transform( F,
                           (lambda a => comp(alpha(a),beta(a))),
                           H )
    --- ringcomp(cat(_,_,_,comp)) <=
       lambda  nat_transform(F,alpha,G), nat_transform(J,beta,L)
         => nat_transform( F.J,
                           (lambda a =>
                             comp(beta(F ofo a),L ofm alpha(a))),
                           G.L)
```

(Where 'ofo' and 'ofm' apply a functor to objects and morphisms respectively.) In fact, it is the "dotcomp" that is the composition in the category of functors.

The identity natural transformation is:

```
    dec id : Cat(o,m)#Cat(o1,m1) -> (Functor(o,m,o1,m1) ->
                Nat_transform(o,m,o1,m1))
    --- id(A,cat(_,_,i,_)) <= lambda F =>
                nat_transform(F,(lambda a => i(F ofo a)),F)
```

We can now define the category of functors between two categories.

```
    dec cat_of_functors : Cat(o,m)#Cat(o1,m1) ->
            Cat(Functor(o,m,o1,m1),Nat_transform(o,m,o1,m1))
    --- cat_of_functors(A,B) <=
                cat((lambda nat_transform(s,_,_) => s),
                    (lambda nat_transform(_,_,t) => t),
                    id(A,B),
                    dotcomp(A,B) )
```

It can be seen that this encoding of categorical concepts is quite easy once one has grasped the basic idea. We give a final example which will be of use later - the category of diagrams in a category.

Diagrams have a shape - a graph - and maps from nodes of this graph to objects in a category and from edges in the graph to morphisms. These maps 'preserve' the source and target. As a data type:

```
    data Diagram(o,m) ==
       diagram(Graph(Name),(Node->o),(Edge->m))
```

The types 'Name', 'Node' and 'Edge' are all defined to be the type 'Tag(Label)' where 'Label' is:

```
    type Label == word(List(Char)) ++ number(Num).
```

That is graphs are labelled with either words (character lists) or numbers.

If C̲ is a category, a diagram morphism in C̲ from D to D' (where fo,fo' and fm,fm' are the object and morphism maps of these diagrams) is a pair (α_N,α_E) of maps between nodes and edges of the diagrams, together with a map, f, which takes nodes in the source diagram to morphisms in C̲. The latter map satisfies:

$\mathrm{source}_C(f(n)) = fo(n)$ and

$\mathrm{target}_C(f(n)) = fo'(\alpha_N(n))$

together with certain commutation conditions. In HOPE, diagram morphisms are to be:

```
    data Diagram_Mor(o,m) ==
       diagram_mor( Diagram(o,m),
                    (Node->Node),
                    (Edge->Edge),
                    (Node->m),
                    Diagram(o,m) )
```

If C̲ is a category, the category of diagrams on C̲, <u>Diagram(C)</u>, is then,

```
    dec cat_of_diagrams : Cat(o,m) ->
                          Cat(Diagram(o,m),Diagram_Mor(o,m))
    --- cat_of_diagrams(c) <= cat(s(c),t(c),id(c),comp(c))
```

where 's', 't', 'id', 'comp' are the obvious source, target, identity and composition in the category of diagrams.

We shall need the following diagrams: the empty diagram, the diagram of two nodes and no morphisms and the diagram with a pair of objects and a parallel pair of morphisms between them. For example the diagram of two nodes and no morphisms is declared as:

```
    dec cpdiagram : o#o -> Diagram(o,m)
    --- cpdiagram(a,b) <= diagram( cpgraph( just(word("left")),
                                            just(word("right"))),
                                   (lambda just(word(x)) => a
                                             if x="left" else b),
                                   nil_fn)
```

Alternatively diagrams could be encoded as functors (possibly functors from 'small categories') and then their morphisms would be natural transformations (together with a functor between the 'shapes'). This would be altogether more elegant yet somewhat more obscure. As it is an equivalent formulation, we keep to the explicit declaration of diagrams and their morphisms above.

These category constructions all have a special property concerning colimits. Colimits in categories 'lift' through these category building operations, enabling us to express colimits of complex data types in terms of colimits of simpler types. Moreover, this fact can be encoded as a program. We now turn to the general problem of computing colimits in categories and show how this "lifting" is accomplished.

Notes

The programming of category theory described here was a joint project between Prof. Rod Burstall, Don Sannella and myself as explained in chapter one. Goguen and Burstall (1978) noticed that graphs and signatures can be viewed as objects in comma categories.

CHAPTER THREE

THE COMPUTATION OF COLIMITS

Colimits are a means of defining objects in a category in terms of other objects and morphisms between them. In other words, they allow us to build new objects by connecting together some given objects - a familiar concept in programming whenever we need to build complex objects from simpler components.

Colimits have been used in building specifications from theories ([Ehrich 1978] and [Burstall,Goguen 1980a]), in graph grammars [Ehrig,Pfender,Schneider 1973] and in the interpretation of data types ([Lehmann,Smyth 1977] and others). We will in the course of this paper be investigating other applications of colimits - for instance in program development. In this chapter we show how to compute colimits in categories using fairly general techniques.

Implementation of Colimits

Firstly we explain how we handle colimits and categories with colimits within our programming of category theory. So far, we have available an encoding of categories, functors, diagrams and of the category building operations. In order to define colimits we need cones. Fortunately, as with graphs, it turns out that cones may be represented by objects in a comma category.

A cone is a diagram (its base), an object (its apex), and morphisms to the apex from objects at the nodes in the base satisfying certain commutation conditions (see [Herrlich, Strecker 1973]). This can be represented as a diagram morphism by considering the apex to be a diagram with one object. Using this trick we can represent the category of cones as a comma category as follows:

If $\underline{C}$ is a category, $\underline{\text{Diagram}}(\underline{C})$ the category of diagrams on $\underline{C}$, then this 'making a diagram from an object' is functorial:

    unit_diagram : $\underline{C}$ -> $\underline{\text{Diagram}}(\underline{C})$.

For each object c, in $\underline{C}$, this gives a diagram with one object, c, and one morphism, the identity on c. Each morphism f : c -> c' in $\underline{C}$ gives a diagram morphism from unit_diagram(c) to unit_diagram(c').

The computation of this looks like:

```
    dec unitdiag : Cat(o,m) -> (o -> Diagram(o,m))
    --- unitdiag(C) <=  lambda o =>
            diagram(discrete_graph({just(word "solo")}),
                    constant(o),
                    constant(identity(C)(o)) )

    dec unitdiag : Cat(o,m) -> (m -> Diagram_Mor(o,m))
    --- unitdiag(C & cat(s,t,_,_)) <=
            lambda m => diagram_mor( unitdiag(C)(s(m)),
                                     ident,
                                     ident,
                                     constant(m),
                                     unitdiag(C)(t(m)) )

    dec unit_diagram :
           Cat(o,m) -> Functor(o,m,Diagram(o,m),Diagram_Mor(o,m))
    --- unit_diagram(c) <= functor(unitdiag(c),unitdiag(c))
```

The graph constructor 'discrete_graph' takes a set and gives a graph whose set of nodes and of edges are both the given set and the source and target of each edge is the node of the same name. It is defined on page 83.

The category of cones on a category C, Cone(C), is then (isomorphic to) the right comma category, (Diagram(C),unit_diagram). This is given by the following comma category construction:

```
    type Cone(o,m) == Diagram(o,m) # Diagram_Mor(o,m) # o
    type Cone_Mor(o,m) ==
           Right_Comma_Mor(Diagram(o,m), Diagram_Mor(o,m), o, m)

    dec cat_of_cones : Cat(o,m) -> Cat(Cone(o,m),Cone_Mor(o,m))
    --- cat_of_cones(c) <=
           right_comma_cat(cat_of_diagrams(c),unit_diagram(c),c)
```

As before with graphs, the representation of cones as comma objects includes redundancies, so we introduce a function to construct a cone as a comma object from the relevant data:

```
    dec cone : Cat(o,m) ->
            ((Diagram(o,m) # (Node->m) # o) -> Cone(o,m))

    --- cone(C) <= lambda d,f,a =>
       ( d,
         diagram_mor( d,
                      constant(just(word "solo")),
                      constant(just(word "solo")),
                      f,
                      unit_diagram(C) ofo a),
         a )
```

We also introduce the decomposition functions giving parts of a cone.

```
    dec base : Cone(o,m) -> Diagram(o,m)
    dec apex : Cone(o,m) -> o
    dec sides : Cone(o,m) -> (Node -> m)

    --- base(d,_,_) <= d
    --- apex(_,_,a) <= a
    --- sides(_,diagram_mor(_,_,_,f,_),_) <= f
```

The function, "apex", for instance, is functorial - it can be extended to cone morphisms by simply extracting the morphism between the apices :

```
    dec apex_morphism : Cone_Mor(o,m) -> m
    --- apex_morphism(cm) <= right ofm cm
```

The functor is :

```
    dec apex_functor: Functor(Cone(o,m),Cone_Mor(o,m),o,m)
    --- apex_functor <= right
```

The functor "right" is the projection functor associated with the comma category.

With cones available we can define colimits. Colimits are a special type of cone on a diagram. They have a universal property which we need to include alongside the colimiting cone itself. The universal property says that, given any other cone (on the same base), there is a unique cone morphism from the colimiting cone to the given cone. This defines a function and colimits can thus be expressed as a data type:

```
    type Colimit(o,m) == Diagram(o,m) ->
            (Cone(o,m) # (Cone(o,m)->Cone_Mor(o,m)))
```

A category with such a colimit operation upon it is a new data type which we call a colimit category.

```
    data Colimit_Cat(o,m) == colimit_cat(Cat(o,m),Colimit(o,m))
```

In keeping with familiar terminology we perhaps ought to call this a cocomplete (or finitely cocomplete) category but we reserve this name for later use for a somewhat different data type.

A colimit takes a diagram in a category to a cone. This can be extended to a functor by using the universal part of the colimit to define a morphism part of a functor. The object part is defined as follows:

```
    dec colimit_cone : Colimit_Cat(o,m) ->
            (Diagram(o,m) -> Cone(o,m))
    --- colimit_cone(colimit_cat(_,colim)) <=
            lambda d => let c,_ == colim(d) in c
```

Then the functor is:

```
    dec colimit_functor : Colimit_Cat(o,m)
             -> Functor( Diagram(o,m),Diagram_Mor(o,m),
                         Cone(o,m),Cone_Mor(o,m) )

    --- colimit_functor( cC & colimit_cat(C, colimit)) <=
          functor( colimit_cone(cC),
             (lambda diagmor =>
               let diagcat & cat(s,t,_,_) ==
                                          cat_of_diagrams(C) in
               let _,univ == colimit(s(diagmor)) in
               let tcone,_ == colimit(t(diagmor)) in
             univ(left_compose(diagcat,diagcat,I(diagcat))
                 (diagmor,tcone))) )
```

Here 'left_compose' composes a diagram morphism into the base of a cone with the cone (treated as a diagram morphism itself) to give a new cone. We can also define a functor which takes a diagram to the colimiting object on the diagram. It is given rather neatly as a composition of functors:

```
    dec colimit_apex_functor : Colimit_Cat(o,m) ->
            Functor(Diagram(o,m),Diagram_Mor(o,m),o,m)

    --- colimit_apex_functor(CC) <=
            colimit_functor(CC).apex_functor
```

A SHORT-CUT TO THE COMPUTATION OF COLIMITS

Thus far, we have set up data types to correspond to the definition of colimits, yet we have given no example of a category with its colimits. The problem is this: Colimits on arbitrary (finite) diagrams are awkward creatures to handle. The category of sets is cocomplete - has colimits of arbitrary diagrams - but to express an arbitrary (finite) colimit computationally is a lengthy process. With more complicated cocomplete categories it it is out of the question. However, what is feasible is to encode the colimits of diagrams of small, fixed shape. For instance we may encode coproducts of two objects, coequalisers or pushouts. What we then need is a means of extending these simple colimits to those of arbitrary (finite) diagrams. This can be achieved by "colimit existence" theorems. We shall give one such theorem and show how it translates into a program. We will then show how this short-cut to the computation of colimits works in the case of the category of sets.

The theorem that we intend to encode is:

Theorem (Colimit Existence)

If a category has an initial object, coproducts of pairs of objects and coequalisers of parallel pairs of morphisms then it has all finite colimits.

We will not give a proof of the theorem partly because it can be found in category theory texts (e.g. [Herrlich,Strecker 1973]) but also because our program translates the proof into HOPE text omitting only the verification of properties of constructed objects. In this sense the proof is constructive - given a means of computing initial objects, coproducts of pairs and coequalisers, the proof offers a means of computing arbitrary finite colimits.

Before looking at the construction, we use this theorem as an excuse for introducing some further data types. The colimits mentioned in the theorem, remembering that we include the universal parts along with the colimiting cone, can be given as data types:

```
    type InitialObj(o,m) == o # (o->m)

    type Coproduct(o,m) == o#o -> (o#m#m) # (o#m#m->m)

    type Coequaliser(o,m) == m#m -> (o#m) # (o#m->m)
```

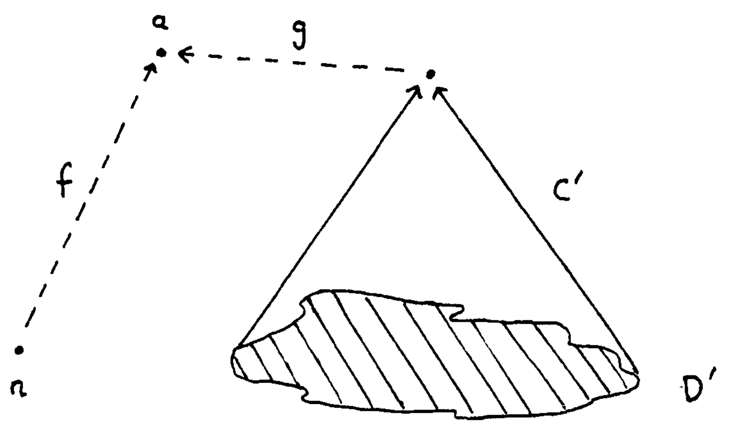
A category with these operations upon it is a new type, which by virtue of the theorem we may call a cocomplete category:

```
    data C_Cat(o,m) == c_cat( Cat(o,m),
                              InitialObj(o,m),
                              Coproduct(o,m),
                              Coequaliser(o,m))
```

The idea behind the construction is that given a finite diagram D we build the colimit stepwise, at each step using only the colimits available (the initial object, coproduct of pairs of objects and coequalisers of pairs of morphisms).

The first step is to take the coproduct of all the objects at the nodes in the diagram (ignoring the morphisms in the diagram). This can be done stepwise using the binary coproduct.

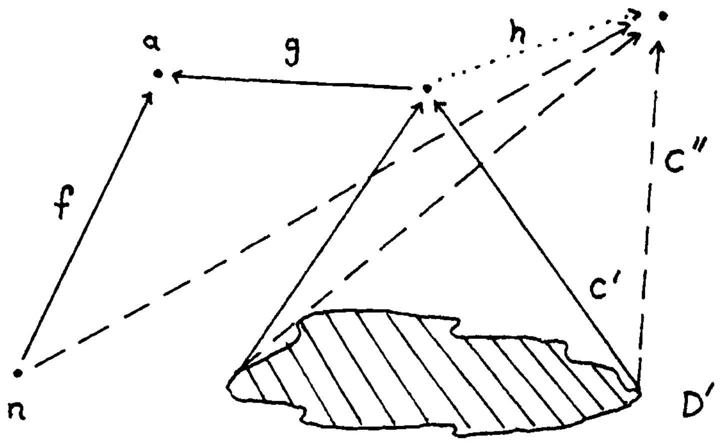
We begin by taking the coproduct of the empty diagram, which is the initial object. We then iterate through the nodes of the diagram. At each step we are given the coproduct of the objects at the nodes so far considered together with another node. The step is to construct a new colimiting cone by taking the binary coproduct of the apex of the colimiting cone that we are given and the object at the new node. A picture may help:



The new cone is constructed on the diagram, D', together with the new node and has the object 'a' as apex and as sides either f or a composition of sides of cone C' followed by g.

The universal part is computed along with the colimiting cone. We first construct the universal part of the initial cone as a function which takes a cone and gives a cone morphism. The cone morphism has as apex morphism the unique morphism from the initial object and as diagram morphism the unique morphism from the empty diagram.

Consider the picture above and let us try to construct a new universal part, given a universal part for the cone C' and also for the binary coproduct of b and c. Thus given any cone C" on the diagram D' augmented with the extra node, we have a picture as follows:



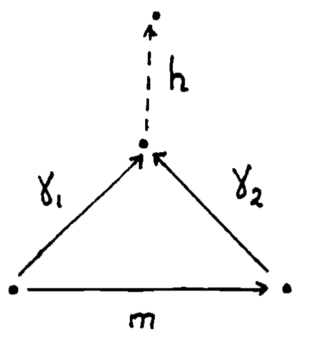
This cone C" is, by restriction of the base, a cone on the diagram D' and so the universal part of C' provides a cone morphism from C' to C" (on the smaller base). Thus there is a morphism h from the apex of C' to that of C". Now the binary coproduct comes into play since we have morphisms from b and c to the apex of C". Thus the universal part of the coproduct provides a morphism u : a -> c". This is the required lifting of the universal part.

As HOPE code, this is the routine "multicoprod":

```
dec multicoprod : C_Cat(o,m) ->
   (Diagram(o,m) -> Cone(o,m) # (Cone(o,m) -> Cone_Mor(o,m)))

--- multicoprod(k & c_cat(C & cat(_,_,id,cmp),init,cp,_)) <=
        lambda d & diagram((E,p,N),fo,fm) =>
            (let i,univinit == init in
             let initialcone ==
                         cone(C)(nil_diagram,nil_fn,i) in
        initialcone,
        (lambda pcone =>
                  cone_mor(C)(initialcone,
                              univinit(apex(pcone)),
                              pcone)) )
        if N=nil_set else
           (let n,N1 == singleton_split(N) in
            let c_cone,univc ==
                  multicoprod(k)(diagram((E,p,N1),fo,fm)) in
            let (a,f,g),univcp ==
                        cp(fo(n),apex(c_cone)) in
            let resultcone ==
              cone(C)( d,
                       (lambda m =>
                          f if m=n else
                                cmp(sides(c_cone)(m),g)),
                       a)  in
            let univpart ==
              (lambda pcone =>
                 let h == apex_functor ofm univc(pcone) in
              cone_mor(C)(resultcone,
                          univcp( apex(pcone),
                                  sides(pcone)(n),
                                  h ),
                          pcone )) in
        (resultcone,univpart)  )
```

What about the edges in the diagram and the associated morphisms? The edges are taken into account in the definition of a cone only in the fact that certain triangles commute. When we add edges to the diagram we must ensure that the appropriate commutation condition holds. This can be done by using the coequaliser to construct a new cone. Consider a picture,



where m is a morphism in the diagram, γ_1 and γ_2 are sides of a "cone" but with $m\gamma_2$ not necessarily equal to γ_1. Now if we take the coequaliser of $m\gamma_2$ and γ_1 (h in the above picture) we can construct a new "cone" with sides $\gamma_1 h$ and $\gamma_2 h$ such that $\gamma_1 h = m\gamma_2 h$ by definition. So, for each edge in the diagram, we construct such a coequaliser and thus a new cone, remembering of course to lift the universal part through the construction. The computation of the universal part is very much like that of the previous case. This construction is encoded as the routine "addedge":

```
dec addedge : C_Cat(o,m) ->
    (( Cone(o,m) # ( Cone(o,m) -> Cone_Mor(o,m) ) ) # Edge
           -> Cone(o,m) # ( Cone(o,m) -> Cone_Mor(o,m) ) )

--- addedge(c_cat(C, _, _, ce)) <=
        lambda (cone1,puniv), e =>
           let diagram(g,fo,fm) == base(cone1) in
           let (b,h),univ ==
             ce( sides(cone1)(dom(g)(e)),
                 compose(C)( fm(e),
                             sides(cone1)(range(g)(e))) ) in
           let result_graph ==
             graph(e::edges(g),graph_map(g),nodes(g)) in
           let resultdiag == diagram(result_graph,fo,fm) in
           let resultcone ==
             cone(C)( resultdiag,
                      (lambda p =>
                         compose(C)(sides(cone1)(p),h)),
                       b ) in
           let universal == (lambda pcone =>
             let univ_mor == puniv(pcone) in
           cone_mor(C)( resultcone,
                        univ(apex(pcone),apex_mor(univ_mor)),
                        pcone) ) in
        (resultcone,universal)
```

The colimit on an arbitrary finite diagram is calculated recursively by splitting off an edge of the diagram if there are any (if not then "multicoprod" is called), recursively computing the colimit on the remaining diagram and then using "addedge" to add the extra edge.

```
    dec finite_colimit : C_Cat(o,m) -> Colimit(o,m)

    --- finite_colimit(CC) <=
            (lambda d & diagram((E,mor(_,f,xN),N),fo,fm) =>
               (let (_,gamma,a),univ == multicoprod(CC) d in
               (d,gamma,a), univ  )  if E=nil_set
                                     ! coproduct if no edges
              else
              (let e,E1 == singleton_split(E) in
        addedge(CC)
            ( finite_colimit(CC)
                  (diagram((E1,mor(E1,f,xN),N),fo,fm) ),
              e )) )         ! run through edges coequalising
```

This is one of the major routines of the program. Its fascination is partly that it is a direct encoding of the proof of the "existence of colimits" theorem. Moreover, as we shall demonstrate, it makes the computing of colimits in categories at least tractable for we now need to construct only initial objects, binary coproducts and coequalisers. The application of the above routine will then compute any finite colimit for us.

Notice that a proof of the program needs to establish not only the properties of "finite_colimit" but also its well-definedness (e.g. that it is independent of the order in which "addedge" is used to add edges). This follows, of course, from standard category theoretic results.

COLIMITS IN THE CATEGORY OF SETS

We are now in a position to construct arbitrary finite colimits in the category of sets. We use the program above and so need the initial object, binary coproducts and coequalisers of parallel pairs of morphisms in the category of sets.

The initial object is the empty set, "nil_set". Its universal part is the unique morphism to any other set from the empty set:

```
    dec nil_mor : Set(alpha) -> Set_Mor(alpha)
    --- nil_mor(x) <= mor(nil_set,nil_fn,x)
```

("nil_fn" is a function without a definition, in fact any function would do e.g. the identity.) So the initial object together with its universal prt is:

```
    dec init : InitialObj(Set(Tag alpha),Set_Mor(Tag alpha))
    --- init <= nil_set,nil_mor
```

The coproduct of two sets is their disjoint union (together with a universal part). The disjointness of the union is ensured, as usual, by a tagging of the elements to make the elements of different sets distinct elements in the coproduct set. We thus introduce two new tags, "pink" and "blue" (or "girl" and "boy"):

```
    Tag(alpha) == ... ++ pink(Tag(alpha)) ++ blue(Tag(alpha)) ...
```

The idea now is to construct a new set consisting of the elements of one set "pinked" and of the other set "blued". The morphisms into this colimiting set are essentially the colouring operations. Given any other cone on the two sets, the universal part of the coproduct is the morphism from the disjoint union which removes the colour from an object and then uses the morphisms in this given cone. As a program, the coproduct looks like this:

```
    dec coprod : Coproduct(Set(Tag alpha),Set_Mor(Tag alpha))

    --- coprod(s,t) <= let u == (pink*s) U (blue*t) in
                       (u, mor(s,pink,u), mor(t,blue,u)),
                       (lambda v,mor(a,f,b),mor(c,g,d)=>
                               undef if not(v=b and v=d and a=s and c=t)
                               else let fg == (lambda pink(x) => f(x)
                                                    | blue(x) => g(x)) in
                                      mor(u,fg,v) )
```

("lambda ... => ... | ... => ... etc" is a lambda expression with cases.)

All that remains now is to compute the coequaliser of a parallel pair of set morphisms. This is usually given as a set of equivalence classes. Instead, to make the computation simpler, we choose elements of the classes to represent them. This routine uses a rather elegant recursion. Suppose that we want to calculate the coequaliser of f,g : S -> T. First split S into two pieces, P and Q, (if not possible then the coequaliser is trivial), and recursively compute the coequaliser of f and g restricted to P to get:

$$\begin{tikzcd} P \arrow[r, shift left, "f|_P"] \arrow[r, shift right, "g|_P"'] & T \arrow[r, "u"] & U \end{tikzcd}$$

To include the remainder of S, namely Q, we notice that there are morphisms

f'u, g'u : Q -> U,

where f' and g' are the restrictions of f and g to Q. We recursively compute the colimit of these two morphisms, to arrive at:

$$\begin{tikzcd} Q \arrow[r, shift left, "f'"] \arrow[r, shift right, "g'"'] & T \arrow[r, "u"] & U \arrow[r, "h"] & R \end{tikzcd}$$

The coequaliser of f,g is then the morphism 'u.h' together with a universal part which lifts from the two coequalisers used. In code this looks like:

```
    dec coeq : Coequaliser(Set(Tag alpha),Set_Mor(Tag alpha))

    --- coeq(mor(S,f,T),mor(R,g,V)) <=
                                      ! the undefined case
       (nil_set, undef), (lambda _,_=>undef) if not(S=R and T=V)
       else                           ! the empty case
       (let cat(_,_,id,comp) == cat_of_sets in
        (T, id(T)), (lambda _,j=>j) if S=nil_set else
                                      ! if S has one object
        (let x,_ == singleton_split(S) in
         ( (T,id(T)), (lambda _,j=>j) if f(x)=g(x)
           else
           let W == T - {g(x)} in    ! target of coeq
           (W, mor(T,(lambda y=>f(x) if y=g(x) else y),W)),
           (lambda a,mor(_,j,_)=>mor(W,j,a)) ) )
       if cardinal(S)=1 else
        (let P,Q == split(S) in      ! the recursive case
         let (_,u),univ == coeq(mor(P,f,T),mor(P,g,T)) in
         let (R,h),univ1 ==
            coeq(comp(mor(Q,f,T),u),comp(mor(Q,g,T),u)) in
         (R, comp(u,h)), (lambda a,j=>univ1(a,univ(a,j))) ) )
```

The cocomplete and colimit categories of sets then look like:

```
    dec c_cat_of_sets : C_Cat(Set(Tag alpha),Set_Mor(Tag alpha))
    --- c_cat_of_sets <= c_cat(cat_of_sets,init,coprod,coeq)

    dec colimit_cat_of_sets :
           Colimit_Cat(Set(Tag alpha),Set_Mor(Tag alpha))
    --- colimit_cat_of_sets <=
           colimit_cat(cat_of_sets,finite_colimit(c_cat_of_sets))
```

We now turn to techniques by which this explicit computation of colimits for the category of sets can, without any further computation of colimits, be extended to the category of, say, graphs or of signatures or of more complex data types.

COLIMITS AND CATEGORY CONSTRUCTORS

Let us, for the moment, have a look at colimits in the category of graphs, Graph. As we have seen, the comma category (Set,X), where 'X' is the cross-product functor, is isomorphic to the category of graphs. Can we use this fact to simplify the computation of colimits of graphs?

First notice that there are two functors - the projection functors arising from the comma category:

edges, nodes : Graph -> Set

Given a diagram of graphs we can, by applying these functors to the objects and morphisms in the diagram, obtain two diagrams in Set, one of the edges of the graphs and the other of the nodes. The colimit object of the diagram of graphs is a graph with the following property: its edges are given by the colimit object of the diagram of edges, and the nodes likewise. The morphism from edges to pairs of nodes can also be constructed from these colimits in Set as we shall now show in a general setting. Thus we can reduce the calculation of colimits of graphs to those of sets provided that we have the means available to "lift" these colimits through to those of graphs.

We now look at this in the more general setting of right comma categories. We state a theorem (brought to our attention by Prof. Joe Goguen) which says that colimits in right comma categories arise from those in the categories on which the comma category is built. We give a detailed proof from first principles of this theorem and then code this into a routine to lift colimits. To emphasize how close the proof is to a program, labels appear in the proof to match corresponding labels in the program. Thus the proof is a commentary on the program. More formally it ought to be the kernel of a proof of program correctness (with respect to some semantics).

Theorem

If $\underline{A}$ and $\underline{B}$ are cocomplete categories and $F : \underline{B} \to \underline{A}$ is any functor, then $(\underline{A},F)$ is cocomplete.

We could restrict the cocompleteness to, say, finite cocompleteness or to the existence of colimits of a specified type, e.g. pushouts.

Proof

Let D be a diagram in $(\underline{A},F)$ with nodes n in the set N, and objects

$d_n = (a_n,\ f_n\colon a_n \to b_nF,\ b_n)$.

A morphism in the diagram is a morphism in $(\underline{A},F)$ of the form;

$$\begin{tikzcd}
a_m \arrow[d, "g"'] & a_m \arrow[r, "f_m"] \arrow[d, "g"'] \arrow[dr, phantom, "(*)"] & b_mF \arrow[d, "hF"] & b_m \arrow[d, "h"] \\
a_n & a_n \arrow[r, "f_n"'] & b_nF & b_n
\end{tikzcd} \qquad (1)$$

where (*) commutes.

There are projection functors,

$P_A : (\underline{A},F) \to \underline{A},\ P_B : (\underline{A},F) \to \underline{B}$.

These functors extend to functors on diagrams, D_A and D_B.

Let $\zeta_n : a_n \to a$ be a colimiting cone on the diagram $D_A(D)$ in $\underline{A}$, then if $g\colon a_m \to a_n$ is a morphism in $D_A(D)$

$$\begin{tikzcd}
a_m \arrow[rr, "g"] \arrow[dr, "\zeta_m"'] & & a_n \arrow[dl, "\zeta_n"] \\
& a &
\end{tikzcd} \qquad (2)$$

commutes.

Likewise, let $\gamma_n : b_n \to b$ be a colimiting cone on $D_B(D)$ in $\underline{B}$, then if $h : b_m \to b_n$ is a morphism in $D_B(D)$,

$$\begin{tikzcd} b_m \arrow[rr, "h"] \arrow[dr, "\gamma_m"'] & & b_n \arrow[dl, "\gamma_n"] \\ & b & \end{tikzcd} \qquad (3)$$

commutes.

Thus as F is a functor the following is a cone in $\underline{A}$:

$\gamma_n F : b_n F \to bF.$ (4)

Consider the following cone in $\underline{A}$, with base $D_A(D)$:

$f_n . \gamma_n F : a_n \dashrightarrow b_n F \dashrightarrow bF.$ (5)

It is a cone because, if $g : a_m \to a_n$ is a morphism in $D_A(D)$, then this morphism is a left projection via P_A of a morphism of the form (1), hence

$$\begin{tikzcd} a_m \arrow[r, "f_m"] \arrow[d, "g"'] \arrow[dr, phantom, "(*)"] & b_m F \arrow[d, "h"] \arrow[dr, "\gamma_m F"] \arrow[d, phantom, shift left=6, "(**)"] & \\ a_n \arrow[r, "f_n"'] & b_n F \arrow[r, "\gamma_n F"'] & bF \end{tikzcd}$$

commutes, because (*) is the condition that (g,h) is a morphism in the comma category and (**) commutes because γ_n is a cone.

Thus, by the colimiting property of $\zeta_n : a_n \to a$, there is a unique morphism

$f : a \to bF$ (6)

such that

$$\begin{tikzcd}
a_n \arrow[r, "f_n"] \arrow[d, "\zeta_n"'] & b_n F \arrow[d, "\gamma_n F"] \\
a \arrow[r, "f"'] & b F
\end{tikzcd} \tag{6a}$$

commutes for all n.

(6a) together with (2) and (3) state that

$$(\zeta_n, \gamma_n) : (a_n, f_n, b_n) \to (a, f, b) \tag{7}$$

is a cone on D.

We show that it is the colimiting cone. Let $(p_n, q_n) : (a_n, f_n, b_n) \to (a', f', b')$ be any other cone on D so that for any morphism of the form (1):

$$\begin{tikzcd}
a_m \arrow[rr, "g"] \arrow[dr, "p_m"'] & & a_n \arrow[dl, "p_n"] \\
& a' &
\end{tikzcd} \qquad \begin{tikzcd}
b_m \arrow[rr, "h"] \arrow[dr, "q_m"'] & & b_n \arrow[dl, "q_n"] \\
& b' &
\end{tikzcd} \tag{7a}$$

both commute for all n, and

$$\begin{tikzcd}
a_n \arrow[r, "f_n"] \arrow[d, "p_n"'] & b_n F \arrow[d, "q_n F"] \\
a' \arrow[r, "f'"'] & b' F
\end{tikzcd} \tag{7b}$$

commutes for all n. Thus by the colimiting properties of ζ_n and γ_n we have unique morphisms:

α : a -> a' in A and (8)

β : b -> b' in B (9)

such that, for all n in N,

$$\begin{tikzcd} & a_n \arrow[dl, "\zeta_n"'] \arrow[dr, "p_n"] & \\ a \arrow[rr, "\alpha"'] & & a' \end{tikzcd} \qquad \begin{tikzcd} & b_n \arrow[dl, "\gamma_n"'] \arrow[dr, "q_n"] & \\ b \arrow[rr, "\beta"'] & & b' \end{tikzcd}$$ (9a)

both commute. Then

$$\begin{tikzcd} a \arrow[r, "f"] \arrow[d, "\alpha"'] & bF \arrow[d, "\beta F"] \\ a' \arrow[r, "f'"'] & b'F \end{tikzcd}$$ (9b)

commutes from (6), (7a) and (9a) and the colimiting property of ζ_n : a_n -> a. That this morphism,

(α : a -> a', β: b -> b'), (10)

is the unique morphism from (a,f,b) to (a',f',b') such that (9a) and (9b) both hold follows directly from the uniqueness properties of α and β separately. $\square$

Implementation of the Lifting of Colimits

We now turn to the encoding, in HOPE, of this construction.

In the proof we refer to several functors which are extensions of functors to diagrams and to cones. For example the functor D_A may be programmed as:

```
    dec left : Cat(o,m)#Cat(o1,m1) ->
                 (Diagram(o#m#o1,Right_Comma_Mor(o,m,o1,m1)) ->
                    Diagram(o,m) )

    --- left(_,_) <= lambda diagram(gr,fo,fm) =>
                diagram( gr,
                         (lambda x => (left ofo fo(x))),
                         (lambda y => (left ofm fm(y))) )
```

The routine for the construction of colimits in a comma category is parameterised on the colimits in the two categories from which the comma category is built;

```
    dec lift_colimit:
            Colimit_Cat(o1,m1) #
            Functor(o2,m2,o1,m1) #
            Colimit_Cat(o2,m2) ->
                 Colimit(o1#m1#o2, Right_Comma_Mor(o1,m1,o2,m2))
```

and is defined by:

```
--- lift_colimit(cA & colimit_cat(A,colimA),
                 F,
                 cB & colimit_cat(B,colimB)) <=
        let dcat == cat_of_diagrams(A)  in
        let commacat == right_comma_cat(A,F,B) in
    lambda D & diagram(_,fo,_) =>
        let coneA, univA == colimA(left(A,B)(D)) in         (2)
        let coneB, univB == colimB(right(A,B)(D)) in        (3)
        let FconeB  == apply(B,A)(F,coneB) in               (4)
        let pretendcone ==                                  (5)
          left_compose(dcat,dcat,I(dcat))
                   (diagram_mor(base(coneA),
                                ident,
                                ident,
                                (lambda n => fc
                                  where _,fc,_ == fo(n)),
                                base(FconeB)),
                   FconeB ) in
        let u == apex_morphism(univA(pretendcone)) in       (6)
        let colim_obj == (apex(coneA), u, apex(coneB) ) in
        let resultcone ==                                   (7)
          cone(commacat)
              (D,
               (lambda n =>
                 comma_mor(fo(n),
                           (sides(coneA)(n),sides(coneB)(n)),
                           colim_obj) ),
               colim_obj) in
        let universal ==
          (lambda pcone =>
        let uA ==                                           (8)
               apex_morphism(univA(left(A,B)(pcone))) in
        let uB ==                                           (9)
               apex_morphism(univB(right(A,B)(pcone))) in
           cone_mor(commacat)                               (10)
               ( resultcone,
                 comma_mor(colim_obj,(uA,uB),apex(pcone)),
                 pcone) )  in
    (resultcone,universal)
```

The steps in the proof may be followed (with the help of the corresponding of labels) to understand the text of the program above.

How do we put this to use? Access to the procedure is through a colimit routine in a new data type 'colimit_comma_category'. That is, we produce a comma category along with its colimit routine:

```
    dec colimit_comma_cat :
           Colimit_Cat(o1,m1) #
           Functor(o2,m2,o1,m1) #
           Colimit_Cat(o2,m2) ->
             Colimit_Cat((o1#m1#o2),Right_Comma_Mor(o1,m1,o2,m2))

    --- colimit_comma_cat( kA & colimit_cat(A,_),
                           F,
                           kB & colimit_cat(B,_))  <=
        colimit_cat(right_comma_cat(A,F,B),lift_colimit(kA,F,kB))
```

Now, we can immediately compute the colimits of diagrams in data types which can be expressed as comma categories, as long as we have routines to compute colimits in the categories on which the comma category is built.

Graphs serve as a good example. We can define the colimit category of graphs - which contains a colimit routine for graphs - as:

```
    dec colimit_cat_of_graphs :
           Colimit_Cat(Graph(Tag alpha), Graph_Mor(Tag alpha))

    --- colimit_cat_of_graphs <=
           colimit_comma_cat( colimit_cat_of_sets,
                              cross_product,
                              colimit_cat_of_sets)
```

Likewise, colimits of signatures:

```
    dec colimit_cat_of_signatures :
           Colimit_Cat( Signature(Tag alpha),
                        Signature_Mor(Tag alpha) )

    --- colimit_cat_of_signatures <=
           colimit_comma_cat( colimit_cat_of_sets,
                              strings,
                              colimit_cat_of_sets)
```

Lifting techniques like this are not new. For instance, many languages have the facility to "lift" an equality on a type, 'alpha', through to constructed types such as 'Set(alpha)' (set comprehension) or to maps 'alpha->alpha'.

What about left comma categories? The above theorem does not hold if we simply replace "right comma category" with "left comma category". We need the extra requirement that the functor preserves colimits:

Theorem

If L : $\underline{A}$ -> $\underline{B}$ is (finitely) cocontinuous and $\underline{A}$ and $\underline{B}$ are (finitely) cocomplete then (L,$\underline{B}$) is (finitely) cocomplete.

We could encode the construction used in the proof of this theorem, very much as we did for the right comma category. We shall not, however, be needing it. These two theorems embody a more general concept which is useful in describing the computational aspects of theorems of this form - that of the 'creation' of colimits. Before turning to this we will look briefly at colimits in functor categories.

Colimits in Functor Categories

Functor categories lend themselves to a treatment similar to that of comma categories. Colimits in categories of functors arise from colimits in the target category of the functors. The passage from colimits in the target category to colimits of functors is described as the 'pointwise' computation of colimits; that is, for each object of the source category, a diagram of functors gives, by application to this object, a diagram in the target category. By suitable manipulation of the colimits of these diagrams the colimit of the diagram of functors may be computed.

If $\underline{B}^{\underline{A}}$ denotes the category of all functors from $\underline{A}$ to $\underline{B}$, we have the theorem:

Theorem (Colimits in Functor Categories)

If $\underline{B}$ is (finitely) cocomplete then so is $\underline{B}^{\underline{A}}$. □

As in the case of comma categories, the proof is constructive and may be translated into a program. Since we shall not need this program, we relegate it to an appendix (Appendix One).

CREATION

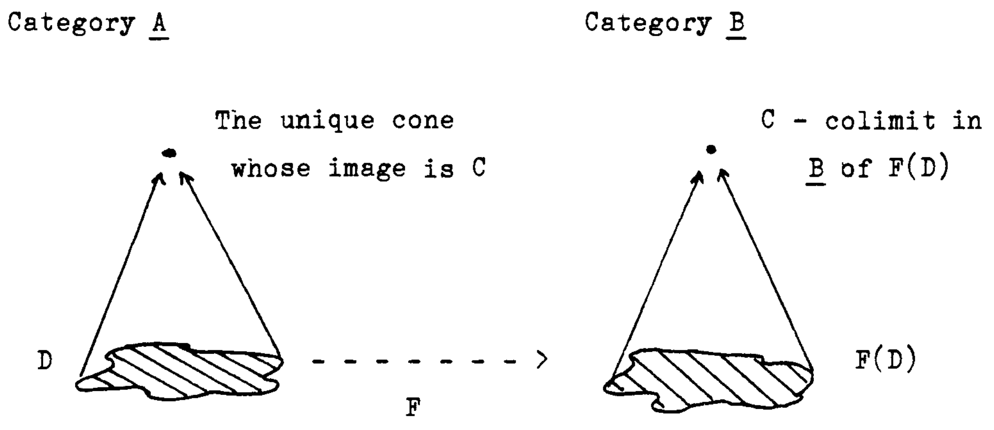
From the discussion of the colimits of graphs we see that the essence of the construction lies in the functors 'edges' and 'nodes', in particular their behaviour with respect to colimits.

Functors may preserve colimits. This property alone gives us no means of constructing colimits. Reflection of colimits is a similar non-constructive property of functors. There is, however, a means of describing the behaviour of the functors, 'edges' and 'nodes', which allows us to lift colimits through these functors. It is known as 'the creation of colimits' and is described in [Schubert 1972].

Definition

A functor F : A -> B is said to create colimits, if for any diagram D in A, there is a colimiting cone C in B on the image of D under F, such that, there is a unique cone C' in A with base D and such that the image of C' under F is C and, moreover, C' is a colimiting cone on D. ▯

A picture may be helpful:



Thus, a constructive proof that a functor creates colimits in A from those in B will give a means of computing the colimits in A given a means of computing those in B. This is exactly what we have done in the case of colimits in comma categories where the proof that 'left x right' (the product of the projection functors) creates colimits is encoded as a program. Other examples of creation will occur in the following chapters.

There are several notions along the same lines as the creation of colimits but somewhat weaker:

Definition

A functor F : A -> B is said to lift colimits uniquely, if for any diagram D in A, there is a colimiting cone C in B on the image of D under F, such that, there is a unique colimiting cone C' in A with base D and such that the image of C' under F is C. □

Notice that the only difference between this and the creation of colimits is that here we can say only that there is a unique colimiting cone, not a unique cone. Thus this is rather weaker. Computationally it says that part of the work of computing colimits in A can be done by calling a routine for computing colimits in B and that although more work is required to compute the colimit in A no further colimit computations will be of any use (because a choice of a colimit in B determines a colimit in A).

Many forgetful functors lift colimits uniquely-for example that from partial-orders to the underlying sets. A computational example of the unique lifting of colimits will be found on page 69.

A weaker notion still is the 'lifting of colimits' dropping the requirement of uniqueness. This is then equivalent to A having colimits and their being preserved by F. The functors 'edges' and 'nodes' separately (or more generally 'left' and 'right') lift colimits but not uniquely.

INFINITE COLIMITS

We can handle colimits of infinite diagrams in the same way as those of finite diagrams but we must be careful to avoid non-terminating routines and attempts to compute infinite objects.

Colimits of infinite diagrams, especially those of ω-chains, will of use in the next few chapters. For instance, they arise when we pass from signatures to free theories. More generally they provide a categorical formulation of program iteration.

An ω-chain in a category C, is a diagram of the form:

$$a_0 \to a_1 \to a_2 \to \dots a_n \to a_{n+1} \to \dots \qquad - (*)$$

As a data type we can describe them by:

```
    type w_Chain(o,m) == (Num -> o) # (Num -> m)
```

Notice that this is not itself a diagram. However, each such ω-chain gives rise to a diagram:

```
    dec w_diagram : w_Chain(o,m) -> Diagram(o,m)
    --- w_diagram(omap,mmap) <=
            diagram( w_graph,
                     (lambda just(number n) => omap(n)),
                     (lambda just(number n) => mmap(n)) )
```

where 'ω-graph' is the underlying graph of an ω-chain:

```
    dec w_graph : Graph(Name)
    --- w_graph <=
            let f ==
             (lambda just(number(n)) =>
                 pair(just(number(n)),just(number(n+1))) ) in
         (bigset,mor(bigset,f,bigset),bigset)
```

All the sets are infinite and so are given by the undefined constant set 'bigset'. Attempts to evaluate this set will fail.

We can define morphisms of ω-chains hence define a category of ω-chains. A morphism of ω-chains from ω-chain (*) to the following chain,

$b_0 \to b_1 \to b_2 \to \ldots b_n \to b_{n+1} \to \ldots$

is a sequence of $\underline{C}$-morphisms, $\langle h_n : a_n \to b_n \; n \geq 0 \rangle$, such that for all $n \geq 0$ the following square commutes:

$$\begin{tikzcd}
a_n \arrow[r, "f_n"] \arrow[d, "h_n"'] & a_{n+1} \arrow[d, "h_{n+1}"] \\
b_n \arrow[r, "g_n"'] & b_{n+1}
\end{tikzcd}$$

As a data type such a morphism is simply:

```
    type w_Chain_Mor(o,m) ==
                      w_Chain(o,m) # (Num -> m) # w_Chain(o,m)
```

and the category of ω-chains is:

```
dec cat_of_chains : Cat(o,m) ->
        Cat(w_Chain(o,m),w_Chain_Mor(o,m))

--- cat_of_chains(cat(_,_,id,comp)) <=
        cat( (lambda (s,_,_) => s),
             (lambda (_,_,t) => t),
             (lambda wc & (omap,_) =>
                (wc,(lambda n => id(omap(n))),wc)),
             (lambda (wc1,m1,_),(_,m2,wc3) =>
                (wc1,(lambda n => comp(m1(n),m2(n))),wc3)))
```

Now the function 'ω-diagram' can be extended to a functor - morphisms of ω-chains become diagram morphisms.

```
dec w_diagram_functor :
        Functor( w_Chain(o,m),w_Chain_Mor(o,m),
                 Diagram(o,m),Diagram_Mor(o,m))
--- w_diagram_functor <=
        functor( w_diagram,
                 (lambda s,f,t =>
                     diagram_mor( w_diagram(s),
                                  ident,
                                  ident,
                                  (lambda just(number(n))
                                     => f(n)),
                                  w_diagram(t))) )
```

An ω-colimit is a colimit of an ω-chain:

```
type w_Colimit(o,m) == w_Chain(o,m) ->
        (Cone(o,m) # (Cone(o,m) -> Cone_Mor(o,m)))
```

In the same way that we defined cocomplete categories as data types, we can introduce ω-cocomplete categories.

```
data w_Cocomplete_Cat(o,m) ==
        w_cocomplete_cat(Cat(o,m),w_Colimit(o,m))
```

In the case of cocomplete categories the construction of a colimit category required the extension of initial object, coproduct and coequaliser to arbitrary finite colimits. The coercion of an ω-cocomplete category to a colimit category requires no such elaborate work since an ω-colimit differs from a colimit only in that its argument is a ω-chain and not a diagram. Thus, if we have a function which takes a diagram of the form (*) into a ω-chain:

```
dec diag_to_chain : Diagram(o,m) -> w_Chain(o,m)
--- diag_to_chain(diagram(_,fo,fm)) <=
        ( (lambda n => fo(just(number n))),
          (lambda n => fm(just(number n))) )
```

then a ω-cocomplete category gives rise to a category together with a colimit function on it:

```
    dec w_colimit_cat : w_Cocomplete_Cat(o,m) -> Colimit_Cat(o,m)
    --- w_colimit_cat(w_cocomplete_cat(C,w_colim)) <=
            colimit_cat(C, diag_to_chain.w_colim)
```

So far the work has been the setting up of the types required to handle colimits of ω-chains. We have not provided any means of computing ω-colimits.

In general, even for the category of sets, computation of ω-colimits is not possible - requiring non-terminating steps. However there are simple cases of ω-chains for which this computation is possible and indeed arises commonly in applications.

Firstly, a trivial remark: in any category, if an ω-chain eventually becomes constant (i.e. the morphisms in the chain eventually become the identity on an object) then the chain has a colimit. The colimiting object is then the object in the chain at which the chain becomes constant. Whilst mathematically trivial, computationally this is an important case of ω-colimits as it gives a categorical interpretation of program iteration. We shall use this to cast iterative programs into a categorical form so that the iteration inherent in the program is translated into the computation of a colimit (see page 140).

To compute colimits of these, eventually constant, chains, we first find when they become constant:

```
    dec fixed_point : Cat(o,m) ->
                       (w_Chain(o,m) -> (Num -> Num))

    --- fixed_point(C)  <=
            lambda w_chain & (fo,fm) =>
               (lambda n =>
                  n if fm(n) = identity(C)(fo(n))
            else fixed_point(C)(w_chain)(n+1) )
```

The termination condition is an equality of morphisms. We can either assume that there is a predefined equality or include equalities for objects and morphisms as part of the definition of a category. In the latter case the equality may not always be computable (for instance, in the category of functors). Notice further that we assume that the first occurrence of an identity in a chain is the point at which all succeeding morphisms in the chain are identities. For the applications we have in mind this is indeed the case. The colimiting object is then the object at which the chain becomes constant.

The colimiting cone has sides that are the identity for objects beyond the point at which the chain becomes constant and are multiple compositions for objects before this point:

```
    dec w_cone_sides : Cat(o,m) ->
            (w_Chain(o,m) -> (Num -> (Num -> m)))

    --- w_cone_sides(C)  <=
            lambda w_chain & (fo,fm) =>
               (lambda fixpoint => (lambda n =>
                    identity(C)(fo(n)) if n>=fixpoint
               else compose(C)
                     ( fm(n),
                       w_cone_sides(C)(w_chain)(fixpoint)(n+1))))
```

The ω-colimit is then given by:

```
    dec finite_w_colimit : Cat(o,m) -> w_Colimit(o,m)

    --- finite_w_colimit(C) <=
            lambda w_chain & (fo,fm) =>
               let d == w_diagram(w_chain) in
               let N == fixed_point(C)(w_chain)(0) in
               let colim_cone ==
                    cone(C)( d,
                             (lambda just(number(n)) =>
                                w_cone_sides(C)(w_chain)(N)(n)),
                             fo(N) ) in
               let univ ==
                  (lambda pcone =>
                    cone_mor(C)( colim_cone,
                                 sides(pcone)(just(number(N))),
                                 pcone ) ) in
            (colim_cone, univ)
```

Categories with such ω-colimits are given by:

```
    dec finite_w_cocomplete_cat :
            Cat(o,m) -> w_Cocomplete_Cat(o,m)
    --- finite_w_cocomplete_cat(C) <=
            w_cocomplete_cat(C, finite_w_colimit(C))
```

These can be translated to colimit categories by:

```
    dec finite_w_colimit_cat :
            Cat(o,m) -> Colimit_Cat(o,m)
    --- finite_w_colimit_cat(C) <=
            w_colimit_cat(finite_w_cocomplete_cat(C))
```

We now turn to computation of non-trivial ω-colimits in the category of sets. As mentioned, we cannot, in general, compute such colimits, but there is an important special case which is computable and occurs widely - particularly in the construction of initial algebras and free theories (see page 125). This is the case when the morphisms in the ω-chains are monics, which means, in the category of sets, that the morphisms are 1-1.

Suppose that

$$a_0 \to a_1 \to a_2 \to \dots a_n \to a_{n+1} \to \dots$$

is any ω-chain in the category of sets. The colimiting set is the disjoint union of all the a_n, $a = \bigcup_{n \in \omega} a_n$, quotiented by the reflexive, symmetric, transitive closure of the relation, $\sim$, given by:

If $x \in a_n$, $x \sim f_n(x)$.

If the f_n are all monics then the equivalence classes are all of the form

$$\{ x, f_n(x), f_{n+1} f_n(x), \dots \}$$

for some n and some $x \in a_n$.

In our computation we choose representatives of these equivalence classes, choosing the 'x' as a representative. We need to record the n as well so we introduce a new tag:

```
    data Tag(alpha) == ..... ++ origin(Tag(Label),Tag(alpha))
```

tagging an object with the label of the set from which it comes (the type 'Label' is defined on page 30).

Now, in general, the colimiting set will be infinite. We assume that we are dealing with chains of sets of ever increasing cardinality. Let $\mathfrak{z}_n : a_n \to a$ be the colimit of the above chain with all the f_n monics. The following function calculates the $\mathfrak{z}_n$, assuming that the elements of the colimiting set are representatives of the equivalence classes (chosen as above and tagged with the label of the set in the chain from which they come).

```
dec first_occur :
        (w_Chain(Set(Tag alpha),Set_Mor(Tag alpha)) # Num)
                -> Set_Mor(Tag alpha)

--- first_occur(w_chain & (omap,mmap),n) <=
      mor( omap(0),
           (lambda x => origin(just(number(n)),x)),
           bigset )
                if n = 0 else
      mor(omap(n),
          (lambda x => origin(just(number(n)),x)
             if x is_in (omap(n) -
                          image omap(n-1) through mmap(n-1))
             else first_occur(w_chain,n-1) of
                             (inv(mmap(n-1))(x)) ),
          bigset)
```

Here 'inv' gives an inverse of a morphism (i.e. inv(m).m = identity morphism whenever the composition is defined). The multifix operation "image ... through ..." is the image of a set through a set morphism. This construction relies upon the factorisation of morphisms in the category <u>Set</u>.

The colimit of such an ω-chain is then given by:

```
    dec infinite_w_colimit :
        w_Colimit(Set(Tag alpha), Set_Mor(Tag alpha))

    --- infinite_w_colimit(w_chain) <=
        let d == w_diagram(w_chain) in
        let colim_cone ==
           cone(cat_of_sets)
               ( d,
                 (lambda just(number n) =>
                        first_occur(w_chain,n)),
                 bigset ) in
        let univ ==
           (lambda pcone =>
                cone_mor(cat_of_sets)
                        ( colim_cone,
                          mor( bigset,
                               (lambda origin(label,x) =>
                                   sides(pcone)(label) of x),
                               apex(pcone) ),
                          pcone) ) in
        (colim_cone,univ)
```

We can thus form an ω-cocomplete category of sets where the ω-chains are restricted to those in which the morphisms are all monics and in which the sets increase in cardinality indefinitely:

```
    dec infinite_w_cocomplete_cat_of_sets :
            w_Cocomplete_Cat(Set(Tag alpha),Set_Mor(Tag alpha))

    --- infinite_w_cocomplete_cat_of_sets <=
            w_cocomplete_cat(cat_of_sets, infinite_w_colimit)
```

This may be translated into a colimit category:

```
    dec infinite_w_colimit_cat_of_sets :
            Colimit_Cat(Set(Tag alpha),Set_Mor(Tag alpha))

    --- infinite_w_colimit_cat_of_sets <=
            w_colimit_cat(infinite_w_cocomplete_cat_of_sets)
```

With these ω-cocomplete categories, we can use the same lifting techniques as for finite colimits, thus constructing complex data types from simple ones whilst lifting through the ω-colimit constructors. For instance, we could define a category of graphs with colimits of certain ω-chains by:

```
    dec infinite_w_colimit_cat_of_graphs :
                    Colimit_Cat(Graph(alpha),Graph_Mor(alpha))

    --- infinite_w_colimit_cat_of_graphs <=
            colimit_comma_cat( infinite_w_colimit_cat_of_sets,
                               cross_product,
                               infinite_w_colimit_cat_of_sets )
```

In this discussion of ω-colimits there is something anomalous as can be seen most clearly in the function 'diag-to-chain'. Not every diagram can be coerced into a chain. This same problem actually occurred in our discussion of finite colimits - only finite diagrams can be arguments to the function 'finite-colimit' . The problem is essentially one of handling 'diagram schemes' - collections of diagrams, for instance, coproduct diagrams, diagrams obtained from ω-chains, or all finite diagrams - within the strong type discipline of HOPE. We have avoided the problem by such partial functions as mentioned above.

This ends our presentation of the basic category theory program. We now turn to applications of the routines we have programmed.

Notes

The programming described here (up to and including the lifting of colimits to comma categories) was a joint project between Prof. Rod Burstall, Don Sannella and myself as explained in chapter one. Prof. Joe Goguen [Goguen and Burstall 1978] brought our attention to the lifting of colimits to comma categories. Creation of colimits is standard category theory.

CHAPTER FOUR

IMPLEMENTING THE SEMANTICS OF A SPECIFICATION LANGUAGE

- An application of the programming of categories.

We claim to have to have powerful general routines which encode in programs part of category theory. What use are they?

Certainly, given any construction expressed in the language of category theory - more particularly, the category theory that we have displayed so far - we ought to have a direct means of translating this construction into a program. An example of this is the categorical semantics which has been developed for a specification language [Burstall,Goguen 1980a]. We use the basic category theory program to build an implementation of this semantics. Given a text of a specification (in the language in question), we can run the text through the program to obtain a denotation for the specification - which essentially is a "theory" (a term we define below). We will briefly describe the program in this chapter.

What of the interplay of programming theory and the running of category theory on a machine advertised in the introduction? In the next chapter we will look at some ways of using the categorical routines to handle programming tasks which (at first sight) have nothing to do with category theory. We will see that for certain restricted types of specifications the ability to compute colimits in categories of algebras gives us a means of developing programs from specifications. More work will be found on the topic of program synthesis when we turn our attention to the properties of theories.

Here we describe a program which uses the category theory routines we have developed to implement a semantics of a specification language. This can be seen as an experiment in developing programs using very general routines which can be fitted together and specialised to the case in hand by the application of the routines. This gives a convenient way of building programs without writing large amounts of complicated code but is inherently inefficient since special cases of general routines can often be performed more quickly than the generality of the routines will allow. We will discuss the problem of efficiency later (page 78).

As mentioned before, it is hoped that the general routines obtained via the programming of category theory will encapsulate program iteration and recursion and thus that programming with these routines will eliminate the difficulties associated with these loop constructs. The following program can be considered to be an extended exercise in 'combinatorial programming' - programming without explicit recursion or iteration. With this in mind we ask : Where do recursions and iterations occur in the program? The answer is:

1. In the colimit routines as designed.

2. In several environment handling operations - which are not formulated in categorical programming terms (if indeed it be possible).

3. In the translation of recursively defined functions into code (for example the semantic function (page 73)).

The aim of the program is to take the text of a specification and give as a result a denotation of this text - that is, we run the semantics on a computer. The specification language in question is called CLEAR and is described in [Burstall,Goguen 1980b] (summarised in Appendix Four), whilst the semantics is fully explained in the paper [Burstall,Goguen 1980a] which can be read in parallel with the following program description.

Specifications in CLEAR are built from algebraic theories using operations to modify or combine theories in various ways. In the following program we choose a representation for algebraic theories and then interpret the specification-building operations as operations on theories in accordance with the semantics. There is an inherent difficulty here in that theories usually contain an infinite set of equations. How are we to represent these? Moreover, the operations we need to perform on these equations assume that we have the power of a theorem-prover at hand. We could avoid these problems by ignoring the equations altogether. Then the denotation of a specification would be the signature of a theory and the operations of CLEAR would reduce to operations on signatures (via the forgetful functor from theories to signatures). This would be of use in checking the sorts and operations available in a specification. What we actually do, as explained below, is leave the operations on equations uninterpreted so that later we may use a theorem prover to check whether an equation is true in a given specification.

This running of a semantics of a specification language is of some interest in itself. A formal specification language and its semantics is a prerequisite for any systematic attempt at program development or program verification. The running of a semantics of such a language on a machine means that we can automatically "check" the intended meaning of a specification. More than this it is one step on the way to an automation of program design and development.

The program itself is a large interpreted HOPE program (about a thousand lines of code) and runs slowly. Approximately half of this code is the, previously described, category program upon which this implementation is based. The semantic operations of the specification language are described as operations within categories - principally colimit operations. It is here that the previous work on simplifying the computation of colimits will come into play and thus we will find that much of the program is already written.

A denotational semantics for a language of any complexity is beset with many details which are incidental to the main object of giving the semantics. This is partly because we are transforming clauses in a language which allow side effects - whose meaning is dependent upon the context of the clause - into functions, in which all arguments are made explicit. We model the dependence upon context by giving the functions "environments" as arguments. These environments must be able to handle all the interactions of a piece of text with its context. Thus, in general, environments are not simple objects at all. In the case of the semantics which we are to implement, environments are diagrams of theories. Moreover, a language of any complexity has many syntactic classes (e.g. commands, expressions) and with each class is associated a semantic function. Each of these needs defining before we are in a position to give the semantics of programs or, in our case, of specifications. As our main interest lies outside that of language semantics we will not be concerned with all these details in the following description of the program which implements the semantics of CLEAR.

The Program

The program description follows closely the semantics to be found in [Burstall and Goguen 1980a].

As explained above a semantics of an algebraic specification language needs some notion of a "theory" as a denotation of specifications. CLEAR is designed to deal with very general theories including equational theories and predicate calculus theories as well as theories based upon categories other than that of sets, for instance the continuous theories of [Wagner,Wright,Goguen,Thatcher 1978]. However, at some points in the text of a specification we require theories that behave like equational theories in that they have initial algebras (and, more generally, algebras free in some sense). Elsewhere more general theories are permitted.

A presentation of an equational theory contains equations to describe, or constrain, the operations in the theory. In predicate calculus theories instead of equations there are general predicate calculus clauses. The theories which we are to use to describe a semantics for CLEAR generalise these cases so that the constraints become sentences in an 'institution'. An institution is a rather complicated object (called a 'language' in [Burstall,Goguen 1980a]), an example of which is predicate calculus with equality. Institutions are defined as follows:

Definition

An institution I consists of an arbitrary category SIG (of 'signatures') equipped with two functors and a relation: -

1. Mod: SIG -> $\underline{\text{Set}}^{op}$ - the set of models over a signature

2. Sen: SIG -> Set - the set of sentences over a signature

3. $\models \subseteq \mathrm{Mod}(\Sigma) \times \mathrm{Mod}(\Sigma)$ for each object Σ of <u>SIG</u>, such that for each $\sigma : \Sigma \to \Sigma'$ in <u>SIG</u>, $s \in \mathrm{Sen}(\Sigma)$ and $m' \in \mathrm{Mod}(\Sigma')$ we have $m' \models \mathrm{Sen}(\sigma)(s)$ iff $\mathrm{Mod}(\sigma)(m') \models s$.

Institutions can be declared as a data type in HOPE,

```
    data Institution(o,m,alpha,beta) ==
            institution( Colimit_Cat(o,m),
                         Functor(o,m,Set(alpha),Set_Mor(alpha)),
                         Functor(o,m,Set(beta),Set_Mor(beta)),
                         (o -> (Set(alpha)#Set(beta)->truval)) )
```

where 'alpha' is the type of models and 'beta' that of sentences. Later we will restrict ourselves to the familiar notion of a signature as sorts and operations - that is, as an object of a certain comma category (page 23). For the moment any category will do - well, any cocomplete category for we will need colimits of signatures.

Suppose I is a institution as above, a presentation (of a theory) over I consists of a signature Σ (an object of <u>SIG</u>) and a subset of the set of sentences $\mathrm{Sen}(\Sigma)$. A theory is then a presentation in which the set of sentences is "closed". In equational theories this closure is the deductive closure. In this more general setting, closure is a semantic closure in terms of models and arises from the above definition of a institution.

Whatever form of closure is used it is not implementable. Even in the case of equational theories, the closure of a set of equations is, in general, an infinite set. We therefore define 'closure' as an unimplemented constructor for a type which represents closed sets of sentences.

```
    data Closure(o,m,beta) ==
                closure(Set(beta))
             ++ Closure(o,m,beta) closeU Closure(o,m,beta)
             ++ closetrans(m,Closure(o,m,beta))
             ++ invtrans(m,Closure(o,m,beta))
             ++ star(o,Closure(o,m,beta))
```

To understand this data declaration, consider the operations which we want to perform on closures, for instance the union of two closures. Since closures are no longer sets of sentences but are a new data type we need to be careful. Notice that if S, T are sets of sentences and $\bar{S}$, $\bar{T}$ are their closures then

$$\bar{S} \cup \bar{T} \neq \overline{S \cup T}$$

Indeed $\bar{S} \cup \bar{T}$ is generally not even closed. We thus must leave such operations unimplemented. The operations in the data declaration are interpreted as follows: Firstly, a closure may be just the closure of a set of sentences, or it may (by 'closeU') be a closure of a union of closures. It may also be either the closure of a translation of a closure by a signature morphism or the inverse translation (inverse image). In the paper referred to above it is shown that the inverse translation of a closure is itself closed. Finally, the 'star' operation is concerned with adding explicit equality operations to a presentation and then closing the result.

There are several identities which hold between these operations. We may use these identities to manipulate expressions for closures, for example,

```
    invtrans(sigmor,C1 closeU C2) =
            invtrans(sigmor,C1) closeU invtrans(sigmor,C2)
```

(Of course, this is part of an equational semantics for the closure expressions.) We have dwelt at some length on these closure operations as they are the key to writing a modular theorem prover based upon CLEAR. For, whilst the operations themselves are not implementable, we should be able to use a theorem prover to check whether a given sentence is in a given closure. This involves interpreting the above closure operations in the setting of a theorem prover. This has been done by Don Sannella using Edinburgh LCF (paper to appear).

We can now define theories. A theory is a pair consisting of a signature, of type 'o', and a closure.

```
    data Theory(o,m,beta) == theory(o,Closure(o,m,beta))
```

Theory morphisms are simply signature morphisms which preserve the sentences. Thus, remembering that we need to include the source and target theories, a theory morphism is:

```
    data Theory_Mor(o,m,beta)
            == theory_mor(Theory(o,m,beta),m,Theory(o,m,beta))
```

In the now-familiar fashion, we can define the category of theories over a institution:

```
    dec cat_of_theories : Institution(o,m,alpha,beta) ->
              Cat(Theory(o,m,beta),Theory_Mor(o,m,beta))
    --- cat_of_theories(i) <=
              cat(s(i),t(i),id(i),comp(i))
```

Here 's' and 't' are the projections of morphisms whilst 'id' and 'comp' arise from the identity and composition in the category of signatures in the institution 'i'.

The semantics of CLEAR which we are to implement is 'categorical' in several senses. Firstly, we can express the operations of CLEAR as operations in a category - principally colimit operations. We thus need to compute colimits of theories. Can we use the techniques developed so far for this task? Notice first that there is a functor from theories to signatures:

```
    dec signature :
           Functor(Theory(o,m,beta),Theory_Mor(o,m,beta),o,m)
    --- signature <=
           functor( (lambda theory(sig,_) => sig),
                    (lambda theory_mor(_,sigmor,_) => sigmor) )
```

This functor lifts colimits uniquely - the signature part of the colimit of theories is the colimit of the signatures of the theories. We cannot go much further as the sentences of the theory require special treatment. We can however use the means of extending simple colimits to those of arbitrary finite diagrams. To this end we define, for instance, the initial theory:

```
    dec init : Institution(o,m,alpha,beta) ->
           InitialObj(Theory(o,m,beta),Theory_Mor(o,m,beta))

    --- init(institution(colimit_cat(csig,sigcolim),_,_,_)) <=
         let (_,_,siginit),siguniv == sigcolim(nil_diagram) in
         let initth == theory(siginit,closure(nil_set)) in
         initth, (lambda pth & theory(psig,_) =>
                   let univmor ==
                         siguniv(nil_diagram,
                                 diagram_mor(nil_diagram,
                                             nil_map,
                                             nil_map,
                                             unit_diagram(csig)
                                                    ofo psig),
                                 psig)   in
                   theory_mor(initth,apex_morphism(univmor),pth))
```

We can define the binary coproduct and the coequaliser of parallel pairs in a similar manner and thus arbitrary finite colimits of theories:

```
    dec c_cat_of_theories : Institution(o,m,alpha,beta) ->
                    C_Cat(Theory(o,m,beta),Theory_Mor(o,m,beta))
    --- c_cat_of_theories(i) <=
          c_cat(cat_of_theories(i),init(i),coprod(i),coeq(i))

    dec colimit_cat_of_theories : Institution(o,m,alpha,beta) ->
              Colimit_Cat(Theory(o,m,beta),Theory_Mor(o,m,beta))
    --- colimit_cat_of_theories(i) <=
          colimit_cat(cat_of_theories(i),
                 finite_colimit(c_cat_of_theories(i)))
```

The semantics of CLEAR is categorical in another sense. Environments become diagrams in a category. An object with its environment is a cone with the environment diagram as base. This cone describes how the apex object depends upon the objects from which it is built (those in the base). We call such objects together with their environments 'based objects' and their morphisms 'based morphisms':

```
    type BasedObj(o,m) == Cone(o,m)

    type BasedObj_Mor(o,m) == Cone_Mor(o,m)
```

What about colimits of these based objects? The category of based objects is not the comma category of cones but a subcategory of it, with the same objects but with a restricted class of morphisms which describe the behavior of environments. There is an explicit construction of these colimits in [Burstall,Goguen 1980a]. Unfortunately it does not seem possible to construct colimits of based objects by lifting techniques (the category of based objects as defined in [Burstall,Goguen 1980a] is not expressible as a comma category built from the category of sets). We thus implement these colimits directly as a procedure 'bo_colimit' of type:

```
    dec bo_colimit : Colimit_Cat(o,m) ->
                       Colimit(BasedObj(o,m),BasedObj_Mor(o,m))
```

(we do not give the definition - it is much like our previous encodings of colimits). The category of based objects and their colimits is then:

```
    dec colimit_cat_of_based_objects : Colimit_Cat(o,m) ->
              Colimit_Cat(BasedObj(o,m),BasedObj_Mor(o,m))

    --- colimit_cat_of_based_objects(CC & colimit_cat(C,clim)) <=
          colimit_cat(cat_of_cones(C),bo_colimit(CC))
```

The denotation of a CLEAR text is going to be a based object in the category of theories - a thing which we call a based theory. Some of the semantic operations are described in terms of colimits in the category of based theories. For example the COMBINE operation is the coproduct of two theories:

```
    dec combine_op : Institution(o,m,alpha,beta) ->
          ((BasedObj(Theory(o,m,beta),Theory_Mor(o,m,beta))
           # BasedObj(Theory(o,m,beta),Theory_Mor(o,m,beta)))
                -> BasedObj(Theory(o,m,beta),Theory_Mor(o,m,beta)))

    --- combine_op(i) <= lambda t1,t2 =>
        let colimit_cat(_,bthcolim) ==
                colimit_cat_of_based_objects(
                            colimit_cat_of_theories(i)) in
        let (_,_,coproduct),_ == bthcolim(cpdiagram(t1,t2)) in
                    coproduct
```

('cpdiagram' takes two objects and forms a two-object, no-morphism, diagram)

The operations of ENRICH and procedure application may be treated similarly. The application of a procedure is, in fact, described by a pushout in the category of based theories. The DERIVE operation, which is a data abstraction facility, can be thought of as a factorisation in a category. In the program we give an explicit operation on theories for this DERIVE operation.

The operation DATA requires us to add extra constraints to a theory - not simply sentences of an institution but also data constraints. A data constraint is a pair consisting of a theory morphism and a signature morphism. See the reference above to understand why this is so. Data constraints behave like sentences in that we may form their closure to get a theory from a presentation. This motivates the following type declaration:

```
    data  Constraint(o,m,beta) ==                            - (*)
            equation(beta) ++
            data_constraint( Theory_Mor(o,m,Constraint(o,m,beta)),
                             m )
```

So far we have dealt with fairly general theories whose signatures lie in an arbitrary cocomplete category and whose constraints are of the form (*) for an arbitrary institution. To give explicit denotations for CLEAR texts, we need to specialise these general signatures and constraints. We choose our signatures to be the familiar 'sorts and operations', that is, objects of a certain comma category (page 23). The sentences are to be either equations or data constraints built from equations. Equations will be of the form:

```
    data Eqn == all Set(V_Name#S_Name) . Term = Term
```

(where 'V_Name' is the class of variable names, 'S_Name' that of sort names and a 'Term' is either a variable name or an operation name followed by a list of 'Term')

We call theories of this restricted type 'Ordinary_Theory' and their morphisms 'Ordinary_Theory_Mor'. Based objects in the category of these theories will be of type 'Based_Theory'. The institution of these signatures and constraints will be called 'Ordinary_Institution(alpha)' where 'alpha' is the type of models in the institution.

Omitting some details concerning syntactic classes and the so-called dictionaries, we are at last in a position to define the semantics of CLEAR. Based theories are described by the CLEAR expressions:

```
    data E == just(T_Name)
            ++ theory Enr endth
            ++ E + E
            ++ enrich E by Enr enden
            ++ derive Enr using set E from E by Sic endde
            ++ let_th T_Name = E in_th E
```

Here T_Name is the syntactic class of theory names whilst Sic is that of signature changes (DERIVE and procedure application use signature changes). Notice the use here of distributed-fix operations (e.g. enrich _ by _ enden).

Environments are diagrams,

```
    type Env ==
       Diagram(Ordinary_Theory,Ordinary_Theory_Mor)
```

whilst procedure environments associate with a procedure-name a based theory morphism (the denotation of the procedure) and a list of based theories (the meta-sorts):

```
    type ProcEnv ==
            T_Name -> (Based_Theory_Mor # list Based_Theory).
```

The semantic function is defined in terms of the following function which gives the denotation of the above expressions for (based) theories:

```
    dec E : E -> (Ordinary_Institution(alpha) ->
                    (Env -> (ProcEnv -> Based_Theory)))
```

For example for a COMBINE expression of type E + E,

```
    --- E(e1 + e2) <=
            (lambda i => (lambda rho => (lambda pi =>
          combine_op(i)(E(e1)(i)(rho)(pi),E(e2)(i)(rho)(pi)) )))
```

The other operations of CLEAR can be handled in a similar fashion.

The semantics of CLEAR is then encoded as follows: The syntax of specifications is declared as a data type,

```
    data Spec ==    const T_Name = E in_spec Spec
             ++ proc T_Name << List(T_Name#E) >> = E in_spec Spec
             ++ just(E)
```

where the first term is the definition of a constant theory in a specification, the second the definition of a procedure, whilst the third is just an expression for a (based) theory.

The semantic function for these CLEAR specifications is:

```
    dec Spec : Spec -> (Ordinary_Institution(alpha,Eqnlike) ->
                    (Env -> (ProcEnv -> Based_Theory)))
```

For the first case (definition of constant theories), the function is defined by:

```
    --- Spec(const tname = e in_spec s) <=
            (lambda i =>(lambda rho => (lambda pi =>
          Spec(s)(i)( bind(tname,E(e)(i)(rho)(pi),rho) )(pi) )))
```

This is simply the binding of a theory name to a theory in an environment (a diagram). The function 'bind' flattens a based object into an environment. The case of procedure definition in a specification is:

```
    --- Spec(proc pn << list_tnxe >> = e in_spec s) <=
        (lambda i =>(lambda rho => (lambda pi =>
             let list_T ==
                (lambda _,ee => E(ee)(i)(rho)(pi)) * list_tnxe in
             let list_tn ==
                    (lambda tname,_ => tname) * list_tnxe in
             let rho1 == bind(list_tn,list_T,rho) in
             let T1 == E(e)(i)(rho1)(pi) in
             let F ==
              node_morphism(colimit_cat_of_theories(i))
                           (list_tn,T1,rho1)   in
             let pi1 ==
                (lambda pnn =>
                          (F,list_T) if pnn = pn else pi(pnn)) in
         Spec(s)(i)(rho)(pi1) )))
```

To evaluate a procedure definition we need to evaluate the metasorts (which denote based theories, hence we use the function 'E'), bind the formal parameter names to these theories, and then evaluate the body of the procedure. Finally we modify the procedure environment by binding all this to the procedure name. The final case is:

```
    --- Spec(just(e)) <=
            (lambda i =>(lambda rho => (lambda pi =>
      E(e)(i)(rho)(pi) )))
```

To evaluate a specification of this form we simply use 'E' to find the denotation.

This completes our exposition of the program. A parser for CLEAR has been written by D. Sannella and thus we may apply the semantic function "Spec" to the parsed text to get a meaning for the text in the form of a (based) theory. As mentioned before the performance of the program is poor - needing about half a hour of CPU time on a DEC10 to evaluate a simple specification. However it does illustrate this program development technique on a large and complex program.

For the sake of comparison and to get a usable program, D. Sannella [1981] has written a direct implementation of a variant of this semantics (also in the language HOPE). The semantic operations, instead of being interpreted as operations in a category, specialised to the category of theories in question, are implemented directly on theories. This gives an improvement of a factor of a thousand or so in the running time! Whilst some of this increase in efficiency can be attributed to the inefficiency of using general routines for specific problems it seems that most of it arises from the sheer size of the program in question causing, for instance, a large amount of page thrashing.

Notes

The language CLEAR and its semantics are described in [Burstall and Goguen 1977], [Burstall and Goguen 1980a] and [Burstall and Goguen 1980b]. The encoding of the semantics described here was rather simple (as it was meant to be) though it has proved difficult to "debug". The program is now running albeit rather slowly. It was a joint project between Don Sannella and myself (shared roughly equally).

CHAPTER FIVE

THE DEVELOPMENT OF PROGRAMS FROM SPECIFICATIONS - SOME EXPERIMENTS

We here start some experiments in systematic program development by various techniques made available through the programming of category theory. This discussion continues for the rest of this paper where we will be investigating properties of certain types of "theories" which we use in program specification. Many of the properties of these theories can be expressed categorically by colimit constructions and can be made amenable to the programming efforts already available. This gives further insight into the process of program synthesis. Here we investigate a few problems using a method, known as "colimit recursion", of translating functors into recursive programs

COLIMIT RECURSION - RECURSION ON COMPLEX DATA TYPES

Some stress has been laid on the fact that colimits provide us with a means of building up and breaking down objects in a category. We now show how this can be put to use in programming with a technique called "colimit recursion" introduced in [Burstall 1980].

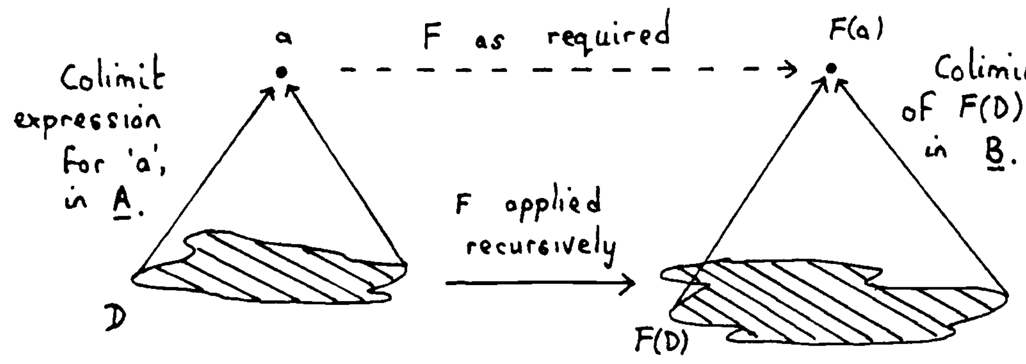
The divide and conquer paradigm for the recursive computation of a procedure on, say, sets means that we split a set in parts, calculate the procedure recursively on the parts then assemble the results in some fashion. The splitting up of sets is easy - they can simply be partitioned. For more complex data structures, graphs for example, partitioning is not possible. When splitting a graph we need to have a shared part telling us how to reassemble the pieces into the original graph. We are saying that to decompose a graph we need, in general, a pushout rather than a coproduct.

In general a splitting of an object in a category, $\underline{C}$, is a diagram in the category. Now suppose that we wish to compute some function of these objects. Assume that the function is the object part of a functor, F, and that the results of the computation lie in some category, $\underline{B}$,

$F : \underline{A} \rightarrow \underline{B}$

We can calculate F on an object, a, by expressing 'a' as a colimit object of a diagram, D. There are, of course, many ways of doing this. We then apply F recursively to the objects and the morphisms in D to get a diagram, D ', in the category B̲. We then take the colimit of this diagram in B̲. Now provided that F preserves colimits (is cocontinuous), the colimit object of this colimit in B̲ is (isomorphic to) F(a), as required. The proviso that F is cocontinuous is crucial but as we shall see, functions that we wish to compute can often be expressed as (the object parts of) free functors in which case cocontinuity is assured by a general theorem.

A picture of this process may be helpful.



Notice that the recursive call of F requires us to compute the morphism part of F recursively as well as the object part. The objects are broken up into diagrams and thus the morphisms are to be expressed as diagram morphisms. That is, our recursive splitting is a functor

decompose : A̲ -> <u>Diagram</u>(A̲)

such that 'colimit_apex_functor.decompose' is naturally equivalent to the identity functor, where 'colimit_apex_functor' is the functor taking a diagram to the colimiting object on the diagram. It is defined on page 35. A similar general decomposition of objects which give rise to recursive programs has been investigated in [Burstall,Landin 1969] and [Klaeren 1980]. In these papers the decomposition takes place in the carrier of an algebra. The relation between this and the decomposition functor above is formalised at the end of this chapter (page 96).

What about the base cases on which we compute F directly? These are determined by our choice of a recursive splitting, that is, by the decomposition functor above.

Examples of the use of this general means of recursion will appear in the next section but it should be said at the outset that it is often not at all obvious how a familiar function that we are required to compute can be expressed as a functor. In particular, the choice of the category of results is not at all obvious. This may, of course, be merely our own failure to look at the problem in the right light or our being hindered by our programmer's intuition. An example of this difficulty is given in [Burstall 1980] where the problem is to calculate the shortest distance between nodes in a distance graph. The category of results suggested for this problem is the category of metric spaces - a choice familiar to mathematicians but possibly not to programmers (metric spaces are not a common data type in programming) although, of course, the structure of the problem is captured in the concept of metric spaces.

The role of the computation of colimits and the work done in the previous sections in finding short cuts to these computations by lifting colimits through category constructors ought now to be apparent. The whole work of the recursive program (apart from the computation on the irreducible objects of the recursion) is in the computation of colimits in the category $\underline{B}$. The expression of objects as colimit objects of a diagram in $\underline{A}$ is a consequence of our choice of representation of the data types, although sometimes we are able to write some code for a functor 'decompose' as above. Notice how we have split the control structure and the computation in such a way that the computation is always a calculation of colimits.

Finally, a word about the efficiency of these programs. In general the programs obtained are not highly efficient. As mentioned before, the building of programs from general routines, like the colimit routines, may well lead to inefficiency when we have only special cases to calculate for which quick methods may be available. There is always the hope that program transformation may provide a means of increasing the efficiency of the programs and the further hope that the structure of these programs will lend itself readily to program transformation techniques such as those in [Burstall,Darlington 1977]. A future for program transformation may be in its application to programs of a fixed structure such as these programs obtained via categorical insights. However, notice that in these recursive programs we do not specify how the decomposition of objects is to be achieved. So far, any 'decompose' functor, as above, will suffice. This means that the programs are well adapted to parallel processing, as we may decompose the objects so as to distribute pieces among processors in a parallel array and compute the procedure recursively on these pieces. Once general purpose parallel machines are available, efficiency may well not be a prime concern whereas the ability to incorporate parallelism into a routine may well come to the fore. As mentioned in [Schwartz 1980] the mere possibility of large parallel processor arrays does not create ways of exploiting the technology. We need to understand how to incorporate parallelism into computations. Another reason for not being too concerned about the inefficiency is that once a fixed repertoire of general routines is agreed upon in a certain field of programming, these routines may be 'hardwired' and thus a great improvement in speed might be achieved.

In formal complexity terms, the programs are (with a suitable choice of machine model) polynomial time if the basic category (on which others are built with category constructors) is that of sets. Exactly what degree polynomial and what constant factor depends upon how we choose to decompose the objects in question. Some decompositions require less duplication of work and converge faster than others.

EXERCISES IN PROGRAM DEVELOPMENT

We now show how specifications can give rise to programs with two examples both chosen from graph theory. Recall that graphs are directed multigraphs possibly with loops and cycles.

Connected components of a graph

Two subgraphs of a graph are said to be disconnected if their nodes are disjoint and there are no edges in the graph from nodes in either subgraph to nodes in the other. If a subgraph cannot be expressed as two non-empty disconnected subgraphs it is said to be connected. Connected components of a graph are maximal connected subgraphs. These components can be characterised as the equivalence classes under the reflexive, symmetric, transitive closure of the relation on nodes, $\sim$, defined by:

$a \sim b \Leftrightarrow$ There is an edge, e, in the graph whose source is a and target b.

How are we to convert this 'mathematical' definition into an algebraic specification? A rather simple (and high level) specification of a graph and its connected components can be given by labelling the nodes of a graph in such a way that, if the labels of two nodes are the same then the nodes are in the same component and if the labels differ, then they are in different components.

The specification, then, of the connected components of a graph is given as a theory enrichment, taking the theory of graphs and giving a theory of graphs with labelled nodes. In the specification language CLEAR (whose main features are summarised in Appendix Four), graphs can be given as:

```
constant Graph =
     sorts node, edge
     opns  source : edge -> node
           target : edge -> node
                                end
```

Then the specification becomes

```
constant Component Graph =
   enrich  Graph + Triv by
      data  opns label : node -> element
            eqns label(source(e)) = label(target(e))
                                                 end
```

Here the theory Triv is the trivial theory of one sort:

```
constant Triv =
         sorts element
                    end
```

This is a high level specification in that it gives no hint as to how we are to compute such a labelling.

Let us look carefully at this specification. The first thing to notice is that the equation merely says that nodes in the same component have the same label. The converse requirement, that nodes in different components have different labels, not only is difficult to state using equations but need not be included! It arises from the freeness condition of a functor. This free interpretation of the specification is indicated by the appearance of the keyword "data". Data constraints are explained in [Burstall, Goguen 1980a]. Here we see a universal condition at work. The specification is simplified whilst the universal condition is implicit in any use of the specification, in particular it is a correctness condition for any program developed from the specification. It should be said that this is a rather unorthodox use of data constraints. Data constraints were originally introduced as a means of defining new data in a specification whereas we are using them in the more general sense of universal conditions on a specification. Nevertheless our usage accords with the semantics of the language CLEAR and can be seen as a by-product of it.

To understand what is going on, let us first consider the specification without the data constraint:

```
constant Labelled Graph =
     enrich Graph + Triv by
        opns label : node -> element
        eqns label(source(e)) = label(target(e))
                                          end
```

Algebras of this specification are graphs with nodes labelled such that nodes in the same component have the same label, the converse requirement may not hold.

The enrichment above gives rise to a theory morphism

α : Graph -> Labelled Graph

(because the theory Graph is included in Labelled Graph).

Algebras of Graph are just graphs and algebras of Labelled Graph are graphs with their nodes labelled. Now dropping down to the level of algebras, the theory morphism above induces a "forgetful" functor

U : *Alg*(*Labelled Graph*) -> *Alg*(*Graph*)

which takes any graph with its nodes labelled and simply forgets the labelling, thus giving a graph. Our task is to assign a component labelling to a graph. Indeed, there is a *left-adjoint* of the functor above, the free functor:

F : *Alg*(*Graph*) -> *Alg*(*Labelled Graph*).

Now, by definition of a data constraint, any image of a graph under F is an algebra of *Component Graph*. Thus it is this F that we wish to compute.

The existence of such free functors arising from theory morphisms is not automatic. In this case the functor exists and, as we shall prove later (page 147), it exists for all equational specifications. It is the presence of the word "data" which ensures that it is the free functor that is intended.

How then do we compute this functor? We use colimit recursion, remembering that free functors are always cocontinuous.

To use colimit recursion, we need a means of computing colimits in *Alg*(*Labelled Graph*). Fortunately, the techniques that we have developed provide us with such a means. We can lift colimits from the category of sets because *Alg*(*Labelled Graph*) can be expressed as a comma category as follows.

Let D : *Set* -> *Alg*(*Graph*) be the functor taking a set onto the discrete graph on the elements of the set. A discrete graph has the same set for the nodes and for the edges and each edge has as source and target the node with the same name.

Then *Alg*(*Labelled Graph*) ≅ (*Alg*(*Graph*),D) because a graph, g, with its nodes labelled (not necessarily distinct labels for distinct components) is essentially a graph morphism, f : g -> sD where sD is a discrete graph. The fact that f is a graph morphism ensures that two nodes in the same component of g must map onto the same node in the discrete graph, sD.

Now, colimits of sets lift to those in Alg(Graph) and so, by the lifting construction for comma categories, we can compute colimits in Alg(Labelled Graph).

Translation of this observation into code goes as follows. The functor, D, has object part:

```
    dec discrete_graph : Set(Tag alpha) -> Graph(Tag alpha)
    --- discrete_graph(S) <=
            graph(S,(lambda s => pair(s,s)),S)
```

and the functor is then given by:

```
    dec D : Functor( Set(Tag alpha),Set_Mor(Tag alpha),
                     Graph(Tag alpha),Graph_Mor(Tag alpha))
    --- D <=
            let C == cat_of_sets in
        functor( discrete_graph,
                 lambda m =>
                    comma_mor( discrete_graph(source(C)(m)),
                               (m,m),
                               discrete_graph(target(C)(m))) )
```

As a comma object a labelled graph is:

```
    type LG_object(alpha) ==
            Graph(alpha) # Graph_Mor(alpha) # Set(alpha)
```

We can construct such a comma object from a graph, a set, and a suitable map from nodes of the graph to elements of the set:

```
    dec labelled_graph :
            Graph(Tag(alpha)) #
            (Tag(alpha)->Tag(alpha)) #
            Set(Tag(alpha)) ->
                  LG_object(Tag(alpha))

    --- labelled_graph(G,f,S) <=
            let n_mor == mor(nodes(G), f, S) in
            let e_mor == mor( edges(G),
                              (lambda e => f(dom(G)(e)) ),
                              S ) in
            let g_mor == comma_mor( G,
                                    (e_mor,n_mor),
                                    discrete_graph(S)) in
        ( G, g_mor, S )
```

Morphisms of such comma objects are comma morphisms.

```
    type LG_morphism(alpha) ==
            Right_Comma_Mor( Graph(alpha),
                             Graph_Mor(alpha),
                             Set(alpha),
                             Set_Mor(alpha) )
```

The category, LG, of labelled graphs is thus a right comma category:

```
    dec LG : Cat( LG_object(Tag alpha),
                  LG_morphism(Tag alpha))
    --- LG <= right_comma_cat(cat_of_graphs,D,cat_of_sets)
```

This information now gives us directly the colimits in the category of connected component graphs as the tiny piece of code demonstrates:

```
    dec cLG : Colimit_Cat( LG_object(Tag alpha),
                           LG_morphism(Tag alpha))
    --- cLG  <=
          colimit_comma_cat( colimit_cat_of_graphs,
                             D,
                             colimit_cat_of_sets )
```

The other part of the colimit recursion is a splitting of graphs as colimits. How are we to do this? We are quite at liberty to choose how we want to do this splitting and there may be cases when we want to leave this question open (e.g. for a maximal use of resources in parallel processing). However, we choose to restrict ourselves to expressing graphs as pushouts in the category of graphs. This seems a natural way to decompose graphs.

The irreducible graphs under this recursion are: the graph with no nodes, "nil", the graph with one node and no edges, "node(n)", and the graph with just one edge, "edge(s,e,t)", where the parameters s,t are the source and target nodes of the edge, e.

This decomposition allows us to represent any graph as either an irreducible graph or as a pushout of graphs. Remembering that graph morphisms must also be recursively split, we have the interdefined types:

```
    data Graph1(alpha) ==
            nil ++
            node(alpha) ++
            edge(alpha,alpha,alpha) ++
            po_graph(Po_Diagram(Graph1(alpha),Graph1_Mor(alpha)))

    with Graph1_Mor(alpha) ==
            nil_mor(Graph1(alpha)) ++
            node_mor(alpha,alpha,Graph1(alpha)) ++
            edge_mor( (alpha#alpha#alpha),
                      (alpha#alpha#alpha),
                      Graph1(alpha)) ++
        po_mor(Po_Diagram_Mor(Graph1(alpha),Graph1_Mor(alpha)))
```

The types of pushout diagram and pushout diagram morphism in the above are:

```
    type Po_Diagram(o,m) == m # m

    type Po_Diagram_Mor(o,m) ==
              Po_Diagram(o,m) # (m # m # m) # Po_Diagram(o,m)
```

That is, a pushout diagram is treated as a pair of morphisms. Morphisms of pushout diagrams contain a source and target diagram as well as morphisms from the three objects in the source diagram to the objects in the target diagram.

The irreducible graph morphisms above are: "nil_mor(g)", the unique morphism from the empty graph to the graph, g, "node_mor(a,b,g)", the morphism from the graph with one node 'a' to the graph, g, where the image of 'a' is 'b', and "edge_mor((a,e,b),(c,f,d),g)", the morphism from "edge(a,e,b)" into g with the image of 'e' as 'f' and of 'a' and 'b' as 'c' and 'd'.

Notice that so far we have done no calculation except that of the functor, D, and the calculation implicit in the basic lifting of colimits. Now without further calculation we can compute the functor, F, whose object part takes a graph onto a connected component graph by recursive application to graphs and their morphisms:

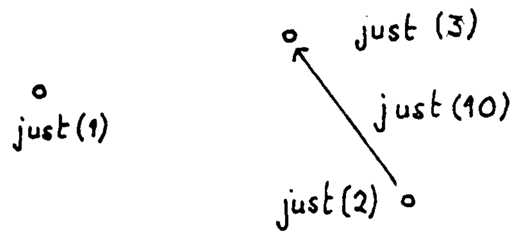
```
dec F : Graph1(Tag(alpha))     -> LG_object(Tag(alpha))
dec F : Graph1_Mor(Tag(alpha)) -> LG_morphism(Tag(alpha))

                ! object part of F - recursive routine :
--- F(nil)
        <= labelled_graph(nil_graph,nil_fn,nil_set)
--- F(node(n))
        <= labelled_graph(node_graph(n),ident,{n})
--- F(edge(n1,e,n2))
        <= labelled_graph( edge_graph(n1,e,n2),
                           (lambda _ => n1),
                           {n1} )
--- F(po_graph(m1,m2))
        <= pushout(cLG) ofo (F(m1),F(m2))

                ! morphism part of F - also recursive :
--- F(nil_mor(g1))        <=    let lg == F(g1) in
                                let (g,_,s) == lg in
        comma_mor(F(nil),(nil_morphism(g),nil_mor(s)),lg)
--- F(node_mor(a,a1,g1)) <=     let lg == F(g1) in
                                let (g,gm,s) == lg in
                                let mor(_,f,_) ==
                                         right ofm gm in
        comma_mor( F(node(a)),
                   ( node_morphism(a,a1,g),
                     singleton_mor(a,f(a1),s)),
                   lg)
--- F(edge_mor((a,e,b),(a1,e1,b1),g1)) <=
                                let lg == F(g1) in
                                let (g,gm,s) == lg in
                                let mor(_,f,_) ==
                                         right ofm gm in
        comma_mor( F(edge(a,e,b)),
                   ( edge_morphism((a,e,b),(a1,e1,b1),g),
                     singleton_mor(a,f(a1),s)),
                   lg)
--- F(po_mor( (m1,m2),(l,c,r),(m3,m4) )) <=
        pushout(cLG) ofm
          ( (F(m1),F(m2)), (F(l),F(c),F(r)), (F(m3),F(m4)) )
```

Here "pushout(cLG)" is the functor taking a pushout diagram to the colimit object and a morphism of pushout diagrams to the universal morphism between the colimit objects. It uses the colimit computation implicit in the colimit category, "cLG".

This then is the program we have developed from the specification of the problem. To show that it is a genuine program we give an example of it running. Consider the graph pictured below:



It can be expressed as a pushout of irreducible graphs as follows:

```
    dec g : Graph1(Tag num)
    --- g <= po_graph( nil_mor(node(just 1)),
                       nil_mor(edge(just(2),just(10),just(3))))
```

We then compute F(*g*). The value of F(*g*) is a connected component graph and contains a graph whose nodes are

```
    { pink(just(1)),
      blue(blue(pink(just(3)))),
      blue(blue(pink(just(2)))) }
```

and whose edges are

```
    { blue(blue(pink(just(10)))) }.
```

Notice the colouring that takes place in the evaluation of this graph by colimits. The connected component graph, F(*g*), is then a set of labels and a graph morphism from the above graph to this set of labels (considered as a discrete graph). The set of labels turns out to be:

```
    { pink(just(1)), blue(blue(pink(just(2)))) }
```

showing that *g* has just two components as it certainly does.

In keeping with our aim of encapsulating recursion in general routines, we would need to introduce a routine, say "colimit_recursion", which given sufficient information would compute the functor F by colimit recursion. It is not immediately apparent how we are to do this. The problem is to find the irreducibles of the decomposition of objects and morphisms (perhaps these ought to be given as arguments to the routine). It seems that we have not formulated the explanation of colimit recursion sufficiently abstractly.

This program development was made easier by noticing that the target data type could be expressed as a comma category. Thus the computation of colimits was available by the lifting routines. In general, data types cannot be expressed as comma categories. We now look at an altogether different and more difficult graphical problem, that of the transitive closure of a graph. For this problem no such lifting device is available. We need the explicit construction of colimits in a rather complicated category - that of small categories (categories whose collections of objects and morphisms both are sets and not "bigger" entities). We do not completely solve the problem here. The discussion spills over into the next few chapters where new techniques are made available to us. There a complete program is given for the transitive closure of a graph.

Transitive closure of a graph

We now look at the problem of developing a program for the transitive closure of a graph. That is, given a graph, we want to construct a new graph with the same nodes but with edges corresponding to paths in the original graph (for each path we have an edge in the transitive closure). This is an extension of the transitive closure of a relation so is quite a useful procedure. The exact relationship between these two problems and an interesting discussion on further analogues may be found in [Aho,Hopcroft,Ullman 1974]. Notice that programs to compute this transitive closure will not halt if the graph has cycles in it.

We first generalise the problem slightly so as to produce a small-category from a graph (strictly this is not necessary but it is helpful to deal with familiar data types). That is, we produce not only the transitively closed graph but also the identities for each node and the composition operation under which the transitive closure is closed.

A small category differs from a category in that the objects and morphisms both form sets (rather than any "larger" collection):

```
    data Small_Cat(o,m) ==
            small_cat( Set o,         ! set of objects
                       Set m,         ! set of morphisms
                       (m->o),        ! source
                       (m->o),        ! target
                       (o->m),        ! identity
                       (m#m->m) )     ! composition
```

That is, it is a category with only a set of objects and a set of morphisms. Morphisms between small categories are small functors:

```
    data Small_Functor(o,m) ==
            small_functor( Small_Cat(o,m),
                           ( Set_Mor(o) # Set_Mor(m) ),
                           Small_Cat(o,m) )
```

We can thus form a category, SC, of small categories.

```
    dec SC : Cat(Small_Cat(o,m),Small_Functor(o,m))

    --- SC <= let comp == compose(cat_of_sets) in
         cat( (lambda small_functor(s,_,_) => s),
              (lambda small_functor(_,_,t) => t),
              (lambda s_cat => let id == identity(cat_of_sets) in
                 small_functor( s_cat,
                               ( id(objects(s_cat)),
                                 id(morphisms(s_cat))),
                               s_cat)),
              (lambda small_functor(s,(f,g),_),
                      small_functor(_,(f1,g1),t) =>
                 small_functor(s,(comp(f,f1),comp(g,g1)),t)) )
```

Notice that there are projection functors giving the set of objects and the set of morphisms respectively.

```
    dec objects :   Functor(Small_Cat(o,m),Small_Functor(o,m),
                            Set(o),Set_Mor(o))
    dec morphisms : Functor(Small_Cat(o,m),Small_Functor(o,m),
                            Set(m),Set_Mor(m))

    --- objects   <= functor((lambda small_cat(oset,_,_,_,_,_)
                                   => oset),
                             (lambda small_functor(_,(mo,_),_)
                                   => mo) )
    --- morphisms <= functor((lambda small_cat(_,mset,_,_,_,_)
                                   => mset),
                             (lambda small_functor(_,(_,mm),_)
                                   => mm) )
```

The functor 'objects' lifts colimits - objects in the colimit are the colimit of objects in the diagram. However, unlike the case of graphs, 'morphisms' does not lift colimits - extra work is required to compute the morphisms of the colimit small category.

A formal specification of the problem goes as follows. Graphs are to be as before:

```
    constant Graph =
         sorts node, edge
         opns  source : edge -> node
               target : edge -> node
                                 end
```

The required program is specified by the theory enrichment:

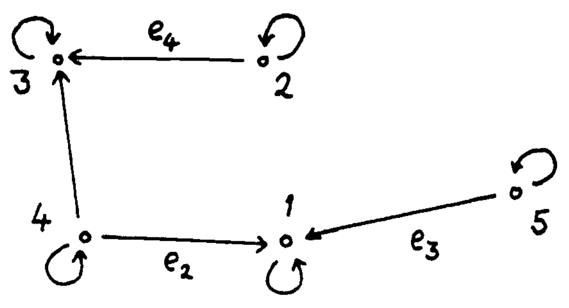
```
    constant Transitive Closure =
       enrich Graph by
          data  opns identity : node -> edge
                     compose  : target # source -> edge

                eqns source(compose(e,f)) = source(e)
                     target(compose(e,f)) = target(f)
                     source(identity(n)) = n
                     target(identity(n)) = n
                     compose(e,compose(f,g)) =
                            compose(compose(e,f),g)
                     compose(identity(n),e) = e
                     compose(f,identity(n)) = f
                                                     end
```

Here 'target # source' is defined for any graph as the set of pairs of edges (e,f) such that target(e) = source(f), that is, it is the set of all composable pairs. This is a new facility in algebraic specification which we have not yet investigated thoroughly. The notation is that of a pullback in category theory. Further discussion will be found on page 191.

Before plunging into the development of a program for the transitive closure of graphs, let us have a look at some algorithms which exist for this problem.

Our first attempt at a recursive program may be to extract an edge from a graph, recursively close the remainder of the graph, then replace the edge adding all the new edges which will arise by composition with the replaced edge. Thus if the closed graph (as the underlying graph of a small category) is given by



adding an edge e1 from '1' to '2' will require us to add extra edges for the paths e2,e1,e4 and e3,e1,e4, as well as those which result from the composition with the identities. We see that any new edge required to be added will be of the form

(An edge with target as source of e1),
e1,
(An edge with source as target of e1).

Such a composition will include the new edge, e1, itself, by choosing the identities at its source and target.

Because we are deriving these morphisms in a small category from paths of edges in a graph we can introduce an explicit representation of these morphisms (later we will see that this representation comes from a free theory construction):

```
    data Morphism(alpha) ==
            id(alpha) ++ morphism(alpha,string(alpha),alpha)
```

Thus a morphism is either the identity on an object (a node in the graph) or a string of edges (a path) in the graph (together with a source and target node). We can define source, target and composition of these morphisms:

```
dec mor_source : Morphism(alpha) -> alpha
--- mor_source(id(a)) <= a
--- mor_source(morphism(a,_,_)) <= a

dec mor_target : Morphism(alpha) -> alpha
--- mor_target(id(a)) <= a
--- mor_target(morphism(_,_,a)) <= a

dec comp : Morphism(alpha) # Morphism(alpha)
             -> Morphism(alpha)
--- comp(id(n),m) <= m
--- comp(m,id(n)) <= m
--- comp(morphism(a,p1,b),morphism(c,p2,d))
            <= morphism(a,p1.p2,d)
```

If the set of morphisms into the source of the new edge is called 'in_morphisms' whilst the set of morphisms out of the target of the new edge is called 'out_morphisms', then the code for the addition of a new edge looks like:

```
dec add_edge :
        Morphism(Tag(alpha)) #
        Small_Cat(Tag(alpha),Morphism(Tag(alpha)))
            -> Small_Cat(Tag(alpha),Morphism(Tag(alpha)))

--- add_edge( m, s_cat & small_cat(_,_,s,t,_,_) ) <=
      let in_morphisms ==                  ! morphisms into 'a'
        filter morphisms(s_cat) by
            (lambda x => mor_target(x) = mor_source(m)) in
      let out_morphisms ==                 ! morphisms from 'b'
        filter morphisms(s_cat) by
            (lambda x => mor_source(x) = mor_target(m)) in
      let new_morphisms ==                 ! the extra morphisms
        (lambda m1,m2 => comp(m1,comp(m,m2))) *
                (in_morphisms X out_morphisms) in
      small_cat( objects(s_cat),
                 new_morphisms U morphisms(s_cat),
                 mor_source,
                 mor_target,
                 id,
                 comp )
```

The recursion then is to split off an edge of the graph, close the remainder, then add the edge using the function above.

```
    dec closure : Graph(Tag(alpha)) ->
            Small_Cat(Tag(alpha), Morphism(Tag(alpha)))

    --- closure(G)  <=
          discrete_small_cat(nodes(G))      ! initial case
                    if edges(G) = nil_set else
                                            ! recursive case
            let e,E1 == singleton_split(edges(G)) in
            let a == dom(G)(e) in
            let b == range(G)(e) in
            let reduced_graph ==
                    graph(E1,graph_map(G),nodes(G)) in
         add_edge( morphism(a,unit(e),b), closure(reduced_graph))
```

('dom(G)' and 'range(G)' are the functions taking an edge to its source and target, respectively, in graph G.)

The initial case is the discrete small category on a set of nodes whose only morphisms are the identities on the nodes:

```
    dec discrete_small_cat : Set(Tag(alpha)) ->
            Small_Cat(Tag(alpha),Morphism(Tag(alpha)))
    --- discrete_small_cat(N) <=
            small_cat(N, id*N, mor_source, mor_target, id, comp)
```

It is interesting to note that we can analyse the program in categorical terms. The addition of a new edge as encoded in 'add_edge' can be considered to be a special case of the computation of pushouts in the category of small categories. This is because the splitting of an edge from a graph can be seen as a pushout in the category of graphs:

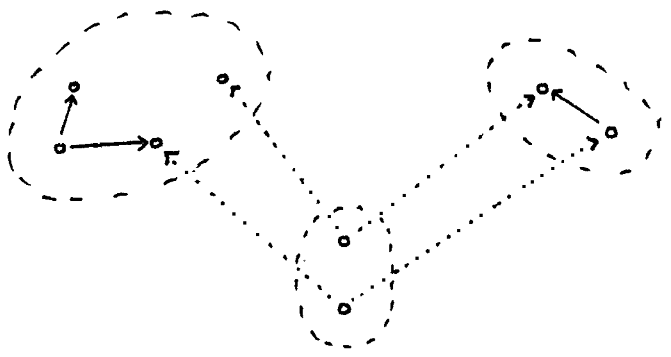


Fig: Extracting an edge from a graph is a pushout.

Thus the recursion in 'closure' is a special case of colimit recursion. This gives us some assurance that recursive programs may be amenable to this type of "categorical analysis". Analysis of programs using category theoretic tools has already been noted, for instance by Burstall and Landin (1969) and Burstall (1980).

What about iterative programs? A 'fast' algorithm for this problem is known, due to Warshall [1962]. It does not correspond to any obvious recursive splitting of graphs. This is all that we can say for the moment for we have no techniques for developing such algorithms, though later we shall see how to handle iteration in category theory (page 135).

We now return to the program development by colimit recursion. We express graphs as colimits exactly as in the example of the connected components of a graph. That is, graphs are given by:

```
    data Graph1(alpha) ==
            nil ++
            node(alpha) ++
            edge(alpha,alpha,alpha) ++
            po_graph(Po_Diagram(Graph1(alpha),Graph1_Mor(alpha)))

    with Graph1_Mor(alpha) ==
            nil_mor(Graph1(alpha)) ++
            node_mor(alpha,alpha,Graph1(alpha)) ++
            edge_mor( (alpha#alpha#alpha),
                      (alpha#alpha#alpha),
                      Graph1(alpha)) ++
         po_mor(Po_Diagram_Mor(Graph1(alpha),Graph1_Mor(alpha)))
```

The recursive program for this problem will then look like:

```
    dec F : Graph1(alpha)     -> Small_Cat(alpha,Morphism(alpha))
    dec F : Graph1_Mor(alpha) ->
                      Small_Functor(alpha,Morphism(alpha))

    --- F(nil)              <= nil_cat
    --- F(node(n))          <= node_cat(n,id(n))
    --- F(edge(m,e,n))      <= edge_cat( (m,id(m)),
                                        morphism(unit(e)),
                                        (n,id(n)))
    --- F(po_graph(m1,m2))  <= SC_pushout ofo (F(m1),F(m2))

    --- F(nil_mor(g))       <= nil_s_functor(F(g))
    --- F(node_mor(a,b,g))  <= node_s_functor( (a,id(a)),
                                              (b,id(b)),
                                              F(g))

    --- F(edge_mor((a,e,b),(c,f,d),g))
                            <=
            edge_s_functor( (a,id(a),morphism(unit(e)),b,id(b)),
                            (c,id(c),morphism(unit(f)),d,id(d)),
                            F(g) )
    --- F(po_mor( (m1,m2), (l,c,r), (m3,m4) ))
                            <= SC_pushout ofm
            ( (F(m1),F(m2)), (F(l),F(c),F(r)), (F(m3),F(m4)) )
```

where the images under F of the irreducible graphs and irreducible graph morphisms are given by the obvious small categories and small functors.

The functor 'SC_pushout', which computes pushouts in the category of small categories is not yet defined. In fact, it is not available by the techniques so far given. We could write a program for this functor but we prefer to wait until a categorical formulation of the program is available (page 140). The construction of colimits in <u>SC</u> may be found in [Lawvere 1963b] and in general requires a rather complex iteration. We shall see later that such an iteration can be expressed as the construction of a free theory on a signature. We now close our discussion of these graphical problems for the moment.

Discussion

Notice in the above work that we have introduced two notions of the data type "graph". One is simply as objects in a category, actually a comma category:

```
    type Graph(alpha) ==                              - (1)
            Set(alpha) # Set_Mor(alpha) # Set(alpha)
```

The other is a constructive means of giving graphs together with their morphisms .

```
    data Graph1(alpha) ==
            nil ++
            node(alpha) ++                              - (2)
            edge(alpha,alpha,alpha) ++
            po_graph(Po_Diagram(Graph1(alpha),Graph1_Mor(alpha)))

    with Graph1_Mor(alpha) ==
            nil_mor(Graph1(alpha)) ++
            node_mor(alpha,alpha,Graph1(alpha)) ++
            edge_mor( (alpha#alpha#alpha),
                      (alpha#alpha#alpha),
                      Graph1(alpha)) ++
          po_mor(Po_Diagram_Mor(Graph1(alpha),Graph1_Mor(alpha)))
```

In programming terms, the second, "constructive", way of expressing data type is the more familiar - for instance "lists" are usually expressed in this form and it is this form that has lead to the association of data types with initial or free algebras.

The formal relationship between these two notions of a data type is that of a language and its semantics - a categorical semantics, of course. We sketch how this may be expressed in algebraic semantics. The following explains how the carrier of a data type can be interpreted as a category and thus that structure on a data type can be translated into structure on a category (e.g. colimit structure).

The expressions in (2) determine a free two-sorted theory i.e a free theory on the category <u>Set</u> x <u>Set</u>, (in fact this construction can be achieved computationally as we shall see). Thus given a pair of sets (the nodes and edges), terms on this pair are expressions for graphs and for graph morphisms derived from the data declaration above (2). The composition of this theory is simply term composition.

In algebraic semantics, the semantic function is interpreted as a theory morphism. In our case the target theory (of denotations) of the semantic function, is also to be a theory on <u>Set</u> x <u>Set</u>. However, we are also claiming that the category of graphs is to be the domain of denotations. How do we reconcile these two views? Notice that a category can be thought of as a two sorted algebra. The sorts being the objects and morphisms of the category (we are ignoring questions of the "size" of sets or classes - this is all right in this case since we are dealing with finite graphs only). Thus a category is an algebra of a theory on Set x Set. Thus the category of graphs can be interpreted as a theory on Set x Set. What is composition in this theory? Given a graph of graphs we need to "flatten" this to a graph. This is the colimit of a diagram of graphs!

The semantic function is a theory morphism from the theory of graph expressions (and graph-morphism expressions) to the theory of the category of graphs. As a theory morphism it gives rise to an adjunction between categories of algebras. Indeed the free functor of this adjunction may be encoded - it is simply the functor F which appears in the discussion of connected component graphs, replacing "pushout(cLG)" with "pushout(colimit_cat_of_graphs)". Notice also that the definition of F as it stands may be read as a semantic function interpreting graph expressions as connected component graphs.

Somehow this looks like special pleading. Graphs occur both as the data type in question and as an integral part of the definition of colimits. Thus the composition in the theory is a colimit operation. Can we do a similar thing for other data types? Can we interpret type expressions as constructions in categories? Sets as a data type can be so interpreted, as sets of sets of elements can be flattened to sets of elements (by a coproduct) and we should be able to handle lists likewise. As for the generality, I do not know.

How close is this, albeit rather limited, scheme for program development to a genuine automatic synthesis of programs from the text of a specification? The problem is twofold. Firstly we have used no formal apparatus to handle specifications, only an intuitive knowledge of what a specification means. Therefore we have not synthesised programs from the text of specifications. In the remaining chapters of this work we are to look at specifications from a formal viewpoint. Notice that the implementation of the semantics of a specification language is part of what is needed for the automation.

Moreover we have produced the required colimits by "fiat". A perusal of the text of the specifications of the problems reveals that connected component graphs can be considered as objects of a comma category (there is one operation and the equation is a 'preservation of structure') and hence that the colimits are computable by the techniques already available, whereas, in the case of the transitive closure of graphs, we can see that some iterative method of computing the colimits is required. Is it possible to formalise these observations? Given a theory, what do colimits of its algebras look like? The "monadic" theories of the next chapter give us one answer but do not provide a direct means of computing these colimits (but see [Adamek,Koubek 1980] for a construction of colimits of algebras).

Notes

Colimit recursion was introduced in [Burstall 1980] though no non-trivial example had been worked through.

CHAPTER SIX

THEORIES

In this chapter we look at various notions of "theories". Eventually we concentrate our attention on the so-called "monadic theories".

Like categories, theories are an abstraction of the properties of functions. Categories deal with all "functions" of a certain kind between objects in a given class. Theories, on the other hand, are intended to handle a small collection, often finite, of specific functions. We need this notion of a theory in the analysis of programs and of specifications simply because programs, and parts of programs (program modules and abstract data types), as well as specifications contain a small collection of "functions" or "operations".

The abstraction in the case of theories includes the composition of functions and their arities. We do not include function application as this is a special property of functions on sets. There are several ways that this abstraction may give rise to a mathematical formulation of theories. We look briefly at some of these.

What we may call a 'theory in a logical form' consists of a signature - that is, a set of sorts (simply names) and operations (or "functions" - again simply names) together with arities of the operations in terms of the sorts. Along with the signature we have "constraints". These constraints may be universally quantified equations, clauses in predicate calculus or something else. We are not interested in the constraints per se, only in the way they restrict the behavior of the operations. Thus two sets of constraints may well entail the same behavior of the operations. We handle this by insisting that the set of constraints should be in some way 'maximal' or 'closed'. An abstract formulation of these logical theories in which signatures are objects in a category and the constraints are sentences in a language may be found in [Burstall,Goguen 1980a].

Another approach is to consider the "terms" derived from a signature (in the form of sorts and operations with arities). If X is a set of variables indexed by the sorts in the signature, then we may form terms on X as follows:

1. If x is a variable in X, x is a term (of sort the index of x).

2. If $\sigma: s_1 s_2 s_3 \ldots s_n \to s$ is an operation in the signature and $t_1, t_2, \ldots t_n$ are terms of sort $s_1, s_2 \ldots s_n$ respectively, then $\sigma(t_1, t_2, \ldots t_n)$ is a term of sort s.

How do we handle possible constraints? Suppose that the constraints, as in the case of universally quantified equations, give an equivalence relation on the terms. Then instead of dealing with the constraints directly we can deal with the equivalence classes of terms under the equivalence relation. This restriction on the type of constraints allowed is quite severe. However two things might be said: Theories with constraints of this type have some special properties (for instance, the existence of initial algebras). Moreover, from the examples of the last chapter, we might speculate that universal conditions on specifications can replace constraints that do not behave like equations (there is some theoretical justification for this view).

The properties of these (equivalence classes of) terms can be abstracted to yield two "categorical" formulations of theories - either as derived from categories (Lawvere theories) or as derived from functors (monadic theories).

The first, due to Lawvere [1963b], arises from the observation that the composition of terms (or of equivalence classes of terms) has formally the properties of composition in a category. Thus, the suggestion is that a theory is to be a category whose objects are the arities of the terms and whose morphisms are to be the the equivalence classes of terms. To handle the arities and the projection functions on the products of sorts (or coproducts, if we take the dual category, as is usual), we have another category whose objects are again the arities but whose morphisms are these projection functions. Thus, if $\underline{A}$ is the category of arities and projection functions and $\underline{C}$ is any category, a Lawvere theory is a functor,

$F : \underline{A} \rightarrow \underline{C}$,

which preserves finite coproducts and whose object part is an isomorphism. We could consider this to be simply the category $\underline{C}$ with distinguished morphisms - those which are the image of morphisms in $\underline{A}$. This formulation of theories as categories has the advantage that higher-order theories can be described using standard category theoretic apparatus as 'cartesian closed categories'. Topoi provide a further extension to theories with a distinguished sort of 'truth values' and thus an internal logic [Mac Lane 1975].

A formulation in a different direction is that of monads or monadic theories. It is with monadic theories that we shall be working in the remaining chapters of this paper. Monadic theories arise from the following observation. Given a presentation of an equational theory, the map which takes a set of variables onto the set of equivalence classes of all terms is functorial. We may define a morphism part as substitution of variables in terms.

Thus an equational presentation gives a functor,

$T : \underline{Set} \rightarrow \underline{Set}$.

Monadic theories are defined in terms of a functor and include a "composition" as follows:

Definition

A monadic theory (or a monad or, in the sequel, simply a theory) $\underline{T}$ on a category $\underline{C}$ is a triple $\underline{T} = (T, \eta, \mu)$ where T is an endofunctor on $\underline{C}$ and η and μ are natural transformations, $\eta : I \dot{\rightarrow} T$ and $\mu : T^2 \dot{\rightarrow} T$ (I being the identity functor on $\underline{C}$), such that the following diagrams commute:

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[dr, "="'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T\eta"'] \arrow[dl, "="] \\
 & T &
\end{tikzcd}
\qquad
\begin{tikzcd}
T^3 \arrow[r, "\mu T"] \arrow[d, "T\mu"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

(These are diagrams in the category of functors from $\underline{C}$ to $\underline{C}$.)

The natural transformation "η" is called the unit of the theory and "μ" its composition. The second diagram then says that the composition is "associative". $\square$

Morphisms of these theories are natural transformations between the endofunctors. Formally, if $\underline{T} = (T,\eta,\mu)$ and $\underline{T}' = (T',\eta',\mu')$ are theories on category $\underline{C}$ then a morphism $\alpha : \underline{T} \to \underline{T}'$ is a natural transformation $\alpha : T \dot{\to} T'$ such that the following commute.

$$\begin{tikzcd}
 & I \arrow[dl, "\eta"'] \arrow[dr, "\eta'"] & \\
T \arrow[rr, "\alpha"'] & & T'
\end{tikzcd}
\qquad
\begin{tikzcd}
T^2 \arrow[r, "\alpha \circ \alpha"] \arrow[d, "\mu"'] & T'^2 \arrow[d, "\mu'"] \\
T \arrow[r, "\alpha"'] & T'
\end{tikzcd}$$

We can form a category of these monadic theories, $\underline{\text{Th}}(\underline{C})$, by defining the identity on the theory $\underline{T}$ to be the identity natural transformation, $i : T \dot{\to} T$, and the composition as the vertical composition of natural transformations.

As suggested above finitary equational theories can be expressed as monadic theories. Later, we extend monadic theories to cover other types of theories, in particular the data theories of [Burstall,Goguen 1980b]. For a detailed exposition of monadic theories consult [Manes 1976].

Algebras of monadic theories are defined as follows. If T = (T,η,μ) is a theory on C, then a T-algebra is a pair (a,ϵ) with 'a' an object of C and ϵ : aT -> a (the "structure map"), such that the following diagrams commute:

$$\begin{tikzcd}
a \arrow[r, "a\eta"] \arrow[dr, "="'] & aT \arrow[d, "\epsilon"] \\
 & a
\end{tikzcd}
\qquad
\begin{tikzcd}
aT^2 \arrow[r, "a\mu"] \arrow[d, "\epsilon T"'] & aT \arrow[d, "\epsilon"] \\
aT \arrow[r, "\epsilon"'] & a
\end{tikzcd}$$

A morphism of T-algebras from (a,ϵ) to (b,γ) is a C-morphism, f : a -> b, such that

$$\begin{tikzcd}
aT \arrow[r, "fT"] \arrow[d, "\epsilon"'] & bT \arrow[d, "\gamma"] \\
a \arrow[r, "f"'] & b
\end{tikzcd}$$

commutes. These algebras then form a category Alg(T).

We have discussed several ways of expressing theories as mathematical objects. An alternative 'axiomatic' approach to theories is suggested in [Goguen,Burstall 1978]. In order that a semantics of the algebraic specification language CLEAR could be formulated, certain properties of theories were required. These properties were then abstracted as axioms. We give a rough statement of a variant of these axioms as follows(a precise statement of the axioms may be found in the above paper).
There are two categories Th and Sig (of theories and signatures respectively) such that:

1. There is an adjunction (T, U,η,ϵ) : Sig -> Th.

2. Sig and Th are both finitely cocomplete.

3. Th has extremal-epi,mono factorisations.

4. With each object of Th (each theory) is associated a category (of algebras) - this association being an indexed category. Moreover, each theory morphism gives rise to a free functor on algebras and a collection of algebras of a theory give rise to another theory with certain properties (page 157).

The adjunction of (1) enables us to generate theories from signatures - the free theory on the signature - and express other theories as quotients of free theories. The existence of colimits in (2) gives us the means of fitting theories (or signatures) together to get new theories (or signatures). The factorisation in the category of theories is connected with data abstraction facilities. Finally (4) allows us to restrict the interpretation of specifications to algebras free in some sense and, thus, we can handle the so-called data theories of [Burstall,Goguen 1980b]. The following chapters will explain these things in detail. We will eventually show that the axioms hold for monadic theories.

Why should we choose to investigate monadic theories? These theories have several special properties which will be of use to us. Firstly, by choosing the base category of the theories to be the category of sets we get the usual set-based theories. Other choices of base category give us other useful theories. For instance, choosing a category of continuous partial orders gives us the continuous theories of [Wagner,Wright,Goguen,Thatcher 1978]. This means that constructions on monadic theories will handle both these cases uniformly. In particular, we will get free continuous theories by the same technique by which we get free set-based theories. Moreover, we will find that some of the constructions used can be translated into programs by the techniques we have developed. This gives us further insight into the relation of specifications to programs.

Another reason for looking at monadic theories is their close relation to data types. Data types in programs normally have an associated property of initiality or freeness. This can be interpreted in categorical terms as an adjunction. Indeed it was one of the early triumphs of category theory to correctly formalise the notion of freeness [Kan 1958]. The following theorem explains the connection between monadic theories and adjunctions.

Theorem

Every adjunction gives rise to a monadic theory and conversely every monadic theory gives rise to an adjunction.

(In fact, this relationship is an adjunction between the dual of the category of monadic theories on a category and the category of adjunctions on this category.)

Construction

Let $\underline{A}$ and $\underline{B}$ be categories and $(F,G,\eta,\epsilon) : \underline{A} \to \underline{B}$ be an adjunction. Then $(FG,\eta, F\epsilon G)$ is a monadic theory on $\underline{A}$.

Conversely, let $\underline{T} = (T,\eta,\mu)$ be a monadic theory on $\underline{A}$. Then define an adjunction $(F,G,\eta,\epsilon) : \underline{A} \to \underline{Alg}(\underline{T})$ where $\underline{Alg}(\underline{T})$ is the category of $\underline{T}$-algebras, by:

$aF = a\mu : (aT)T \to aT$

and if $f : a \to b$ then $fF = fT$. G is the functor mapping an algebra onto its carrier. η is the unit of the theory and, if $\gamma : aT \to a$ is a $\underline{T}$-algebra, then

$\gamma\epsilon : (a\mu : aTT \to aT) \to (\gamma : aT \to a)$

is defined to be the algebra morphism $\gamma : aT \to a$.

The checking of properties and the remainder of the proof is routine. It may be found in [Schubert 1972]. □

The two approaches to theories - the functorial approach of monads and the categorical approach due to Lawvere - are not quite as disparate as they may seem. There is an intermediate notion due to Kleisli [1965] which converts a monad into a category as follows.

Definition

Let $\underline{T} = (T,\eta,\mu)$ be a monad on category $\underline{C}$. The Kleisli category of $\underline{T}$, denoted $\underline{C}(\underline{T})$, is defined as follows:

The objects of $\underline{C}(\underline{T})$ are the objects of $\underline{C}$ whilst the morphisms of $\underline{C}(\underline{T})$, $f : a \to b$, are morphisms of $\underline{C}$ of the form $f : a \to bT$.

The identity on an object a is $a\eta : a \to aT$. Composition uses the composition μ in the monad. If $f : a \to b$ and $g : b \to c$ in $\underline{C}(\underline{T})$, define fg in $\underline{C}(\underline{T})$ by the morphism in $\underline{C}$:

$f.gT.c\mu : a \dashrightarrow bT \dashrightarrow cT^2 \dashrightarrow cT$

The theory laws of $\underline{T}$ establish the category laws of $\underline{C}(\underline{T})$. $\square$

Notice that there is a functor $F : \underline{C} \to \underline{C}(\underline{T})$ which is the identity on objects and takes a morphism $f : a \to b$ into the composition (in $\underline{C}$) $f.b\eta : a \to bT$. This functor has the following rather special property: if $\underline{C}$ has finite coproducts, then this functor preserves them. We thus have that, formally at least, for any $\underline{T}$ over $\underline{C}$, $F : \underline{C} \to \underline{C}(\underline{T})$ is a Lawvere theory.

However neither the choice of $\underline{C}$ as <u>Set</u> or as <u>FinSet</u> (the category of finite sets) will give the standard interpretation of a Lawvere theory. We need the following observation.

Let $\underline{T}$ be a monadic theory on $\underline{C}$, $\underline{J}$ a category with finite coproducts and $H : \underline{J} \to \underline{C}$ a functor which preserves these finite coproducts. Then we can restrict the Kleisli construction to the category $\underline{J}$ as follows. Construct a new category, denoted $H(\underline{T})$, whose objects are those of $\underline{J}$ and whose morphisms $f : a \to b$ are $\underline{C}$-morphisms of the form $f : aH \to bHT$. Identity and composition are similar to those of the Kleisli construction. As before there is a functor

$F : \underline{J} \to H(\underline{T})$, - (*)

which preserves finite coproducts.

Now let $\underline{J}$ be <u>FinSet</u> (or its skeleton) and let $\underline{C}$ be <u>Set</u>, H the inclusion, and $\underline{T}$ a theory on <u>Set</u>. Then (*) is the Lawvere theory associated with $\underline{T}$. It contains only finite sets of variables yet possibly infinite sets of terms.

Colimits in Kleisli categories are of some independent interest. As mentioned above, the functor $F : \underline{C} \to \underline{C}(\underline{T})$ for any monad $\underline{T}$ on $\underline{C}$ preserves finite coproducts. The same holds true for the initial object. Coequalisers are of more interest. Even if $\underline{C}$ has coequalisers, in general, $\underline{C}(\underline{T})$ does not. Moreover coequalisers that do exist in $\underline{C}(\underline{T})$ can be interpreted computationally as the unification of terms - it is actually somewhat more general than this. Thus the definition of unification has been reduced to that of a colimit. Unfortunately this does not give a direct means of computing the unification of terms. We need the iterative methods arising from the free theory construction of chapter eight.

MONADIC THEORIES COMPUTATIONALLY

In the following pages we are to look at properties of monadic theories. Not only do we establish mathematical results but we also investigate these results in the light of our programming of category theory. To this end we need to express monadic theories in programming terms. We now do this and show also that the relationship between adjunctions and monadic theories described above can be encoded as a program.

First we look at adjunctions as a data type. We could give an adjunction as a quadruple consisting of a pair of functors (the left and right adjoints) and a pair of natural transformations (the unit and counit). Instead, we look at another, equivalent, definition of adjunctions.

Suppose (F,G,η,ϵ) is an adjunction with $F : \underline{A} \to \underline{B}$. Then for any object 'a' of $\underline{A}$ and 'b' of $\underline{B}$, and for any morphism $f : a \to bG$, there is a morphism $f^{\#} : aF \to b$ such that the following triangle commutes:

$$\begin{tikzcd}
a \arrow[d, "f"] & a \arrow[r, "a\eta"] \arrow[dr, "f"'] & aFG \arrow[d, dashed, "f^{\#}G"] & aF \arrow[d, dashed, "f^{\#}"] \\
bG & & bG & b
\end{tikzcd}$$

The passage from f to $f^{\#}$ can be extended to a functor from the comma category $(\underline{A},G)$ to the comma category $(F,\underline{B})$. This is part of the bijection associated with an adjunction. Thus there is an inverse functor. We can now define an adjunction in terms of these two functors as:

```
    data Adjunction(o,m,o1,m1) ==
       adjunction(Functor(o,m,o1,m1),      ! free functor
                  Functor(o1,m1,o,m),      ! forgetful functor
                  Functor((o#m#o1),Right_Comma_Mor(o,m,o1,m1),
                          (o#m1#o1),Left_Comma_Mor(o,m,o1,m1)),
                 Functor((o#m1#o1),Left_Comma_Mor(o,m,o1,m1),
                          (o#m#o1),Right_Comma_Mor(o,m,o1,m1)) )
                                           ! the "sharp" functor
                                           ! and its inverse
```

A morphism of adjunctions from (F, G, η, ϵ) to (F', G', η', ϵ') is a pair of natural transformations (σ: F $\dot{-}$> F', τ: G' $\dot{-}$> G) which are conjugate, that is, the following diagram commutes (see [Mac Lane 1971] for details):

$$\begin{tikzcd}
I \arrow[r, "\eta"] \arrow[d, "\eta'"'] & FG \arrow[d, "\sigma G"] \\
F'G' \arrow[r, "F'\tau"'] & F'G
\end{tikzcd}$$

As a data type a morphism of adjunctions is:

```
   data Adj_Mor(o,m,o1,m1) ==
         adj_mor( Adjunction(o,m,o1,m1),
                  ( Nat_transform(o,m,o1,m1) #
                    Nat_transform(o1,m1,o,m) ),
                  Adjunction(o,m,o1,m1))
```

We can define a category of adjunctions in the now-familiar manner. We can also recover the unit and counit of the adjunction from our formulation as follows:

```
    dec unit : Cat(o,m)#Cat(o1,m1) ->
         (Adjunction(o,m,o1,m1) -> Nat_transform(o,m,o,m))
    dec counit : Cat(o,m)#Cat(o1,m1) ->
         (Adjunction(o,m,o1,m1) -> Nat_transform(o1,m1,o1,m1))

    --- unit(A,B) <=
          (lambda adjunction(F,G,_,invsharp) =>
        nat_transform(I(A),
                     (lambda a => eta_a
                         where (_,eta_a,_) ==
                           (invsharp ofo
                             (a,identity(B)(F ofo a),F ofo a))),
                     F.G) )

    --- counit(A,B) <=
          (lambda adjunction(F,G,sharp,_) =>
        nat_transform(G.F,
                     (lambda b => epsilon_b
                        where (_,epsilon_b,_) ==
                          (sharp ofo
                             (G ofo b,identity(A)(G ofo b),b))),
                     I(B)) )
```

A monadic theory, as a data type, is an endofunctor and two natural transformations (the unit η and composition μ):

```
    data Monad(o,m) ==
            monad( Functor(o,m,o,m),
                   Nat_transform(o,m,o,m),
                   Nat_transform(o,m,o,m))
```

Finally the passage from an adjunction to a monadic theory described above can be programmed:

```
    dec monadic: Cat(o,m)#Cat(o1,m1) ->
            (Adjunction(o,m,o1,m1) -> Monad(o,m))

    --- monadic(A,B) <=
            lambda Y & adjunction(F,G,_,_) =>
        monad( F.G, unit(A,B)(Y), F.counit(A,B)(Y).G )
```

There is a large body of results concerning the characterisation and properties of categories of algebras of monadic theories. Much less is known about categories of monadic theories themselves. In the following chapters we are to establish some properties of monadic theories, showing that the four axioms on page (103) hold for these theories and looking at the consequences of this in terms of our programming of category theory.

Constructions in category theory often include a good deal of verification that certain diagrams commute, by rather routine "diagram chasing". We therefore start with a chapter containing results which are intended to make the proofs more succinct and more readable.

Notes

Monadic theories were introduced by Godement (1958) where they were called "standard constructions". Eilenberg and Moore (1965) were the first to link monadic theories to universal algebra and noticed the relationship between adjunctions and monadic theories.

The idea of investigating monadic theories as a basis for program specification came from Dr. Gordon Plotkin.

The axioms required of theories which are to be denotations of CLEAR specifications are found in [Goguen and Burstall 1978].

CHAPTER SEVEN
COLIMIT CONSTRUCTIONS

In this chapter we give some general results from category theory. In the first section we review some properties of colimits in functor categories. In the next we see how the universal properties of colimits may be translated into 'proof rules'.

COLIMITS AND FUNCTORS

Our interest lies with colimits in various categories of functors. We shall be constructing monadic theories using colimits of functors. Properties of these theories will then be determined by the properties of colimits in functor categories. The following results are all of the form: given a diagram of functors all of which have the property, P, then the colimiting object of the diagram, itself a functor, also has the property P. Further details and proofs will be found in [Schubert 1972].

Theorem

If $\underline{A}$, $\underline{B}$ are categories with $\underline{B}$ cocomplete (hence $\underline{B}^{\underline{A}}$ is cocomplete - see page (52)) and $\underline{C}$ is the full subcategory of $\underline{B}^{\underline{A}}$ whose objects are functors that preserve colimits that exist in $\underline{A}$, then $\underline{C}$ is cocomplete and the colimits in $\underline{C}$ are those in $\underline{B}^{\underline{A}}$. $\Box$

Thus, in particular, if $\underline{A}$ is cocomplete, then the colimiting functor of a diagram of cocontinuous functors is itself cocontinuous.

This result may be tightened somewhat:

Theorem

If $\underline{A}$,$\underline{B}$ are categories with $\underline{B}$ cocomplete and $\underline{C}$ is the full subcategory of $\underline{B}^{\underline{A}}$ of functors that preserve ω-colimits (that exist in $\underline{B}$) then $\underline{C}$ is cocomplete and closed with respect to colimits in $\underline{B}^{\underline{A}}$. $\Box$

We may replace ω-colimits by pushouts, binary coproducts etc. More generally, we can extend the theorem to classes of "diagram schemes with commutativity conditions". Moreover, we may restrict $\underline{B}$ to be, say, ω-cocomplete and then the theorem says that $\underline{C}$ has the same cocompleteness property as $\underline{B}$. Another result of a similar form is:

Theorem

If D is a diagram of functors all of which preserve the colimit of a diagram of functors D', then the colimiting functor of D preserves the colimit of D'. $\square$

These results are all corollaries of the following theorem:

Theorem (Interchange of Colimits)

If $\underline{P}$ and $\underline{Q}$ are two (small) categories and $\underline{C}$ any category and $D : \underline{P} \times \underline{Q} \to \underline{C}$ is a diagram in $\underline{C}$, then the colimit of D is computed componentwise as follows: For each object p in $\underline{P}$, consider the functor $(p,\_)D : \underline{Q} \to \underline{C}$. Let its colimit, if it exists, be $\zeta_{p,q} : (p,q)D \to c_p$ for each q in $\underline{Q}$. The map $p \mapsto c_p$ extends to a functor $G : \underline{P} \to \underline{C}$ whose morphism part arises from the universal part of the colimit. Then D has a colimit iff G exists and has a colimit. Moreover, if $\gamma_p : pG \to c$ is the colimit of G, then the colimit of D is given by the composition:

$$\zeta_{p,q}\cdot\gamma_p : (p,q)D \to c$$

for each pair of objects (p,q) in $\underline{P} \times \underline{Q}$. $\square$

Note that this is constructive in the sense that we may encode it as a program to compute double colimits. See the reference above for further details of theorems of this form.

We see from these results that colimits in functor categories are as well-behaved as could be wished. Results of this type are not merely of a theoretical interest, they correspond to results in programming and specification. A typical application of such results would be to prove syntactic identities in algebraic specifications. Details of this may be found in [Ehrich 1978].

COLIMITS AND PROOF RULES

In the following chapters extensive use will be made of colimit constructions. Their universal properties will be used to prove properties of the colimiting objects. The contents of this, rather technical, section are designed to make these proofs easier and more succinct by translating the universal property of a colimit into proof rules. When we turn to colimit constructions we will find that much of the proof is routine "diagram chasing" - verifying that certain diagrams commute. It ought to be automatic. The casting of properties of colimits into proof rules shows how this automation might go.

The idea is to use the fact that the colimiting object of a diagram can be thought of as "approximated" by the objects of the diagram. This gives us a means of reducing proofs of properties of the colimiting objects to proofs of properties of the objects in the diagram.

The important facts are the following two lemmas and the idea that they may be extended to more complicated cases.

Let D be a diagram in a category $\underline{C}$ which has colimits of D-shaped diagrams. Let the objects in D be a_i (i ranging over a suitable index set). Let the colimit of D be $\xi_i : a_i \to a$. Then we have the following proof rule:

Lemma (1-approximation)

To show that f : a -> b is equal to g : a -> b, it suffices to show that, for all i:

$$\begin{tikzcd}
a_i \arrow[r, "\xi_i"] \arrow[d, "\xi_i"'] & a \arrow[d, "f"] \\
a \arrow[r, "g"'] & b
\end{tikzcd}$$

commutes.

Proof

Immediate consequence of the universality of the colimit. □

As a proof rule this is not very powerful. It encodes only the uniqueness part of the universality and not the existence. For instance, in the category of sets, it applies not only to the coproduct of two sets, but also to their union. However, it provides a useful means of presenting proofs. The full universality of the colimit is given in the following lemma which is merely a statement of the colimiting property of colimits. It is included because we intend to extend it to more general cases.

Lemma (1-existence)

If $h_i : a_i \to b$ form a cone on D then there is a unique $h : a \to b$ such that for all i:

$$\begin{tikzcd}
a_i \arrow[r, "\zeta_i"] \arrow[dr, "h_i"'] & a \arrow[d, dashed, "h"] \\
 & b
\end{tikzcd}$$

commutes. ◻

The approximation lemma above (which, like the existence lemma, has a prefix of '1' because there are more to come) grew out of our toils in proving properties of objects defined by colimits of ω-chains which were generated by the successive application of an endofunctor to an object and a morphism. In this context it turned out that the extra structure in the category of endofunctors on some category, that of a composition, gave us further approximation and existence lemmas. We present these lemmas in an abstract form by introducing a general "composition" on objects and morphisms of a category. In our applications we will, however, only be using the case of endofunctors under composition.

Definition

$\underline{C}$ is a strict monoidal category with composition $\square$ and identity e if:

1. $\square$ is a bifunctor $\square : \underline{C} \times \underline{C} \to \underline{C}$

2. $\square$ is associative both on objects and morphisms :

$\Box(\Box \times 1) = \Box(1 \times \Box) : \underline{C} \times \underline{C} \times \underline{C} \to \underline{C}$ (where we identify $(\underline{C} \times \underline{C}) \times \underline{C}$ and $\underline{C} \times (\underline{C} \times \underline{C})$).

3. e is the left and right identity for $\Box$:
   $\Box(e \times 1) = id_{\underline{C}} = \Box(1 \times e)$.

Notice that associativity and identity ensure the following rules:

$i_a \Box i_b = i_{a \Box b}$ for a,b in $\underline{C}$

$(f' \Box g')(f \Box g) = (f'f) \Box (g'g)$

whenever f'f and g'g are defined.

The second is the interchange law. Thus for any a,b in $\underline{C}$, $a \Box b$ is an object of $\underline{C}$, and for any morphisms $f : a \to a'$, $g : b \to b'$ in $\underline{C}$ there is a morphism

$f \Box g : a \Box b \to a' \Box b'$.

Following usual conventions, we will often write $f \Box b$ for the morphism $f \Box i_b$ and $a \Box g$ for $i_a \Box g$. $\square$

This definition occurs in [Mac Lane 1971]. The example we have in mind is that $\underline{C}$ is the category of endofunctors on some category, $\Box$ is the functor composition and e the identity functor.

Now, to progress further we need to make assumptions about the relationship of $\Box$ to the colimits in $\underline{C}$, where $\underline{C}$ is cocomplete (or, at least, has colimits of diagrams of a certain shape). Using the notation of the previous lemmas, we specify that

1. For all a_i in D, $a_i \Box \_$ preserves colimits (at least of D-shaped diagrams)

2. $\_ \Box a$ preserves colimits (again, at least of D-shaped diagrams)

Then we have the following lemmas.

Lemma (2-existence)

If $h_{i,j} : a_i \Box a_j \to b$ is such that for all a_i, a_j, $a_{i'}$, $a_{j'}$ in D and for all $f : a_i \to a_{i'}$ and $g : a_j \to a_{j'}$ in D, the following two triangles commute,

$$\begin{tikzcd}
a_i \Box a_j \arrow[rr, "f \Box a_j"] \arrow[dr, "h_{i,j}"'] & & a_{i'} \Box a_j \arrow[dl, "h_{i',j}"] \\
& b &
\end{tikzcd}
\qquad
\begin{tikzcd}
a_i \Box a_j \arrow[rr, "a_i \Box g"] \arrow[dr, "h_{i,j}"'] & & a_i \Box a_{j'} \arrow[dl, "h_{i,j'}"] \\
& b &
\end{tikzcd}$$

(In this case we say that $h_{i,j}$ is a cone on D separately in i and j.)

then there is a unique h : a $\Box$ a -> b such that, for all i,j (with a_i, a_j in D),

$$\begin{tikzcd}
a_i \Box a_j \arrow[r, "\zeta_i \Box \zeta_j"] \arrow[dr, "h_{i,j}"'] & a \Box a \arrow[d, dashed, "h"] \\
& b
\end{tikzcd}$$

commutes.

Proof

Construct such an h as follows:

Consider the diagram $a_i \Box D$. Its colimit is

$a_i \Box \zeta_i$: $a_i \Box a_j$ -> $a_i \Box$ a

(as $a_i \Box$ _ preserves colimits).

Now the $h_{i,j}$ form a cone on $a_i \Box D$ so there is a unique h_i : $a_i \Box$ a -> b such that

$$\begin{tikzcd}
a_i \Box a_j \arrow[r, "a_i \Box \zeta_j"] \arrow[dr, "h_{i,j}"'] & a_i \Box a \arrow[d, dashed, "h_i"] \\
& b
\end{tikzcd}$$

commutes. These h_i form a cone on $D \square a$, that is, for all a_i, a_j in D and $f : a_i \to a_j$ in D

$$\begin{tikzcd}
a_i \square a \arrow[rr, "f \square a"] \arrow[dr, "h_i"'] & & a_j \square a \arrow[dl, "h_j"] \\
& b &
\end{tikzcd}$$

commutes.

To show this, we use the 1-approximation lemma. Thus it suffices to show that the following commutes for all a_i, a_j, a_k and $f : a_i \to a_j$ in D:

$$\begin{tikzcd}
a_i \square a_k \arrow[rr, "a_i \square \xi_k"] \arrow[dd, "a_i \square \xi_k"'] \arrow[dr, "f \square a_k"] \arrow[ddrr, bend right, "h_{i,j}"'] & (1) & a_i \square a \arrow[d, "f \square a"] \\
(4) & a_j \square a_k \arrow[r, "a_j \square \xi_k"] \arrow[dr, "h_{j,k}"] & a_j \square a \arrow[d, "h_j"] \\
a_i \square a \arrow[rr, "h_i"'] & (3) \quad (2) & b
\end{tikzcd}$$

But (1) is the interchange law, (2) is the definition of h_j, (3) commutes because $h_{i,j}$ is a cone in the first argument and (4) by the definition of h_i.

Thus the h_i form a cone on $D \square a$ and, as $\_ \square a$ preserves colimits (of D-shaped diagrams), we have: there is a unique $h : a \square a \to b$ such that ation)

a be the colimit of diagra

d g : a $\square$ a -> b, to prove that

a_i, a_j in D:

$$\begin{tikzcd}
\square a_j \arrow[r, "\zeta_i \square \zeta_j"] \arrow[d] & a \square a \arrow[d, "g"] \\
\square a \arrow[r, "f"'] & b
\end{tikzcd}$$

commutes

$$\begin{tikzcd}
a_i \Box a_j \Box a_k \arrow[r, "\zeta_i \Box \zeta_j \Box \zeta_k"] \arrow[d, "\zeta_i \Box \zeta_j \Box \zeta_k"'] & a \Box a \Box a \arrow[d, "g"] \\
a \Box a \Box a \arrow[r, "f"'] & b
\end{tikzcd}$$

commutes, then f = g. $\Box$

The previous lemmas can be stated in a weaker form if we make the assumption that the diagram D is both <u>commutative</u> and <u>path-connected</u>. Path-connectedness means that for any two objects a_i and a_j in D there is a directed path of morphisms in D either from a_i to a_j or from a_j to a_i. ω-chains in particular have this property. The weakened version of the 2-approximation lemma using the same colimit preserving properties of $\Box$, are:

<u>Lemma</u> (2'-existence)

If for each i with a_i in D, $h_i : a_i \Box a_j \to b$ has the following property: for all a_i and a_j in D and for all $d,e : a_i \to a_j$ in D,

$$\begin{tikzcd}
a_i \Box a_i \arrow[rr, "d \Box e"] \arrow[dr, "h_i"'] & & a_j \Box a_j \arrow[dl, "h_j"] \\
& b &
\end{tikzcd}$$

commutes, then there is a unique $h : a \Box a \to b$ such that

$$\begin{tikzcd}
a_i \Box a_i \arrow[r, "\zeta_i \Box \zeta_i"] \arrow[dr, "h_i"'] & a \Box a \arrow[d, dashed, "h"] \\
& b
\end{tikzcd}$$

commutes.

The proof constructs a cone on $a_i \square D$ using the h_i and paths in D. □

Again an approximation lemma follows from the existence lemma:

Lemma (2'-approximation)

If for all a_i in D,

$$\begin{tikzcd}
a_i \square a_i \arrow[r, "\zeta_i \square \zeta_i"] \arrow[d, "\zeta_i \square \zeta_i"'] & a \square a \arrow[d, "g"] \\
a \square a \arrow[r, "f"'] & b
\end{tikzcd}$$

commutes, then f is equal to g. □

This completes our preparation. We now turn to various categorical constructions using the above results to help us with the proofs.

Notes

The properties of colimits of functors are standard category theory.

The approximation lemmas were introduced in a restricted form (for ω-chains generated by a "signature" - see the next chapter) in [Burstall and Rydeheard 1979].

CHAPTER EIGHT

FREE THEORIES

For equational theories and various extensions of equational theories (for example continuous theories and Lawvere theories) there is an important result which says, roughly, that we can define signatures for these theories in such a way that, for each signature, a free theory on the signature exists.

The existence of free theories is of interest for several reasons. It provides an interpretation of the syntax of a language and then the 'unique extension lemma' associated with the free theory gives a semantics of the language. Free and initial algebras arise directly from free theories and they allow us to formalise the notion of a data type. Moreover, as we shall see, free theories are of importance in understanding the structure and development of programs.

Our work here is to show that we can define signatures (and presentations) in such a way that free monadic theories exist on these signatures. This then will provide a uniform means of constructing free theories for several cases for which ad hoc methods have been known, for instance for continuous theories (see [Wagner,Wright,Goguen,Thatcher 1978]). Notice also that this is Axiom (1) of the requirements for theories to be of use in algebraic specifications.

The construction is by means of an ω-colimit in a category. This in itself is not new. The connection between ω-colimits and free theories (and also initial algebras) has appeared in various guises. Lehmann and Smyth [1977] and Smyth and Plotkin [1977] as well as others have used ω-colimits to construct initial algebras in categories of continuous partial orders. Adamek and Trnkova [1978] have investigated the case when the functor generating the chain is not ω-cocontinuous. Barr [1970] has given necessary and sufficient conditions that a functor "generates" a free theory whilst Dubuc [1974] has generalised the construction from categories of functors to arbitrary monoidal categories. The novelty here is the definition of signatures in the framework of monadic theories, noting that this handles several disparate cases uniformly, and the investigation of the computational aspects of this construction.

We now define signatures.

Definition

A signature (in a monadic form) on a category $\underline{C}$ is a pair $\underline{\Sigma} = (\Sigma, \sigma)$ where Σ is an endofunctor on $\underline{C}$ and σ is a natural transformation, $\sigma : I \overset{\cdot}{\to} \Sigma$ (I is the identity functor on $\underline{C}$). □

Thus a signature differs from a theory in that it has no composition within it. A signature of the usual type - a collection of sorts and operations on the sorts - can be understood as a signature in a monadic form as the following example illustrates.

Consider the signature of the theory of natural numbers with addition:

```
sorts nat
opns  0 : nat
      succ : nat -> nat
      add : nat,nat -> nat
```

This gives a signature in a monadic form on the category of sets as follows. The endofunctor Σ is given by:

$$a\Sigma = a + 1 + a + a\text{x}a$$

for any set 'a'. Here 1 is the (one element) terminal set, x the product of sets and + the coproduct of sets. The morphism part of Σ is defined exactly as the object part, that is, if f is a morphism of sets,

$$f\Sigma = f + 1 + f + f\text{x}f.$$

The natural transformation, $\sigma : I \overset{\cdot}{\to} \Sigma$, is defined such that, for any set 'a', $a\sigma$ is the morphism, arising from the coproduct, mapping a into the leftmost component of the coproduct. Notice that the endofunctor Σ is defined only up to an isomorphism. Notice further that we have dealt with signatures containing a single sort. Many sorted signatures and theories will be investigated later (page 190).

In the general case, we have, for each operation ρ in the signature, a component of the form a^n in the expression for $a\Sigma$, where n is the length of the arity of ρ, together with a component 'a' alone. Essentially then, Σ maps a set of variables into terms of depth at most one in the operations of the signature.

We can form a category of signatures by defining morphisms.

Definition

A signature morphism from $\underline{\Sigma} = (\Sigma, \sigma)$ to $\underline{\Sigma}' = (\Sigma', \sigma')$ (both signatures over a category $\underline{C}$) is a natural transformation, $\alpha : \Sigma \dot{\to} \Sigma'$, such that

$$\begin{tikzcd} & I \arrow[dl, "\sigma"'] \arrow[dr, "\sigma'"] & \\ \Sigma \arrow[rr, "\alpha"'] & & \Sigma' \end{tikzcd}$$

commutes. □

We can define the category of signatures and signature morphims on a category $\underline{C}$, $\underline{\text{Sig}}(\underline{C})$.

What about finite colimits in the category of signatures, $\underline{\text{Sig}}(\underline{C})$? Unlike colimits of monadic theories (page 173), these colimits are easy. The coproduct of two signatures (Σ, σ) and (Σ', σ') on a category $\underline{C}$ has endofunctor Σ'' given by the pushout:

$$\begin{tikzcd} I \arrow[r, "\sigma"] \arrow[d, "\sigma'"'] & \Sigma \arrow[d, dashed] \\ \Sigma' \arrow[r, dashed] & \Sigma'' \end{tikzcd}$$

where I is the identity functor on $\underline{C}$. The natural transformation σ'' of the coproduct signature $\underline{\Sigma}'' = (\Sigma'', \sigma'')$ is the diagonal of the pushout square. The initial signature on category $\underline{C}$ is (I,i) with I the identity on $\underline{C}$ and i the identity natural transformation in $\underline{C}$. Coequalisers of signatures on $\underline{C}$ are created by those in $\underline{C}^{\underline{C}}$. Thus, by the colimit existence theorem, if $\underline{C}$ has finite colimits so does

$\underline{\mathrm{Sig}}(\underline{C})$.

In a similar manner to defining algebras of theories we can define algebras of signatures. If $\underline{\underline{\Sigma}} = (\Sigma, \sigma)$ is a signature on $\underline{C}$, a $\underline{\underline{\Sigma}}$-algebra is an object 'a' of $\underline{C}$ together with a morphism $\epsilon : a\Sigma \to a$, such that

$$\begin{tikzcd}
a \arrow[r, "a\eta"] \arrow[dr, "="'] & a\Sigma \arrow[d, "\epsilon"] \\
& a
\end{tikzcd}$$

commutes.

There is a functor from theories to signatures which "forgets" the composition in the theories:

$U : \underline{\mathrm{Th}}(\underline{C}) \to \underline{\mathrm{Sig}}(\underline{C})$

given by: $(T,\eta,\mu)U = (T,\eta)$ on objects and $\alpha U = \alpha$ on morphisms. Our aim is to construct a free monadic theory on a signature, $\underline{\underline{\Sigma}}$, with respect to U. That is, a theory $\underline{T}$ together with a signature morphism $\alpha : \underline{\underline{\Sigma}} \to \underline{T}U$ such that, for any other theory $\underline{T}'$ and any signature morphism $\beta : \underline{\underline{\Sigma}} \to \underline{T}'U$, there is a unique morphism $\beta^{\#} : \underline{T} \to \underline{T}'$ such that

$$\begin{tikzcd}
\underline{\underline{\Sigma}} \arrow[r, "\alpha"] \arrow[dr, "\beta"'] & \underline{T}U \arrow[d, dashed, "\beta^{*}U"] \\
& \underline{T}'U
\end{tikzcd}$$

commutes.

As mentioned before, the construction of such a free theory on a signature, $\underline{\underline{\Sigma}}$, uses an ω-colimit in a category. Thus we need at least that this category has ω-colimits. In fact we need more - we need that the endofunctor of $\underline{\underline{\Sigma}}$ preserves ω-colimits. Fortunately, for signatures on the category of sets that can be expressed in the polynomial form above it is known that the endofunctor is ω-cocontinuous (see [Lehmann, Smyth 1977] for details and extensions to various categories of continuous partial orders). We now turn to the construction.

THE CONSTRUCTION OF A FREE MONADIC THEORY ON A SIGNATURE

We here show how a signature gives rise to a monadic theory.

Let $\underline{C}$ be a category which has colimits of ω-chains (is ω-cocomplete) and $\underline{\underline{\Sigma}} = (\Sigma, \sigma)$ a signature on $\underline{C}$ with Σ an endofunctor preserving all ω-colimits in $\underline{C}$.

We now construct a theory $\underline{T} = (T, \eta, \mu)$ together with the required signature morphism from $\underline{\underline{\Sigma}}$ to $\underline{T}U$, where U is the forgetful functor from theories to signatures. We will then show that $\underline{T}$ is free on $\underline{\underline{\Sigma}}$ with respect to U.

To define the functor T consider the diagram (which we call Δ);

$$I \xrightarrow{\sigma} \Sigma \xrightarrow{\sigma\Sigma} \Sigma^2 \cdot\cdot\ \Sigma^n \xrightarrow{\sigma\Sigma^n} \Sigma^{n+1} \longrightarrow \ \cdots \quad - (*)$$

(where I is the identity functor on $\underline{C}$).

Let T be the colimiting object of this chain, with colimiting cone,

$\zeta_n : \Sigma^n \overset{\cdot}{\to} T$

for $n \geq 0$. By a result in the previous chapter, T is ω-cocontinuous. The approximation lemmas arising from this colimit take the form:

The 1-approximation:

If, for all $n \geq 0$

$$\begin{tikzcd}
\Sigma^n \arrow[r, "\zeta_n"] \arrow[d, "\zeta_n"'] & T \arrow[d, "f"] \\
T \arrow[r, "g"'] & V
\end{tikzcd}$$

commutes then $f = g$. $\square$

Because T is ω-cocontinuous we have the 2-approximation:

If, for all $m,n \geq 0$

$$\begin{tikzcd}
\Sigma^m\Sigma^n \arrow[r, "\zeta_m\zeta_n"] \arrow[d, "\zeta_m\zeta_n"'] & T^2 \arrow[d, "f"] \\
T^2 \arrow[r, "g"'] & V
\end{tikzcd}$$

commutes, then $f = g$. $\square$

Likewise for the 3-approximation. The 2-existence lemma becomes in this context:

If $h_{m,n} : \Sigma^m\Sigma^n \to V$ is a cone on Δ separately in m and n, then there is a unique $h : T^2 \to V$ such that, for all $m,n \geq 0$,

$$\begin{tikzcd}
\Sigma^m\Sigma^n \arrow[r, "\zeta_m\zeta_n"] \arrow[dr, "h_{m,n}"'] & T^2 \arrow[d, dashed, "h"] \\
& V
\end{tikzcd}$$

commutes. $\square$

We now show that T is the endofunctor of a theory by defining η and μ.

<u>Definition</u>

η is defined to be $\zeta_0 : I_{\underline{C}} \overset{\cdot}{\to} T$ and

μ is defined to be the unique natural transformation, such that, for all $m,n \geq 0$,

$$\begin{tikzcd}
\Sigma^m\Sigma^n \arrow[r, "\zeta_m\zeta_n"] \arrow[dr, "\zeta_{m+n}"'] & T^2 \arrow[d, dashed, "\mu"] \\
& T
\end{tikzcd}$$

commutes (using the 2-existence lemma above).

Proposition

$\underline{T} = (T,\eta,\mu)$ is a theory.

Proof

We need to show that η and μ satisfy the commuting diagrams required of a theory. First, we need that

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[dr, "="'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T\eta"'] \arrow[dl, "="] \\
& T &
\end{tikzcd}$$

commutes. Recall that $\zeta_m\zeta_n.\mu = \zeta_{m+n}$ by the definition of μ. Set m=0 remembering that $\zeta_0 = \eta$ then we have

$$\begin{tikzcd}
\Sigma^n \arrow[rr, "\zeta_n"] \arrow[d, "\zeta_n"'] & & T \arrow[d, "="] \\
T \arrow[r, "\eta T"'] & T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

commutes for all $n \geq 0$. So by the 1-approximation lemma, $\eta T.\mu = i_T$. By a similar argument, taking n=0, we have $T\eta.\mu = i_T$. Secondly we need to show that the following commutes:

$$\begin{tikzcd}
T^3 \arrow[r, "\mu T"] \arrow[d, "T\mu"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

For this we use the ζ-approximation lemma. Thus it suffices to show that the square below commutes.

$$\begin{tikzcd}
\Sigma^l\Sigma^m\Sigma^n \arrow[rr, "\zeta_l\zeta_m\zeta_n"] \arrow[dd, "\zeta_l\zeta_m\zeta_n"'] & & T^3 \arrow[d, "\mu T"] \\
 & & T^2 \arrow[d, "\mu"] \\
T^3 \arrow[r, "T\mu"'] & T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

But we can show that either way round this square is ζ_{l+m+n}. Thus consider,

$$\begin{tikzcd}
\Sigma^l\Sigma^m\Sigma^n \arrow[rr, "\zeta_l\zeta_m\zeta_n"] \arrow[dr, "\Sigma^l\Sigma^m\zeta_n"'] \arrow[dd, "="'] & \text{(1)} & T^3 \arrow[d, "\mu T"] \\
 & \Sigma^l\Sigma^m T \arrow[ur, "\zeta_l\zeta_m T"] \arrow[r, "\zeta_{l+m}T"] \arrow[phantom, "\text{(3)}", ld] & T^2 \arrow[d, "\mu"] \arrow[phantom, "\text{(2)}", u] \\
\Sigma^l\Sigma^m\Sigma^n \arrow[urr, "\zeta_{l+m}\zeta_n"] \arrow[rr, "\zeta_{l+m+n}"'] & \text{(4)} & T
\end{tikzcd}$$

where (1) commutes by definition of $(\zeta_l\zeta_m)\zeta_n$, (2) by the definition of μ and the fact that T is a functor, (3) by the definition of

$\zeta_{l+m}\zeta_n$ and (4) by the definition of μ. An analogous argument holds for the bottom left part of the square.

The Freeness of the Theory

Having shown that $\underline{T}$ is a theory we show that it is the free theory on the signature $\underline{\Sigma} = (\Sigma, \sigma)$. First define a signature morphism,

$$\phi : (\Sigma, \sigma) \to (T, \eta, \mu)U$$

by $\phi = \zeta_1 : \Sigma \xrightarrow{.} T$. This is indeed a signature morphism since we have

$$\begin{tikzcd} I \arrow[r, "\sigma"] \arrow[dr, "\eta"'] & \Sigma \arrow[d, "\phi"] \\ & T \end{tikzcd}$$

commutes as $\eta = \zeta_0$ and $\phi = \zeta_1$.

Proposition

The pair $((T,\eta,\mu), \phi)$ is free on (Σ, σ) with respect to the forgetful functor U from theories to signatures.

Proof

We must show that for any other theory $\underline{T}' = (T', \eta', \mu')$ and any signature morphism $\psi : \underline{\Sigma} \to \underline{T}'U$, there is a unique theory morphism $\theta : \underline{T} \to \underline{T}'$ such that

$$\begin{tikzcd} \underline{\Sigma} \arrow[r, "\phi"] \arrow[dr, "\psi"'] & \underline{T}U \arrow[d, "\theta U"] \\ & \underline{T}'U \end{tikzcd}$$

commutes.

To construct $\theta : T \xrightarrow{.} T'$, we first define a sequence of natural transformations $\theta_n : \Sigma^n \xrightarrow{.} T'$, for any $n \geq 0$, by induction as follows:

$$\theta_0 = \eta' \text{ and } \theta_{n+1} = (\theta_n\psi).\mu'$$

We note that $\theta_1 = \psi.\eta'T'.\mu' = \psi$. To show that the θ_n form a cone, that is for all $n \geq 0$

$$\begin{tikzcd}
\Sigma^n \arrow[r, "\sigma\Sigma^n"] \arrow[dr, "\theta_n"'] & \Sigma^{n+1} \arrow[d, "\theta_{n+1}"] \\
 & T'
\end{tikzcd}$$

commutes, we use induction on n. The base case n=0 is

$$\begin{tikzcd}
I \arrow[r, "\sigma"] \arrow[dr, "\eta'"'] & \Sigma \arrow[d, "\gamma"] \\
 & T'
\end{tikzcd}$$

commutes as γ is a signature morphism. The inductive step follows from expanding θ_n and θ_{n+1} in the diagram above. We now define $\theta : T \stackrel{\bullet}{\to} T'$ to be the unique natural transformation from T, the colimit of Δ to T' such that for all $n \geq 0$

$$\begin{tikzcd}
\Sigma^n \arrow[r, "\zeta_n"] \arrow[dr, "\theta_n"'] & T \arrow[d, dashed, "\theta"] \\
 & T'
\end{tikzcd}$$

commutes.

Is θ a theory morphism? That is, do both the following diagrams commute?

$$\begin{tikzcd} I \arrow[r, "\eta"] \arrow[dr, "\eta'"'] & T \arrow[d, "\theta"] \\ & T' \end{tikzcd} \qquad \begin{tikzcd} T^2 \arrow[r, "\theta\theta"] \arrow[d, "\mu"'] & T'^2 \arrow[d, "\mu'"] \\ T \arrow[r, "\theta"'] & T' \end{tikzcd}$$

The first is easy as $\eta = \zeta_0$ and $\zeta_0 . \theta = \theta_0 = \eta'$ by definition. We use the 2-approximation lemma for the second. It suffices to show that the outer square below commutes.

$$\begin{tikzcd} \Sigma^m\Sigma^n \arrow[rr, "\zeta_m\zeta_n"] \arrow[dd, "\zeta_m\zeta_n"'] \arrow[dr, "\theta_m\theta_n"'] \arrow[drr, "\theta_{m+n}"] \arrow[rr, bend right=15, "\zeta_{m+n}"] & (4) & T^2 \arrow[d, "\mu"] \\ (1) & T'^2 \arrow[d, phantom, "(2)\ (3)"] & T \arrow[d, "\theta"] \\ T^2 \arrow[r, "\theta\theta"'] & T'^2 \arrow[r, "\mu'"'] & T' \end{tikzcd}$$

To see that (1) commutes recall the interchange law for the composition of natural transformations

$(\sigma\sigma').(\tau\tau') = (\sigma.\tau)(\sigma'.\tau')$

Thus

$(\zeta_m\zeta_n).(\theta\theta) = (\zeta_m.\theta)(\zeta_n.\theta) = \theta_m\theta_n$

Now (3) commutes by the definition of θ and (4) by the definition of μ. It remains to show that (2) commutes. We set this up as a lemma.

Lemma

For all $m,n \geq 0$,

$$\begin{tikzcd}
\Sigma^m\Sigma^n \arrow[r, "\theta_m\theta_n"] \arrow[dr, "\theta_{m+n}"'] & T'^2 \arrow[d, "\mu'"] \\
 & T'
\end{tikzcd}$$

commutes.

Proof

By induction on n. The base case, n=0, is

$$\begin{tikzcd}
\Sigma^m \arrow[r, "\theta_m"] \arrow[drr, "\theta_m"'] & T' \arrow[r, "T'\eta'"] & T'^2 \arrow[d, "\mu'"] \\
 & & T'
\end{tikzcd}$$

which commutes as T' is a theory. The inductive step is - assume the triangle commutes for n, then show that it does for n+1. Expanding the triangle for n+1, we get

$$\begin{tikzcd}
\Sigma^m\Sigma^n\Sigma \arrow[rr, "\Sigma^m\Sigma^n\psi"] \arrow[ddd, "\theta_{m+n+1}"'] & & \Sigma^m\Sigma^n T' \arrow[rr, "\Sigma^m\theta_n T'"] \arrow[dl, "\theta_{m+n}T'"'] \arrow[dr, "\theta_m\theta_n T'"] \arrow[d, phantom, "(2)"] & & \Sigma^m T'^2 \arrow[dl, "\theta_m T'^2"] \arrow[d, "\Sigma^m\mu'"] \\
 & T'^2 \arrow[l, phantom, "(1)"] \arrow[ddl, "\mu'"] & & T'^3 \arrow[ll, "\mu'T'"] \arrow[ur, phantom, "(3)"] \arrow[r, phantom, "(4)"] \arrow[ddr, "T'\mu'"'] & \Sigma^m T' \arrow[dd, "\theta_m T'"] \\
 & & (5) & & \\
T' & & & & T'^2 \arrow[llll, "\mu'"]
\end{tikzcd}$$

where (1) commutes by the definition of θ_{m+n+1}, (2) by the inductive hypothesis, (3) by the definition of $\theta_m\theta_n$, (4) by the naturality of μ' and (5) because $\underline{T}'$ is a theory.

This completes the proof that $(\underline{T}, \phi)$ is free on $\underline{\Sigma}$ with respect to U. Thus we have constructed a functor,

$F : \underline{Sig}(\underline{C}) \to \underline{Th}(\underline{C})$,

left adjoint to the functor U. $\square$

Free theories on a signature, as constructed above, gives us initial algebras on a signature as follows. We have seen (page 105) that, if $\underline{T} = (T, \eta, \mu)$ is any theory on $\underline{C}$ and 'a' an object of $\underline{C}$, then

$a\mu : (aT)T \to aT$

is a $\underline{T}$-algebra and is free with respect to the forgetful functor which takes an algebra to its carrier. Notice that if $\underline{C}$ has an initial object Φ then

$\Phi\mu : (\Phi T)T \to \Phi T$

is the initial $\underline{T}$-algebra. Now suppose that $\underline{T}$ is the free theory on a signature, $\underline{\Sigma} = (\Sigma, \sigma)$, with respect to U. Then T is the carrier of the initial $\underline{\Sigma}$-algebra, the initial algebra itself being,

$\Phi T \mathfrak{Z}_1 . \Phi\mu : \Phi T\Sigma \to \Phi TT \to \Phi T.$

Likewise we can define free $\underline{\Sigma}$-algebras with respect the the forgetful functor giving the carrier of an algebra.

PRESENTATIONS OF THEORIES

Theories are essentially infinite objects but theories which we use often can be expressed in a finite form by a "presentation". We model the idea of a presentation for monadic theories as follows.

Definition

A presentation is a pair of signatures $\underline{\Gamma}$ and $\underline{\Sigma}$ together with a pair of signature morphisms (called 'derivors'):

$$\begin{tikzcd} \underline{\Gamma} \arrow[r, shift left, "\rho"] \arrow[r, shift right, "\lambda"'] & \underline{\Sigma} F U \end{tikzcd}$$

where F is the free theory functor and U the "forgetful" functor from theories to signatures.

In equational theories, this is interpreted as follows. The signature $\underline{\Sigma}$ is that of the operations (i.e. the signature of the presentation, in the usual sense). The signature $\underline{T}$ arises from the equations - each equation gives an operation with arity that of the terms in the equation. The two derivors then simply map the equations to the terms on their left-hand and right-hand sides.

We give an example. We may present the theory of groups as follows:

```
    sorts element
    opns  x : element,element -> element
          e : -> element
          inv : element -> element
    eqns  associativity   (a x b) x c = a x (b x c)
          left identity   a x e = a
          right identity  e x a = a
          right inverse   a x inv(a) = e
          left inverse    inv(a) x a = e
```

Notice that we have named the equations. The signature $\underline{T}$ for this presentation is then:

```
    sorts element
    opns  associativity : element,element,element -> element
          right_identity: element -> element
          left_identity : element -> element
          right-inverse : element -> element
          left_inverse  : element -> element
```

Presentations define theories. Suppose that,

$$\begin{tikzcd} \underline{T}' \arrow[r, shift left, "\rho"] \arrow[r, shift right, "\lambda"'] & \underline{\Sigma} FU \end{tikzcd}$$

is a presentation. The theory defined by it is given by the colimiting object of the coequaliser (in the category of theories) of the pair of morphisms,

$$\begin{tikzcd} \underline{T}F \arrow[r, shift left, "\rho^*"] \arrow[r, shift right, "\lambda^*"'] & \underline{\Sigma} F \end{tikzcd}$$

where $\rho^{\#} = \rho F.Fu$ and (F,U,t,u) is the adjunction between signatures and theories.

The existence of coequalisers in a category of theories is not a trivial result. They exist for equational theories and Lawvere [1963b] has shown that they exist for theories in a categorical form. For various forms of 'logical' theories the result may be found in [Burstall,Goguen 1980a]. The result of most interest to us at the moment is the existence of coequalisers of monadic theories. This is to be found on page (179).

There is an adjunction between signatures and presentations - each signature is trivially a presentation and each presentation contains a signature (that of the sorts and operations).

The idea of a presentation as a parallel pair of morphisms and the theory defined by a presentation as a coequaliser may be found in [Lawvere 1963b] for the case of Lawvere theories.

COMPUTATIONAL ASPECTS OF THE FREE THEORY CONSTRUCTION

The technique, which we have developed, of translating categorical constructions into programs can be applied to this free theory construction. This gives us a means of computing free theories but, more than this, we get a categorical formulation of program iteration. We shall see this at work with an example from graph theory - the transitive closure of a graph.

The construction of a free theory requires ω-colimits. We shall thus be using the encoding of ω-chains and ω-colimits and the computation of ω-colimits (see page 56). We proceed as in the other implementations of constructions in categories. First we give the required data types then an encoding of the constructive part of the proof.

Signatures in a monadic form are pairs of endofunctors and natural transformations. Morphisms of signatures are natural transformations with a certain commutativity property:

```
    type M_Signature(o,m) ==
           Functor(o,m,o,m) # Nat_transform(o,m,o,m)

    type M_Signature_Mor(o,m) == Nat_transform(o,m,o,m)
```

It is rather unwieldy to give explicit expressions for signatures in this form so we have a function to translate signatures in the familiar form of sorts and operations into such a monadic signature. This is the polynomial expression for a monadic signature described above.

```
    dec monadic_signature : Signature(Tag alpha) ->
           M_Signature(Set(Tag alpha),Set_Mor(Tag alpha))

    --- monadic_signature(Opns,mor(_,arity,_),Sorts)  <=
             let C & cat(_,_,id,_) == cat_of_sets in
       let omap ==            ! object part of functor
            (lambda S =>      ! S is a set of variables
               let indexed_set_of_terms ==
                              ! set of terms indexed on operations
                  (lambda rho =>
                     let string(ll) == arity(rho) in
                       (lambda l => string(rho::l))
                          * lists(length(ll)-1)(S) ) * Opns in
               let set_of_terms ==
                              ! either pinked variables or terms
                              ! of depth one
                 (pink*S) U total_union(indexed_set_of_terms) in
            set_of_terms ) in
       let mmap ==            ! morphism part of functor
            (lambda mor(s,f,t) =>
               let f1 == (lambda
                               pink(s) => pink(f(s))
                              | string(rho::l) =>
                                      string(rho::(f*l)) ) in
            mor(omap(s),f1,omap(t)) ) in
       let Sigma == functor(omap,mmap) in
       let sigma ==           ! the natural transformation
            nat_transform( I(C),
                          (lambda S =>
                              mor( S,
                                   (lambda x => pink(x)),
                                   omap(S))),
                          Sigma )    in
       ( Sigma, sigma )
```

Notice the overloading of the '*' operation. On lists it is defined by: f * [a, b, ...] = [f(a), f(b), ...], and similarly on sets. The recursively defined function "lists" takes a natural number, n, and a set, S, and gives the set of lists of length n of elements from S.

Given a monadic signature, a category $\underline{C}$ and an object of $\underline{C}$ we can form an ω-chain of the form of (*) by:

```
    dec mkchain : Cat(o,m) ->
                    (M_Signature(o,m) -> (o -> w_Chain(o,m)))
    --- mkchain(C) <=
          (lambda (Sigma,sigma) => (lambda a =>
              ((lambda n =>
                    (Sigma to_power n) ofo a),
               (lambda n =>
                    (Sigma to_power n) ofm (sigma of a)))))
```

where "to-power" is an infix operation which raises an endofunctor to a power by successive composition with itself.

In fact this function is the object part of a functor from C̲ to ω-chains on C̲ (which themselves form a category).

```
    dec chain_functor : Cat(o,m) -> ( M_Signature(o,m) ->
                  Functor(o, m, w_Chain(o,m), w_Chain_Mor(o,m)) )
    --- chain_functor(C & cat(s,t,_,_)) <=
          (lambda f_sig & (Sigma,sigma) =>
             functor(mkchain(C)(f_sig),
                     (lambda f => ( mkchain(C)(f_sig)(s(f)),
                                    (lambda n =>
                                       (Sigma to_power n) ofm f),
                                    mkchain(C)(f_sig)(t(f)) )) ))
```

The colimit of these ω-chains, which are generated by monadic signatures, can be described as a functor:

```
    dec w_colimit_functor : Colimit_Cat(o,m) ->
                              (M_Signature(o,m) ->
                       Functor(o,m,Cone(o,m),Cone_Mor(o,m)) )

    --- w_colimit_functor(wCC & colimit_cat(C,_)) <=
           (lambda f_sig =>
              chain_functor(C)(f_sig)
                   w_diagram_functor
                       colimit_functor(wCC) )
```

The functor 'colimit_functor' takes diagrams to colimiting cones on the diagrams (page 35).

Now we are in a position to compute the free theory on a signature. The endofunctor of the free theory is simply the colimiting object (treated as a functor - see page 35) of the ω-chain and can be given rather neatly by a composition of functors:

```
    dec term_functor : Colimit_Cat(o,m) ->
           (M_Signature(o,m) -> Functor(o,m,o,m))
    --- term_functor(wCC) <=
           (lambda f_sig =>
                w_colimit_functor(wCC)(f_sig).apex_functor)
```

The unit of the theory, as constructed in the proof of the existence of free theories, is given as:

```
    dec eta : Colimit_Cat(o,m) ->
         ( M_Signature(o,m) -> Nat_transform(o,m,o,m) )
    --- eta(wCC & colimit_cat(C,_)) <=
            (lambda f_sig =>
                let T == term_functor(wCC)(f_sig) in
       nat_transform(I(C),
                      (lambda a =>
                        sides(w_colimit_functor(wCC)(f_sig) ofo a)
                             (just(number 0)) ),
                      T ) )
```

Likewise for the composition in the theory:

```
    dec mu : Colimit_Cat(o,m) ->
         ( M_Signature(o,m) -> Nat_transform(o,m,o,m) )
    --- mu(wCC & colimit_cat(C,w_colim)) <=
            (lambda f_sig & (S,s) =>
         let T == term_functor(wCC)(f_sig) in
         let amu ==
            (lambda a =>
                      ! the colimit cone on the chain on 'a' :
         let c_cone == w_colimit_functor(wCC)(f_sig) ofo a in
         let h ==
            (lambda m =>
               let newchain ==
                     w_diagram(chain_functor(C)(f_sig)
                                  ofo ((S to_power m) ofo a)) in
               let _,univ == w_colim(newchain) in
               let c_cone_mor ==
                    univ( cone(C)( newchain,
                                   (lambda just(number n) =>
                                     sides(c_cone)
                                         (just(number(m+n)))),
                                   a )) in
            apex_morphism(c_cone_mor) )    in
         let T_of_oldchain ==
           ((lambda n => T ofo ((S to_power n) ofo a)),
            (lambda n => T ofm ((S to_power n) ofm (s of a)))) in
         let pcone ==
            cone(C)( w_diagram(T_of_oldchain),
                      (lambda just(number n) => h(n)),
                      apex(c_cone)) in
         let _,universal == w_colim(w_diagram(T_of_oldchain)) in
                      ! here we assume the w_cocontinuity of T
         let c_cone_mor1 == universal(pcone) in
                           apex_morphism(c_cone_mor1) ) in
        nat_transform( T.T, (lambda a => amu(a)), T) )
```

Then the free theory on a signature is computed by the following function.

```
dec free_theory : Colimit_Cat(o,m) ->
        ( M_Signature(o,m) -> Monad(o,m) )
--- free_theory(wCC) <=
         (lambda f_sig =>
             monad( term_functor(wCC)(f_sig),
                    eta(wCC)(f_sig),
                    mu(wCC)(f_sig)) )
```

We are now quite adept at translating existence proofs in category theory into code for machines. What can we use the above program for?

Well, by choosing the category C to be the category of sets, we may construct free theories on Set. In general, these will be rather uninteresting since the set of terms will be infinite - so cannot be represented. However all is not lost since the morphism part of the endofunctor T is non-trivial - it is a substitution function. Indeed the following example shows that we can choose the signature so that the function models the 'mapcar' of LISP. We thus, perhaps, have the most contorted implementation of 'mapcar' which has yet appeared.

Suppose we choose a signature with one binary operation, "rho",

```
dec sig : Signature(Tag(List Char))
--- sig <= ( {just("rho")},          ! operations
             mor( {just("rho")},
                  (lambda _ =>
                     string [just("s"),just("s"),just("s")]),
                  bigset),
             {just("s")} )           ! sorts
```

Terms on this signature look like lists and so the 'mapcar' operation can be modelled by:

```
dec mapcar: Set_Mor(Tag (List Char)) ->
              Set_Mor(Tag (List Char))

--- mapcar(m)   <=
        term_functor(infinite_w_colimit_cat_of_sets)
                        (functorial_signature(sig)) ofm m
```

We now look at a more serious application of the construction. The formalism of monadic theories includes many things which one would not naturally associate with the idea of a theory. We return to the graphical problem left unsolved in chapter five and show how the transitive closure of a graph may be described by a free theory on a suitably chosen signature on graphs. In doing this we will be needing ω-colimits in the category of graphs. Moreover, we shall see that a complicated iteration in a program can be subsumed in the computation of ω-colimits. I first wrote the program in the usual iterative way, only later realising that the work could be cast into a categorical framework and thence that the iterative details had already been encoded for ω-colimits.

The idea is this: Take a graph. Make a new graph by adding new edges one for each composable pair in the old graph (this is not then transitively closed - it is, however, one step towards the closure). This operation actually extends to an endofunctor on the category of graphs. Moreover there is an obvious inclusion of the old graph into the new. This pair, of a functor and a natural transformation, is a signature in a monadic form on the category of graphs. The free theory on the signature is the transitive closure, in the following sense. The endofunctor of the theory takes a graph to its transitive closure. The unit of the theory is the obvious inclusion of a graph into its transitive closure. The composition is saying that transitive closure is a closure - it is idempotent.

This observation now gives us a means of computing the transitive closure of a graph. We first note that there is a comma category associated with this operation of transitive closure. The edges in a transitively closed graph have a partially defined "composition" upon them taking any composable pair of edges into the composition of the pair. The "composition" is associative and so forms a partial semigroup:

```
    data Semigroup(alpha) ==
            semigroup( Set(alpha),  ! the carrier
                      ((alpha # alpha) -> truval),
                                    ! the definedness predicate
                      ((alpha # alpha) -> alpha) )
                                    ! the partial composition
```

Morphisms of these semigroups are functions on the carrier which respect the definedness of composition as well as the composition itself:

```
    data Semigroup_Mor(alpha) ==
         semigroup_mor( Semigroup(alpha),
                        Set_Mor(alpha),
                        Semigroup(alpha) )
```

The category of these partial semigroups is given by the now-familiar construction.

Graphs with such a composition operation upon their edges are given as a comma category as follows. First define a functor, called 'pair_semigroup' from sets to partial semigroups which takes a set onto its crossproduct treated as a partial semigroup. The composition of pairs of elements is:

    compose((a,b), (c,d)) = (a,d) if b=c else undefined.

Notice how this reflects the condition for the composition of edges in a graph. We form the comma category (cat_of_partial_semigroups,pair_semigroup) and call it the category of composition graphs. Thus the functor, pair_semigroup, is given by:

```
                  !  object part
dec pair_semigroup : Set(Tag alpha) -> Semigroup(Tag alpha)
--- pair_semigroup(A) <=
        semigroup( cross_product ofo A,
                            ! carrier is cross_product
                   (lambda pair(a,b),pair(c,d) => b=c),
                            ! composition def
                            ! if target = source
                   (lambda pair(a,_),pair(_,d) => pair(a,d)))
                            ! composition

                 ! morphism part
dec pair_semigroup : Set_Mor(Tag alpha) ->
                        Semigroup_Mor(Tag alpha)
--- pair_semigroup(m) <=
        let C == cat_of_sets in
   semigroup_mor( pair_semigroup(source(C)(m)),
                  cross_product ofm m,
                  pair_semigroup(target(C)(m)) )

dec pair_semigroup :
       Functor(Set(Tag alpha),Set_Mor(Tag alpha),
               Semigroup(Tag alpha),Semigroup_Mor(Tag alpha))
--- pair_semigroup <= functor(pair_semigroup,pair_semigroup)
```

The comma category of composition graphs is then:

```
type CompositionGraph(alpha) ==
        Semigroup(alpha)#Semigroup_Mor(alpha)#Set(alpha)

type CompositionGraph_Mor(alpha) ==
        Right_Comma_Mor( Semigroup(alpha),
                         Semigroup_Mor(alpha),
                         Set(alpha),
                         Set_Mor(alpha))
```

```
    dec cat_of_composition_graphs :
            Cat( CompositionGraph(Tag alpha),
                 CompositionGraph_Mor(Tag alpha))
    --- cat_of_composition_graphs <=
            right_comma_cat( cat_of_partial_semigroups,
                             pair_semigroup,
                             cat_of_sets)
```

There is a forgetful functor from composition-graphs to graphs (giving the underlying graph of a composition graph):

```
    dec U : CompositionGraph(Tag alpha) -> Graph(Tag alpha)
```

Now the idea is that, in the construction of the free theory which gives the transitive closure of a graph, we successively extend the domain of definition of the partially defined composition on the edges by accumulating new edges. Thus we are in fact working in the category of composition graphs. The step of adding new edges to such a graph, one edge for each composable pair of edges, is given by:

```
    dec new_paths : CompositionGraph(Tag alpha) ->
                       CompositionGraph(Tag alpha)

    --- new_paths(g & ( semigroup(E,def,comp),
                        semigroup_mor(_,m,_),
                        N ) )   <=
        let composable_pairs == filter cross_product ofo E  by
             (lambda pair(e1,e2) =>
                   range(U(g))(e1)= dom(U(g))(e2)) in
        let new_edges == filter composable_pairs by
             (lambda pair(e1,e2) => not(def(e1,e2)) ) in
        let new_def == (lambda e1,e2 => true
                                   if def(e1,e2) else
                             pair(e1,e2) is_in new_edges) in
        let new_comp == (lambda e1,e2 => comp(e1,e2)
                                   if def(e1,e2) else
                                         pair(e1,e2) ) in
        let new_map == (lambda e => (m of e)
                                   if e is_in E else
                           (pair(dom(U(g))(e1),range(U(g))(e2))
                                    where pair(e1,e2) == e) ) in
        let E1 == E U new_edges in
        let new_semigp == semigroup(E1, new_def, new_comp) in
        let new_graph ==
            (new_semigp,
             semigroup_mor( new_semigp,
                            mor( E1,
                                 new_map,
                                 cross_product ofo N),
                            pair_semigroup ofo N ),
             N ) in
      new_graph
```

Here the set "filter S by P" is {s in S | P(s)}. The morphism part of the functor is:

```
    dec morphism_part : CompositionGraph_Mor(Tag alpha) ->
                              CompositionGraph_Mor(Tag alpha)
    --- morphism_part(comma_mor( cg1,
                                 ( semigroup_mor(sg1,m,sg2),
                                   m_node),
                                 cg2)) <=
       let new_cg1 & (new_sg1,_,_) == new_paths(cg1) in
       let new_cg2 & (new_sg2,_,_) == new_paths(cg2) in
       let new_mor ==
            semigroup_mor( new_sg1,
                           mor( edges(new_cg1),
                                (lambda e => m of e
                                   if e is_in carrier(sg1) else
                                     pair(m of e1, m of e2)
                                       where pair(e1,e2) == e),
                                edges(new_cg2) ),
                           new_sg2 ) in
       comma_mor(new_paths(cg1),(new_mor,m_node),new_paths(cg2))
```

The functor is then:

```
    dec new_paths :
            Functor( CompositionGraph(Tag alpha),
                        CompositionGraph_Mor(Tag alpha),
                     CompositionGraph(Tag alpha),
                        CompositionGraph_Mor(Tag alpha))
    --- new_paths <= functor(new_paths,morphism_part)
```

This is the endofunctor of a signature whose "unit" is:

```
    dec inject :
            Nat_transform( CompositionGraph(Tag alpha),
                              CompositionGraph_Mor(Tag alpha),
                           CompositionGraph(Tag alpha),
                              CompositionGraph_Mor(Tag alpha))
    --- inject <=
       nat_transform(
            I(cat_of_composition_graphs),
            (lambda cg & (E,_,N) =>
               let new_cg & (E1,_,_) == new_paths(cg) in
                    comma_mor( cg,
                               ( semigroup_mor(E,
                                               mor( carrier(E),
                                                    ident,
                                                    carrier(E1)),
                                               E1),
                                 identity(cat_of_sets)(N) ),
                               new_cg) ),
            new_paths )
```

The signature for this transitive closure operation is then:

```
    dec new_paths_sig :
            F_Signature( CompositionGraph(Tag alpha),
                         CompositionGraph_Mor(Tag alpha))
    --- new_paths_sig <= ( new_paths, inject )
```

Notice that any graph can be considered trivially as a composition graph with composition everywhere undefined. This is the start of the iteration of the free theory construction. We need a category with ω-colimits. Fortunately, we are here dealing with the so-called "finite_w_colimit_categories" (see page 56) in this case the category of composition graphs so treated. Finally we can give the transitive closure operation on a graph as follows:

```
    dec transitive_closure :
            Graph(Tag alpha) -> CompositionGraph(Tag alpha)

    --- transitive_closure(g) <=
       let SG ==
            finite_w_colimit_cat(cat_of_partial_semigroups) in
       let S == finite_w_colimit_cat(cat_of_sets) in
       let CG ==
            colimit_comma_cat( SG,
                               pair_semigroup,
                               S ) in
        term_functor(CG)(new_paths_sig) ofo
                             trivial_composition_graph(g)
```

Notice that if the graph is not acyclic then the ω-chain will fail to become a constant chain and hence the routine "fixed point" (page 57) will not terminate.

Notice how the main iteration implicit in the transitive closure has been "hidden" in the ω-colimit routine and thence that the above routine "transitive_closure" is constructed by the successive application of functionals (indeed it could be made totally variable-free).

This completes our discussion of the transitive closure of a graph.

Notes

The connection between free theories and ω-colimits appears in various guises as mentioned above. The notion of signature, presentation and the construction here arose in discussion with Prof. Rod Burstall. He provided a complete proof using a restricted form of the approximation lemmas. The proof here is a 'polished' version of that in [Burstall and Rydeheard 1979].

CHAPTER NINE

FREE ALGEBRAS AND DATA THEORIES

The equational theories of universal algebra have proved a powerful stimulus in the study of program specification. However, they are, in fact, inadequate to describe the properties even of simple programs. The problem is that some properties arise from induction rules rather than the usual rules of inference available in equational specification - those of equational deduction. It is not so much that we cannot handle these induction rules in universal algebra. In some sense, we can - they arise from initial or free algebras. However, we want specifications in which certain parts are restricted to an initial or free interpretation whilst other parts are to have arbitrary interpretation. Specifications of this form can be formalised under the notion of a "data theory" due to Burstall and Goguen [1980a] (see also [Reichel 1980]). For a full explanation with examples consult [Burstall,Goguen 1980b].

These "partially interpreted" theories can be formalised using the so-called data constraints. Each part of a specification which is required to be interpreted initially or freely (in some sense) gives a corresponding data constraint. In a rather general logical setting, it is known that these data constraints behave like equations and hence that data theories can be defined in the same way that equational theories are defined by presentations.

Our aim here is to show we can handle data constraints within monadic theories. We already know that monadic theories suffice to describe equational theories. This, then, is an extension of this result and is part of our program of showing that monadic theories are not only sufficient for much of program specification but they handle many cases uniformly. Essentially we are showing that Axiom 4 (page 103) is satisfied by monadic theories.

The construction is again by colimits and is in two parts. We first show that an important result concerning theories and their algebras holds for monadic theories. This is interesting not only for the explication of data theories but gives further insight into the computational aspects of specifications. The other part is a novel method of constructing theories from classes of algebras such that the theory contains all the equations which are "true" in all the algebras in the class. This gives us a somewhat different approach to the data constraints of data theories.

FREE FUNCTORS ON ALGEBRAS

There is a very important result concerning certain types of theories and their algebras. It may be stated, somewhat loosely, as:

For any theory morphism $\alpha : \underline{S} \to \underline{T}$, there is a "forgetful" functor $U : \underline{\mathrm{Alg}}(\underline{T}) \to \underline{\mathrm{Alg}}(\underline{S})$ (notice the reversal of arrows) obtained from α. The result then says that U has a left adjoint, a free functor $F : \underline{\mathrm{Alg}}(\underline{S}) \to \underline{\mathrm{Alg}}(\underline{T})$.

For example, let $\underline{S}$ be the theory of semigroups and $\underline{T}$ that of groups, with α the usual 'inclusion' morphism. Then U is the functor which takes any group and simply forgets that the identity and inverses are present - that is, every group is a semigroup. The free functor closes a given semigroup to a group.

The importance of the existence of this left adjoint can be seen from its variety of manifestations. It is the crux of the functorial semantics as introduced by Lawvere [1963b]. As special cases it gives the existence of initial and free algebras of a theory. Moreover, as mentioned before, it provides an understanding of the so-called "data theories" which allow us to restrict the interpretation of parts of specifications to initial or free models [Burstall,Goguen 1980b]. A result of the same shape but dealing with functor composition instead of theories and algebras was introduced by Kan [1958]. Later we will make use of Kan's result to construct free <u>signatures</u> from other signatures - a generalisation of a construction in order-sorted specifications.

For what types of theories does the result above hold? For equational theories and, more generally, for theories in a Lawvere form the result is already known [Lawvere 1963b]. The work of this chapter shows not only that the result holds for certain monadic theories, but also that the free functor left adjoint to the "forgetful" functor on algebras is constructed by a simple colimit - a coequaliser - and is thus ought to be susceptible to the techniques we have acquired of programming category theory constructions. The construction itself resembles that of Linton [1969] who uses a similar coequaliser to construct coproducts of algebras.

The construction below and further constructions which we are to give require certain preservation properties to hold for the endofunctors of monadic theories. In the case of the free theory construction it was mentioned that the preservation condition (preservation of ω-colimits) could be verified for finitary equational theories on the category of sets. Endofunctors of such theories in general preserve only epis, monos and, in terms of colimits, only filtered colimits (see [Mac Lane 1971] for details). Thus ω-colimits are preserved but not pushouts, coequalisers, coproducts or even initial objects. This is why there are many attempts to make constructions of this sort independent of the preservation properties of the endofunctors (e.g. [Adamek, Koubek 1980]). A glance at the constructions below show that we are unable to do this and, moreover, that we have not established that the preservation properties hold even for the case of finitary equational theories on the category of sets. This is an omission. It seems that the conditions do hold for several examples but a general proof is lacking. Notice that the conditions required are not the preservation of colimits of all diagrams of a fixed shape but something weaker.

We now turn to the construction, first stating what we need as a theorem:

Theorem

Let $\alpha : \underline{S} \rightarrow \underline{T}$ be a theory morphism between monadic theories $\underline{S} = (S, \eta_S, \mu_S)$ and $\underline{T} = (T, \eta_T, \mu_T)$, both on the same base category $\underline{C}$ which has coequalisers. Moreover we will assume that the endofunctors T and T^2 preserve coequalisers of pairs of morphisms of the form (*) below.

The forgetful functor U : Alg(T) -> Alg(S) is given by,

(ϵ : aT -> a)U = (aα.ϵ : aS -> aT -> a),

with a morphism part which is essentially the identity on morphisms.

Then there is a left adjoint to U, a free functor

F : Alg(S) -> Alg(T)

Proof

Given any S-algebra, ϵ : aS -> a, we need to construct a T-algebra.

Consider the pair of morphisms between aST and aT:

$$\begin{tikzcd} aST \arrow[r, shift left, "\epsilon T"] \arrow[r, shift right, "a\alpha T . a\mu_T"'] & aT \end{tikzcd} \qquad - (*)$$

Let f : aT -> b be their coequaliser. We can make b into the carrier of a T-algebra by noting that the following diagram commutes:

$$\begin{tikzcd}
aST^2 \arrow[r, "a\alpha T^2"] \arrow[d, "\epsilon T^2"'] \arrow[dr, "aS\mu_T"] & aT^3 \arrow[r, "a\mu_T T"] \arrow[dr, "aT\mu_T"] \arrow[d, phantom, "(2)"] & aT^2 \arrow[r, "a\mu_T"] \arrow[dr, phantom, "(4)"] & aT \arrow[dd, "f"] \\
aT^2 \arrow[d, "a\mu_T"'] \arrow[r, phantom, "(3)"] & aST \arrow[r, "a\alpha T"'] \arrow[dl, "\epsilon T"] & aT^2 \arrow[ur, "a\mu_T"'] \arrow[dr, phantom, "(1)"] & \\
aT \arrow[rrr, "f"'] & & & b
\end{tikzcd}$$

where (1) commutes by the definition of f, (2) and (3) by the naturality of μ_T and (4) by a theory law for T.

Thus aμ_T.f is a cone on diagram (1) with T applied to it. As T preserves coequalisers of the form of (*) above, we have:

There is a unique ϵ' : bT -> b, such that

$$\begin{tikzcd} aT^2 \arrow[r, "fT"] \arrow[d, "a\mu_T"'] & bT \arrow[d, dashed, "\epsilon'"] \\ aT \arrow[r, "f"'] & b \end{tikzcd}$$

commutes.

We need to show that this ϵ' : bT -> b is an algebra. Before doing so, we state an approximation lemma arising from the coequaliser.

The 1-approximation lemma looks like:

If r,s : b -> c and

$$\begin{tikzcd} aT \arrow[r, "f"] \arrow[d, "f"'] & b \arrow[d, "r"] \\ b \arrow[r, "s"'] & c \end{tikzcd}$$

commutes, then r = s.

Further approximation lemmas of this form arise from the fact that fT and fT^2 are both coequalisers because T and T^2 preserves coequalisers of the form (*) above.

Now to show that ϵ' : bT -> b is an algebra, we need that both the diagrams below commute.

$$\begin{tikzcd} b \arrow[r, "b\eta_T"] \arrow[dr, "="'] & bT \arrow[d, "\epsilon'"] \\ & b \end{tikzcd} \qquad \begin{tikzcd} bT^2 \arrow[r, "\epsilon' T"] \arrow[d, "b\mu_T"'] & bT \arrow[d, "\epsilon'"] \\ bT \arrow[r, "\epsilon'"'] & b \end{tikzcd}$$

For the first, by the 1-approximation lemma, it suffices to show that

$$\begin{tikzcd} aT \arrow[rrr, "f"] \arrow[dr, "aT\eta_T"'] \arrow[ddd, "f"'] & & & b \arrow[ddd, "="] \\ & aT^2 \arrow[dr, "a\mu_T"] \arrow[dd, "fT"'] & (1) & \\ (2) & & aT \arrow[dr, "f"] & \\ b \arrow[r, "b\eta_T"'] & bT \arrow[rr, "\epsilon'"'] & (3) & b \end{tikzcd}$$

commutes.

But (1) commutes by a theory law for $\underline{T}$, (2) by the naturality of η_T and (3) by the definition of ϵ'. Hence, the square commutes as required.

For the second algebra law, we again use an approximation lemma. It thus suffices to show that

$$\begin{tikzcd}
aT^3 \arrow[rrrr, "fT^2"] \arrow[drr, "a\mu_T T"] \arrow[ddr, "a\mu_T T"'] \arrow[dddd, "fT^2"'] & & & & bT^2 \arrow[dd, "\epsilon' T"] \\
& & aT^2 \arrow[drr, "fT"] \arrow[dr, "a\mu_T"] & (2) & \\
& aT^2 \arrow[dr, "a\mu_T"] \arrow[ddr, "fT"'] & (1) & aT \arrow[ddr, "f"] \arrow[dr, phantom, "(3)"] & bT \arrow[dd, "\epsilon'"] \\
& (5) & aT \arrow[drr, "f"] \arrow[d, phantom, "(4)"] & & \\
bT^2 \arrow[rr, "b\mu_T"'] & & bT \arrow[rr, "\epsilon'"'] & & b
\end{tikzcd}$$

commutes. But (1) is a theory law for $\underline{T}$, (2), (3), (4) are the definition of ϵ' and (5) is the naturality of μ_T. Thus the square commutes as required.

We have shown that $\epsilon' : bT \to b$ is an algebra. We now show that the passage from the algebra ϵ to the algebra ϵ' is functorial - that is, we can define an action on morphisms.

Let $g : a \to a'$ be an algebra morphism from $\epsilon : aS \to a$ to an algebra $\lambda : a'S \to a'$. Thus, by definition:

$$\begin{tikzcd}
aS \arrow[r, "gS"] \arrow[d, "\epsilon"'] & a'S \arrow[d, "\lambda"] \\
a \arrow[r, "g"'] & a'
\end{tikzcd}$$

commutes.

If $\epsilon' : bT \to b$ is the algebra of $\underline{T}$ obtained by the foregoing construction from the algebra $\epsilon : aS \to a$, and $\lambda' : b'T \to b'$ is the algebra obtained from $\lambda : a'S \to a'$, then we can define an algebra morphism $h : b \to b'$ from ϵ' to λ' by:

$$\begin{tikzcd}
aST \arrow[r, shift left, "a\alpha T.\, a\mu_T"] \arrow[r, shift right, "\epsilon T"'] \arrow[d, "gST"'] \arrow[dr, phantom, "(1)"] & aT \arrow[r, "f"] \arrow[d, "gT"] \arrow[dr, phantom, "(2)"] & b \arrow[d, dashed, "h"] \\
a'ST \arrow[r, shift left, "\lambda T"] \arrow[r, shift right, "a'\alpha T.\, a'\mu_T"'] & a'T \arrow[r, "f'"'] & b'
\end{tikzcd}$$

Here, (1) commutes for the the upper and lower morphisms separately, f' is the coequaliser of the morphisms a'αT.a'μ_T and λT and h is the unique morphism which makes (2) commute.

To show that h really is an algebra morphism, i.e that the following square commutes,

$$\begin{tikzcd}
bT \arrow[r, "hT"] \arrow[d, "\epsilon'"'] & b'T \arrow[d, "\lambda'"] \\
b \arrow[r, "h"'] & b'
\end{tikzcd}$$

we use an approximation lemma. It thus suffices to show that the following commutes.

$$\begin{tikzcd}
aT^2 \arrow[rrr, "fT"] \arrow[drr, "gT^2"] \arrow[dr, "a\mu_T"'] \arrow[dddd, "fT"'] & & & bT \arrow[dd, "hT"] \\
& & a'T^2 \arrow[dr, "f'T"] \arrow[dd, "a'\mu_T"] & \\
& aT \arrow[dr, "gT"] \arrow[ddr, "f"'] & & b'T \arrow[dd, "\lambda'"] \\
& & a'T \arrow[dr, "f'"] & \\
bT \arrow[rr, "\epsilon'"'] & & b \arrow[r, "h"'] & b'
\end{tikzcd}$$

But (1) commutes by the naturality of μ_T, (2) by the definition of ϵ' and (3) by that of λ'. (4) and (5) are both the definition of

h. Thus h is an algebra morphism as required and we may define the functor,

F : <u>Alg</u>(<u>S</u>) -> <u>Alg</u>(<u>T</u>)

by the action on objects and morphisms defined above.

Now we turn to the question of freeness. First, there is an algebra morphism from ϵ to $(\epsilon)FU$:

$a\eta_T . f : a \to b.$

This is indeed an algebra morphism for the following diagram commutes.

$$\begin{tikzcd}
aS \arrow[rr, "a\eta_T S"] \arrow[dr, "a\alpha"] \arrow[ddr, "aS\eta_T"'] \arrow[ddd, "\epsilon"'] & (3) & aTS \arrow[rr, "fS"] \arrow[dr, "aT\alpha"] & (4) & bS \arrow[d, "b\alpha"] \\
& aT \arrow[rr, "a\eta_T T"] \arrow[dr, "aT\eta_T"] & (1) & aT^2 \arrow[r, "fT"] \arrow[d, "a\mu_T"] \arrow[dl, phantom, "(5)"] & bT \arrow[dd, "\epsilon'"] \\
(2) & aST \arrow[r, "a\alpha T"] \arrow[d, "\epsilon T"] & aT^2 \arrow[r, "a\mu_T"] & aT \arrow[dr, "f"] & (6) \\
a \arrow[r, "a\eta_T"'] & aT \arrow[rrr, "f"'] & (7) & & b
\end{tikzcd}$$

(1) and (2) are the naturality of η_T, (3) and (4) that of α, (5) is a theory law for <u>T</u>, (6) is the definition of ϵ' and (7) is the definition of f. Thus,

$$\begin{tikzcd}
aS \arrow[r, "(a\eta_T f)S"] \arrow[dd, "\epsilon"'] & bS \arrow[d, "b\alpha"] \\
& bT \arrow[d, "\epsilon'"] \\
a \arrow[r, "a\eta_T f"'] & b
\end{tikzcd}$$

commutes as required. So $a\eta_T . f$ is an algebra morphism.

We can now show the freeness property of the algebra ϵ'.

We need to show that for any algebra, $\kappa : cT \to c$, and for any algebra morphism, $g : \epsilon \to KU$, there is a unique $g^{\#} : \epsilon' \to K$ such that:

$$\begin{tikzcd}
\epsilon \arrow[r, "a\eta_T f"] \arrow[dr, "g"'] & \epsilon' U \arrow[d, dashed, "g^{\#} U"] \\
& K U
\end{tikzcd}$$

commutes. We construct $g^{\#}$ from g by noting that

$$\begin{tikzcd}
aST \arrow[r, "\epsilon T"] \arrow[d, "gST"'] & aT \arrow[dd, "gT"] \\
cST \arrow[d, "c\alpha T"'] & \\
cT^2 \arrow[r, "KT"'] & cT
\end{tikzcd}
\qquad
\begin{tikzcd}
aST \arrow[r, "a\alpha T"] \arrow[dd, "gST"'] \arrow[ddr, phantom, "(1)"] & aT^2 \arrow[rr, "a\mu_T"] \arrow[dd, "gT^2"'] \arrow[drr, phantom, "(2)"] & & aT \arrow[d, "gT"] \\
& & & cT \arrow[d, "K"] \\
cST \arrow[r, "c\alpha T"'] & cT^2 \arrow[r, "KT"'] \arrow[urr, "c\mu_T"] & cT \arrow[r, "K"'] \arrow[ur, phantom, "(3)"] & c
\end{tikzcd}$$

both commute - the first because g is an algebra morphism and the second because (1) commutes by the naturality of α, (2) by that of μ_T and (3) by an algebra law of K. Thus $gT.K$ is a cone on the parallel pair of morphisms ϵT and $a\alpha T.a\mu_T$ and so there is a unique morphism $g^{\#} : b \to c$ such that

$$\begin{tikzcd}
aT \arrow[r, "f"] \arrow[d, "gT"'] & b \arrow[d, dashed, "g^{*}"] \\
cT \arrow[r, "K"'] & c
\end{tikzcd}$$

commutes.

This $g^{\#}$ makes the freeness triangle commute and is unique in this respect, as can easily be verified. This completes the proof of the existence of a left adjoint to U, the forgetful functor on algebras induced by a theory morphism, α. $\square$

Notice that, in the above proof, there was no necessity that $\underline{S}$ should be a theory (nor that ϵ : aS -> a should be an algebra of a theory). It would have sufficed that these were signatures and signature algebras (or even a functor, S, and a morphism, aS -> a).

What does this construction say computationally? In one sense it says that equational (and, more generally, monadic) enrichments are, from a computational viewpoint, trivial since they arise from a coequaliser in a category (usually that of sets). However, the experience of chapter five might make us think otherwise. Looking back at the coequaliser in question (*), we see that, whilst the carrier of the algebra to be constructed may well be finite, both 'aST' and 'aT' are, in general, infinite. Thus the coequalising is not open to direct computation. What may possibly be done is an iterative computation of the colimit using the fact that the theories have associated signatures and then using the encoding of the free theory construction above.

INDUCTION RULES AND MONADIC THEORIES

We now present another result concerning monadic theories. This result may be thought of as mediating between the approach to specification (and to data types) whereby these objects are viewed as certain algebras and the approach which we have taken whereby these objects denote theories.

A special case of the following construction is the addition of induction rules to equational theories (some work in this direction has been done by [Nourani 1980]). More generally we consider a collection of algebras of a theory and construct a new theory which has all the equations (usually an infinite collection) which are true in all the algebras in the collection. By choosing only the initial algebra we get the "induce" construction of a previous version of CLEAR [Burstall,Goguen 1977]. By choosing the collection of algebras to be those which are free in a certain sense we get an interpretation of the data constraints in monadic theories as will be seen in the next section.

The following theorem explains what we are to do.

Theorem

Let $\underline{C}$ be a category that is co-wellpowered and cocomplete. If $\underline{T} = (T, \eta, \mu)$ is a theory on $\underline{C}$ (with T preserving epis and co-intersections of the form below) and S a class of $\underline{T}$-algebras then there is a theory $\underline{T}' = (T', \eta', \mu')$ on $\underline{C}$ and a theory morphism q: $\underline{T} \to \underline{T}'$ which is an epi, such that, for all algebras $\epsilon : aT \to a$ in S, and for all objects x in $\underline{C}$ and morphisms $f : x \to a$, there is a $g : xT' \to a$ such that

$$\begin{tikzcd}
xT \arrow[r, "xq"] \arrow[d, "fT"'] & xT' \arrow[d, dashed, "g"] \\
aT \arrow[r, "\epsilon"'] & a
\end{tikzcd}$$

commutes. Moreover, $\underline{T}'$ is universal with this property: for any other theory $\underline{T}''$ on $\underline{C}$ and epi $q' : \underline{T} \to \underline{T}''$ with the same property, there is a unique theory morphism, $\rho : \underline{T}'' \to \underline{T}'$ (notice the direction of the arrow), such that

$$\begin{tikzcd}
 & \underline{T} \arrow[dl, "q'"'] \arrow[dr, "q"] & \\
\underline{T}'' \arrow[rr, "\rho"'] & & \underline{T}'
\end{tikzcd}$$

commutes.

This can be thought of as saying that the epi xq "agrees with" all algebras in S, that is, xq "identifies" only things "identified" by all algebras in S. The class of algebras may be a set, it may be something "larger". This will cause no problem (greater than that which occurs with categories).

Proof

We start our construction of such a $\underline{T}'$ with a definition in which we intend to capture the idea of an epi "agreeing with" algebras.

Definition

If $\underline{T}$ is a theory on a category $\underline{C}$ and S is a class of $\underline{T}$-algebras an epi e : xT -> b is an <u>S-epi on x</u> iff for each algebra ϵ : aT -> a in S and for each morphism f : x -> a there is a g : b -> a such that

$$\begin{tikzcd} xT \arrow[r, "e"] \arrow[d, "fT"'] & b \arrow[d, dashed, "g"] \\ aT \arrow[r, "\epsilon"'] & a \end{tikzcd}$$

commutes.

These S-epis have two important properties. In a sense to be made explicit, they are preserved under taking pushouts. Moreover they are closed under the operation of co-intersection. We need both these properties which we now state formally using the notation of the definition above.

Proposition

If e : xT -> b is an S-epi on x and

$$\begin{tikzcd} xT \arrow[r, "e"] \arrow[d, "hT"'] & b \arrow[d, dashed, "v"] \\ yT \arrow[r, dashed, "e'"'] & p \end{tikzcd}$$

is a pushout diagram, then e' : yT -> p is an S-epi on y.

Informally - "if h is a morphism of $\underline{C}$, then pushout along hT preserves S-epis".

Proof

Pushout preserves epis, hence e' is an epi.

Now, if ϵ : aT -> a is any algebra in S and f : y -> a, then there is a g : b -> a such that the following commutes:

$$\begin{tikzcd}
xT \arrow[rr, "e"] \arrow[d, "hT"'] & & b \arrow[dl, "v"'] \arrow[dd, "g"] \\
yT \arrow[r, "e'"'] \arrow[d, "fT"'] & p & \\
aT \arrow[rr, "\epsilon"'] & & a
\end{tikzcd}$$

where (1) is the pushout defining e' and g exists to make (2) commute because e is an S-epi on x.

Hence by the colimiting property of the pushout, there is a unique k : p -> a such that

$$\begin{tikzcd}
yT \arrow[r, "e'"] \arrow[d, "fT"'] & p \arrow[d, dashed, "k"] \\
aT \arrow[r, "\epsilon"'] & a
\end{tikzcd}$$

commutes, i.e. e' is an S-epi on y.

We now introduce the notion of the co-intersection of a family of epis.

Definition

If a is an object of $\underline{C}$ and e_i : a -> b_i is a family of epis then the morphism e : a -> b is the co-intersection of the e_i iff:

1. For each i, there is a morphism, e'_i : b_i -> b such that

$e_i e'_i = e$.

2. If $g : a \to c$ and $g_i : b_i \to c$, for each i, such that $e_i g_i = g$, then there is a unique morphism $f : b \to c$ such that $ef = g$.

It should be recognised that co-intersection is nothing more than a colimit.

Now consider the family $e_i : xT \to b_i$ of S-epis on 'x'. Then:

1. It is a non-empty family as $i_{xT} : xT \to xT$ is an S-epi.

2. Though the family may <u>not</u> be a set, the co-intersection still exists as $\underline{C}$ is co-wellpowered and cocomplete. This is a general result, see [Herrlich,Strecker 1973].

<u>Proposition</u>

If $e : xT \to b$ is the co-intersection of a family $e_i : xT \to b_i$ $i \in I$ of S-epis then e is an S-epi.

<u>Proof</u>

Generally, the co-intersection of epis is an epi, hence e is an epi.

Let the co-intersection of the family be $e : xT \to b$ with morphisms in the colimiting cone $e'_i : b_i \to b$ such that $e_i e'_i = e$. Now as e_i is an S-epi on x we have for all $f : x \to a$ and for all ϵ in S there is a g_i such that the following diagram commutes:

$$\begin{tikzcd}
xT \arrow[r, "e_i"] \arrow[d, "fT"'] & b_i \arrow[r, "e_i'"] \arrow[d, dashed, "g_i"] & b \\
aT \arrow[r, "\epsilon"'] & a &
\end{tikzcd}$$

This can be read as saying that 'a' is the apex of a cone on the diagram whose colimit is the co-intersection. Thus there is a unique h: b -> a such that

$$\begin{tikzcd} xT \arrow[r, "e"] \arrow[d, "fT"'] & b \arrow[d, dashed, "h"] \\ aT \arrow[r, "\epsilon"'] & a \end{tikzcd}$$

commutes. Thus e is an S-epi on x as required.

We may now proceed with the construction of our new theory $\underline{T}' = (T', \eta', \mu')$.

We define the object part of the functor T' as follows. If x is an object of $\underline{C}$ and the co-intersection of S-epis on x is e : xT -> b then the rule T' : x $\mapsto$ b defines the object part as required.

The morphism part of T' uses the preservation by pushouts. Thus given a $\underline{C}$-morphism g : x -> y, we need a morphism gT' : xT' -> yT'. Consider the diagram:

$$\begin{tikzcd} xT \arrow[r, "e_i"] \arrow[d, "gT"'] \arrow[dr, phantom, "(1)"] & b_i \arrow[r, "e_i'"] \arrow[d] & xT' \\ yT \arrow[r, "f_i"'] & c_i \arrow[r, "f_i'"'] & yT' \end{tikzcd}$$

where e_i is an S-epi, (1) is a pushout square, $e_i e'_i$ the co-intersection of S-epis on x and, using the fact that pushouts preserve S-epis, $f_i f'_i$ is the co-intersection of S-epis on y. Thus the colimiting property of the co-intersection says there is a unique h : xT -> xT' such that the square commutes:

$$\begin{tikzcd} xT \arrow[r, "e"] \arrow[d, "gT"'] \arrow[dr, phantom, "(3)"] & xT' \arrow[d, dashed, "h"] \\ yT \arrow[r, "f_i f_i'"'] & yT' \end{tikzcd}$$

This h is the required morphism to be denoted h = gT'.

Note also that (3) says that the map which takes an object, x, onto the co-intersection of S-epis on x is a natural transformation which we will denote by

$q : T \dot{\to} T'$.

That this T' so constructed is a functor is easily established. That it has the same cocontinuity properties as T follows from the 'interchange of colimits' theorem (page 112). We now construct the natural transformations η' and μ' of the new theory $\underline{T} = (T', \eta', \mu')$.

The unit of the theory is simply the vertical composition: $\eta' = \eta \cdot q : I \dot{\to} T \dot{\to} T'$, where I is the identity functor on $\underline{C}$.

Before we define the composition μ' in the new theory $\underline{T}'$ we give a little lemma which is useful in establishing the naturality of maps when we are dealing with epis.

<u>Lemma</u> (Inherited naturality)

If $\underline{C}$ is a category, U, V, W endofunctors on $\underline{C}$ and $\gamma : U \dot{\to} V$, $\delta : U \dot{\to} W$ are natural transformations with γ an epi (i.e. for each x in $\underline{C}$, $x\gamma$ is an epi) then if there is an $\theta_x : xV \to xW$ such that for all x in $\underline{C}$;

$$\begin{tikzcd} xU \arrow[r, "x\gamma"] \arrow[dr, "x\delta"'] & xV \arrow[d, "\theta_x"] \\ & xW \end{tikzcd}$$

commutes, then θ_x is a natural transformation in x.

The proof is merely "diagram chasing" and will be omitted.

We are now in a position to construct the composition $\mu' : T'^2 \dot{\to} T'$. We start with two propositions:

<u>Proposition</u>

If $e_i : xT \to b_i$ is an S-epi and

$$\begin{tikzcd}
xT^2 \arrow[r, "e_i"] \arrow[d, "x\mu"'] & b_iT \arrow[d, dashed] \\
xT \arrow[r, dashed, "u_i"'] & p_i
\end{tikzcd}$$

is a pushout square, then $u_i : xT \to p_i$ is an S-epi on x.

Informally - "pushout of e_iT along preserves S-epis".

<u>Proposition</u>

There is a unique natural transformation $\mu^+ : T'T \dot{\to} T'$ such that

$$\begin{tikzcd}
T^2 \arrow[r, "qT"] \arrow[d, "\mu"'] & T'T \arrow[d, dashed, "\mu^+"] \\
T \arrow[r, "q"'] & T'
\end{tikzcd}$$

commutes.

The proof of these two propositions is rather routine and will be omitted. We note however that the existence of such a μ^+ uses the fact that T preserves co-intersections of the form above. Naturality of μ^+ is a consequence of the Inherited Naturality lemma as qT and $\mu \cdot q$ are natural and qT is an epi as T preserves epis.

<u>Proposition</u>

If $e_i : xT'T \to b_i$ is an S-epi on xT' and

$$\begin{tikzcd}
xT'T \arrow[r, "e_i"] \arrow[d, "x\mu^+"'] & b_i \arrow[d, dashed] \\
xT' \arrow[r, dashed, "u_i"'] & p_i
\end{tikzcd}$$

is a pushout square, then $xq.u_i : xT \to xT' \to p_i$ is an S-epi on x.

The situation is now:

$$\begin{tikzcd}
xT^2 \arrow[r, "xqT"] \arrow[d, "x\mu"'] \arrow[dr, phantom, "(1)"] & xT'T \arrow[r, "e_i"] \arrow[d] \arrow[dr, phantom, "(2)"] & b_i \arrow[r, "e_i'"] \arrow[d] & xT'^2 \\
xT \arrow[r, "xq"'] & xT' \arrow[r, "u_i"'] & p_i \arrow[r, "u_i'"'] & xT'
\end{tikzcd}$$

where (1) is the definition of μ^+, (2) is the pushout square and e_i', u_i' are the morphisms to the co-intersections (using the above proposition).

Thus the universality of the co-intersection of the S-epis on xT' gives a unique μ'_x : xT'T' -> xT' such that

$$\begin{tikzcd}
xT^2 \arrow[r, "xqq"] \arrow[d, "x\mu"'] & xT'^2 \arrow[d, dashed, "\mu'_x"] \\
xT \arrow[r, "xq"'] & xT'
\end{tikzcd}$$

commutes.

Further by the Inherited Naturality lemma, as xqq is an epi for all x, we have that μ'_x is natural in x.

Proposition

There is a unique natural transformation μ' : T'T' $\dot{\to}$ T' such that

$$\begin{tikzcd} T^2 \arrow[r, "qq"] \arrow[d, "\mu"'] & T'^2 \arrow[d, dashed, "\mu'"] \\ T \arrow[r, "q"'] & T' \end{tikzcd}$$

commutes.

Proof : Immediate from the previous work.

It is now an easy task to show that $\underline{T}' = (T', \eta', \mu')$ really is a theory i.e. satisfies the theory laws.

Furthermore we have that $q : T \overset{\cdot}{\to} T'$ is a theory morphism i.e.

$$\begin{tikzcd} & I \arrow[dl, "\eta"'] \arrow[dr, "\eta'"] & \\ T \arrow[rr, "q"'] & & T' \end{tikzcd} \qquad \begin{tikzcd} T^2 \arrow[r, "qq"] \arrow[d, "\mu"'] & T'^2 \arrow[d, "\mu'"] \\ T \arrow[r, "q"'] & T' \end{tikzcd}$$

both commute. Hence we may write $q : \underline{T} \to \underline{T}'$.

This completes the construction of the theory. We now establish its universal property. Suppose $\underline{T}''$ is another theory on $\underline{C}$ and there is an epi $q' : \underline{T} \to \underline{T}''$ which is an S-epi. Then because, for each x in $\underline{C}$, xq is a co-intersection, there is a unique $\rho_x : xT'' \to xT'$ such that the following commutes.

$$\begin{tikzcd} xT \arrow[rr, "xq'"] \arrow[dr, "xq"'] & & xT'' \arrow[dl, "\rho_x"] \\ & xT' & \end{tikzcd}$$

By the inherited naturality lemma ρ_x is natural in x, so we may write ρ : T" $\dot{\to}$ T. It remains to show that ρ is a theory morphism. This follows from the fact that T preserves epis and that q and q' are epis and theory morphisms. This morphism ρ is evidently unique with the property and thus we have established the universal condition on $\underline{T}'$. $\square$

Properties of The Construction

A denotational semantics of a language can be used to check syntactic identities which are believed to hold. That is, we can prove that an equational specification of the semantics is subsumed within the denotational semantics given. These category theory constructions are intended to provide part of a 'categorical' semantics for CLEAR. We would wish that properties of the constructions reflected syntactic identities that are assumed to hold. We look at an example of this.

Choose S to contain only the initial algebra of a theory $\underline{T}$ (assuming that $\underline{C}$ has an initial object). Then denote the theory constructed as above from this S by Induce($\underline{T}$) (this is in keeping with the terminology of [Burstall,Goguen 1977]). The theory Induce($\underline{T}$) can be thought of as $\underline{T}$ together with all the equations which are true in the initial algebra of $\underline{T}$. From this intuitive description of Induce($\underline{T}$) we might expect that 'Induce Induce = Induce' should hold. That is, 'Induce' is idempotent. A priori, the sense in which the two expressions are equal has not been defined. This 'equality' turns out to be a categorical isomorphism.

Using the notation of the previous parts we have:

Proposition

If $\underline{T}'$ = Induce($\underline{T}$) (i.e. S contains only the initial $\underline{T}$-algebra), then q : T $\dot{\to}$ T' is a section (i.e. has a right inverse).

Proof

$\emptyset$ is initial in $\underline{C}$ and the unique morphism $\emptyset \to \emptyset T$ is $\emptyset\eta : \emptyset \to \emptyset T$, hence we have: there is a k : $\emptyset T' \to \emptyset T$ such that:

$$\begin{tikzcd}
\phi T \arrow[r, "\phi q"] \arrow[d, "\phi\eta T"'] \arrow[dr, "="] & \phi T' \arrow[d, "k"] \\
\phi T^2 \arrow[r, "\phi\mu"'] \arrow[ur, phantom, "(1)"] & \phi T
\end{tikzcd}$$

commutes, where (1) commutes by theory law for T and the whole square commutes as ϕq is an S-epi on ϕ. Hence ϕq.k = $i_{\phi T}$, as required.

Proposition

$$\epsilon = \phi T'q.\phi\mu' : \phi T'T \to \phi T'T' \to \phi T'$$

is a T-algebra. (Proof omitted)

Now let Induce Induce(T) = Induce(T') = T" = (T",η",μ") and let q' : T' $\dot{\to}$ T" be the theory morphism taking T' into Induce(T'). Then, given f : x -> ϕT', we have the diagram:

$$\begin{tikzcd}
xT \arrow[r, "xq"] \arrow[d, "fT"'] & xT' \arrow[r, "xq'"] \arrow[d, "fT'"'] \arrow[l, phantom, "(1)"] & xT'' \arrow[d, dashed, "h"] \arrow[l, phantom, "(2)"] \\
\phi T'T \arrow[r, "\phi T'q"'] & \phi T'^2 \arrow[r, "\phi\mu'"'] & \phi T'
\end{tikzcd}$$

where (1) commutes by the naturality of q and there is an h : xT" -> ϕT' such that (2) commutes as q' is a {$\phi\mu'$}-epi.

Hence, q and q' are both {ϵ}-epis where $\epsilon = \phi T'q.\phi\mu'$. We can now show that x(q.q') is an {$\phi\mu$}-epi for all x. For, given an f : x -> ϕT we have,

$$\begin{tikzcd}
xT \arrow[rr, "xq"] \arrow[dr, "fT"] \arrow[dd, "(f\phi_q)T"'] & & xT' \arrow[rr, "xq'"] \arrow[d, phantom, "(2)"] & & xT'' \arrow[dl, dashed, "hk"'] \arrow[dd, "h"] \\
(1) & \phi T^2 \arrow[rr, "\phi\mu"'] \arrow[dl, "\phi_q T"] & {} & \phi T & (4) \\
\phi T'T \arrow[rr, "\phi T'_q"'] & & \phi T'^2 \arrow[rr, "\phi\mu'"'] \arrow[u, phantom, "(3)"] & & \phi T' \arrow[ul, dashed, "k"]
\end{tikzcd}$$

where k is the (unique) right inverse of $\phi q : \phi T \to \phi T'$ and h is the unique morphism which makes the outside square commute - which exists because x(q.q') is an $\{\epsilon\}$-epi. (3) commutes as q is a theory morphism. (2) now commutes as all other polygons commute.

Hence we have x(q.q') is an $\{\phi\mu'\}$-epi. But xq : xT -> xT' is the co-intersection of all such epis thus there is an r : xT" -> xT' such that:

$$\begin{tikzcd}
xT \arrow[r, "xq"] \arrow[drr, "xq"'] & xT' \arrow[r, "xq'"] & xT'' \arrow[d, dashed, "r"] \\
& & xT'
\end{tikzcd}$$

Thus $xq' . r = i_{xT'}$ as xq is an epi. So xq' is a <u>section</u>, it is a epi, hence it is an isomorphism, by a general result (consult [Herrlich,Strecker 1973]). $\square$

An obvious question which springs to mind - is "Induce" a functor between suitable categories? Consider the example:

$$\begin{tikzcd}
\boxed{\begin{array}{l}\texttt{constant } \underline{\texttt{One}} \texttt{ =} \\ \quad \texttt{sorts thing} \\ \quad \texttt{opns tt : thing}\end{array}} \arrow[r, dashed, "\text{Induce}"] \arrow[d, "\alpha = \left| \begin{array}{l} \texttt{thing -> thing'} \\ \texttt{tt -> tt'} \end{array}\right."] & \boxed{\begin{array}{l}\texttt{Induce } \underline{\texttt{One}} \texttt{ =} \\ \quad \texttt{sorts thing} \\ \quad \texttt{opns tt : thing} \\ \quad \texttt{eqns x = tt}\end{array}} \arrow[d, "\text{?? Induce}(\alpha)"] \\
\boxed{\begin{array}{l}\texttt{constant } \underline{\texttt{Two}} \texttt{ =} \\ \quad \texttt{sorts thing'} \\ \quad \texttt{opns tt' : thing'} \\ \quad \texttt{   ff' : thing'}\end{array}} \arrow[r, dashed, "\text{Induce}"] & \boxed{\begin{array}{l}\texttt{Induce } \underline{\texttt{Two}} \\ \quad \texttt{= } \underline{\texttt{Two}}\end{array}}
\end{tikzcd}$$

No morphism, Induce(α), exists which makes the diagram commute.

The error seems to be the α is not 'onto'. However, 'epiness' does not seem to be sufficient to construct the morphism Induce(α). We show that the stronger assumption that α is a retraction (i.e. has a left inverse) is sufficient.

We proceed as in previous cases to show that given an $\alpha : \underline{S} \to \underline{T}$ there is a unique $\alpha' :$ Induce $\underline{S}$ -> Induce $\underline{T}$ such that

$$\begin{tikzcd}
\underline{S} \arrow[r, "q_{\underline{S}}"] \arrow[d, "\alpha"'] & \text{Induce } \underline{S} \arrow[d, dashed, "\alpha'"] \\
\underline{T} \arrow[r, "q_{\underline{T}}"'] & \text{Induce } \underline{T}
\end{tikzcd}$$

commutes, provided that α is a retraction (i.e. x is a retraction for all objects x).

[Here the $q_{\underline{S}}$ and the $q_{\underline{T}}$ are co-intersections of $\{\phi\mu_{\underline{S}}\}$-epis and $\{\phi\mu_{\underline{T}}\}$-epis respectively]

Proposition

If the following is a pushout square

$$\begin{tikzcd} xS \arrow[r, "e_i"] \arrow[d, "x\alpha"'] & b_i \arrow[d, dashed] \\ xT \arrow[r, dashed, "g_i"'] & p_i \end{tikzcd}$$

with α a retraction and e_i a $\{\phi\mu_{\underline{S}}\}$-epi then $g_i : xT \rightarrow p_i$ is a $\{\phi\mu_{\underline{T}}\}$-epi.

Proof

We use the fact that given $g : x \rightarrow \phi T$ there is an $h : x \rightarrow \phi S$ such that

$$\begin{tikzcd} & x \arrow[dl, "h"'] \arrow[dr, "g"] & \\ \phi S \arrow[rr, "\phi\alpha"'] & & \phi T \end{tikzcd}$$

commutes. The proof then proceeds mechanically.

The proof of the existence of the α' (as a theory morphism) is now concluded by the same technique as in the previous existence proofs.

$\square$

We now return to the discussion of data theories.

DATA THEORIES ARE MONADIC

It was shown in [Burstall,Goguen 1980a] that the data constraints of data theories behave somewhat like equations and therefore that a general formulation of theories which encompassed equational theories also included data theories. Here we give a similar result for monadic theories. The previous two propositions enable us to construct monadic theories from data constraints.

Suppose that $\underline{T} = (T, \eta, \mu)$ is a monadic theory on a category $\underline{C}$. Let

(F,G) be a data constraint with $F : \underline{S} \to \underline{T}$ for some theory $\underline{S}$ and $G : \underline{T} \to \underline{U}$ for some theory $\underline{U}$ (G may arise from a signature morphism if $\underline{T}$ and $\underline{U}$ have associated signatures). Thus we have the picture:

$$\begin{tikzcd}\underline{S} \arrow[r, "F"'] & \underline{T} \arrow[r, "G"'] & \underline{U}.\end{tikzcd}$$

Assuming that the theories $\underline{S}$, $\underline{T}$ and $\underline{U}$ satisfy the requisite conditions (those concerned with preservation of colimits), we can construct another theory which is to be $\underline{U}$ "data constrained" by (F,G) as follows:

By the first proposition above, if $F^{+} : \underline{\mathrm{Alg}}(\underline{T}) \to \underline{\mathrm{Alg}}(\underline{S})$ is the forgetful functor obtained from the theory morphism F, then there is a left adjoint to F^{+}, $F^{*} : \underline{\mathrm{Alg}}(\underline{S}) \to \underline{\mathrm{Alg}}(\underline{T})$. If $G^{+} : \underline{\mathrm{Alg}}(\underline{U}) \to \underline{\mathrm{Alg}}(\underline{T})$ is the forgetful functor obtained from G, consider the class of $\underline{U}$-algebras, Q, defined such that

$$G^{+}(Q) = F^{*}(\mathrm{Obj}(\underline{\mathrm{Alg}}(\underline{S})))$$

That is, when forgotten to $\underline{T}$-algebras they are the image of F^{*}, ("Obj" gives the objects of a category).

Now use the construction of the second proposition with $\underline{U}$ as the theory and Q as the class of $\underline{U}$-algebras. We obtain a new theory $\underline{U}'$ and a theory morphism $q : \underline{U} \to \underline{U}'$. This is the required monadic theory obtained by enriching a monadic theory with a data constraint.

As an example consider the following specification:

```
    constant Nat =
            data sorts nat
                 opns  0 : nat
                       succ : nat -> nat
                                               end
    constant NatPlus =
            enrich Nat by
                 opns + : nat,nat -> nat
                 eqns  0 + n = n
                       succ(m) + n = succ(m + n)
                                                       end
```

Now if $\underline{\text{Nil}}$ is the empty theory, the data constraint corresponding to the "data" in the specification consists of the following two morphisms:

$$\begin{tikzcd}\underline{\text{Nil}} \arrow[r, "\Phi"] & \underline{\text{Nat}} \arrow[r, "F"] & \underline{\text{NatPlus}}\end{tikzcd}$$

where Φ is the unique morphism from the empty theory to $\underline{\text{Nat}}$ (this is a morphism of many-sorted theories - not a straightforward monad morphism, see page (14) for an explanation of this). The morphism F is the inclusion of sorts and operations.

Now assuming that the requisite conditions hold for the theories Nil, Nat and NatPlus, we can construct a new theory - NatPlus data constrained by $\langle\Phi,F\rangle$ - as follows. The class of algebras determined by the morphism Φ is simply the class containing only the initial algebra of Nat. In the above notation, the initial algebra is the only element of F^*(Obj(Alg(Nil))). Thus the NatPlus algebras that we are considering must all have the "Nat-part" as this initial algebra. Such algebras must all satisfy extra equations, for example:

all m,n: nat m + n = n + m

which arises by induction rather than by equational deduction. Thus the theory denoted by the specification has this extra equation. The monadic theory constructed as above can thus be thought of as NatPlus with these extra equations.

Notes

Kaphenst and Reichel (1971) with their 'canons' introduced the notion which we call a data constraint. Data constraints are described in [Burstall and Goguen 1980a]. The idea behind the second construction in this chapter arose from a discussion with Dr. Gordon Plotkin.

CHAPTER TEN

COLIMITS AND FACTORISATIONS IN CATEGORIES OF THEORIES

We have seen that colimits give a categorical interpretation of the building of new objects from old and that expressing objects as colimits of diagrams gives an associated decomposition. Putting theories together to build specifications is an example of this and the use of colimits in this setting has (as described above) been noted by [Burstall,Goguen 1980a] as well as by [Ehrich 1978]. Factorisations are another way of obtaining new theories from old and are associated with data abstraction.

COLIMITS OF THEORIES

Theories in a Lawvere form have finite colimits [Lawvere 1963b] and so do those in the logical form of [Burstall,Goguen 1980a]. We therefore must ask - does the category of monadic theories have finite colimits? The answer is yes, at least for suitable theories on a suitable base category. We establish this by showing that binary coproducts and binary coequalisers of theories exist and finally that an initial theory exists. This is sufficient to prove the result by the colimit existence theorem (page 36).

Theorem

The category of monadic theories (with cocontinuous endofunctors) on an (arbitrarily) cocomplete base category $\underline{C}$ has binary coproducts.

(The cocontinuity can be weakened to require only that the endofunctors preserve colimits of the diagram D below.)

Proof

Suppose $\underline{T}_1 = (T_1, \eta_1, \mu_1)$ and $\underline{T}_2 = (T_2, \eta_2, \mu_2)$ are two theories (with T_1, T_2 preserving the colimits of diagrams of the form of D below) on a cocomplete category $\underline{C}$.

Our first attempt may be to suppose that, like the coproduct of monadic signatures (page 123), the coproduct of $\underline{T}_1$ and $\underline{T}_2$ arises by a pushout. This is not the case. We cannot define a composition operation, which preserves the compositions in $\underline{T}_1$ and $\underline{T}_2$, on the endofunctor arising from the pushout. A more elaborate construction is required.

In the Lawvere understanding of theories, the coproduct of two theories is given by a pushout diagram in the category of categories. This involves the transitive closure of part of the underlying graph of a category - i.e. something like a freeness construction. In the following, this is reflected by a colimit of an infinite diagram, just as free theories were given by colimits of ω-chains.

The construction goes as follows. First consider words on the alphabet $\{ T_1, T_2 \}$. Each word denotes, by functor composition, a functor - the empty word being the identity functor on $\underline{C}$. Let W, W', W" be variables ranging over words (and over the functors that the words denote) on this alphabet and let $|W|$ be the length of word W.

Consider the diagram in $\underline{C}^{\underline{C}}$, which we call D, whose nodes are labelled by these words, each node having as object the functor denoted by its label. The morphisms of D are the following:

$W\eta_i W' : WW' \dot{\to} WT_iW'$ and

$W\mu_i W' : WT_i^2W' \dot{\to} WT_iW'$

for i = 1,2 and all words W,W'.

Let ζ_W: W -> U be the colimit of D. Notice that U preserves the colimit of D (page 112). In this case the approximation and existence lemmas that we need look like:

The 1-approximation lemma:

If, for all W in D:

$$\begin{tikzcd} W \arrow[r, "\zeta_W"] \arrow[d, "\zeta_W"'] & U \arrow[d, "f"] \\ U \arrow[r, "g"'] & V \end{tikzcd}$$

commutes, then f = g.

The 2-existence lemma:

If, for all W and W', $h_{W,W'}$: WW' -> V is separately a cone in W and W', then there is a unique h : U^2 -> V such that

$$\begin{tikzcd}
WW' \arrow[r, "\zeta_W \zeta_{W'}"] \arrow[dr, "h_{W,W'}"'] & U^2 \arrow[d, dashed, "h"] \\
& V
\end{tikzcd}$$

commutes.

Now we construct a theory with U as endofunctor, $\underline{U}$ = (U, η, μ).

We define η by $\eta = \zeta_I$: $I \overset{\cdot}{\to} U$ where I is the identity functor on category $\underline{C}$.

Then define μ, from the 2-existence lemma, as the unique morphism such that

$$\begin{tikzcd}
WW' \arrow[r, "\zeta_W \zeta_{W'}"] \arrow[dr, "\zeta_{WW'}"'] & U^2 \arrow[d, dashed, "\mu"] \\
& U
\end{tikzcd}$$

commutes ($\zeta_{WW'}$ automatically is a cone separately in W and W').

To verify that $\underline{U}$ really is a theory, notice that by setting W' = I in the triangle above, we have

$$\begin{tikzcd}
W \arrow[r, "\zeta_W"] \arrow[drr, "\zeta_W"'] & U \arrow[r, "U\eta"] & U^2 \arrow[d, "\mu"] \\
& & U
\end{tikzcd}$$

commutes. Hence by the 1-approximation lemma

$$\begin{tikzcd}
U \arrow[r, "U\eta"] \arrow[dr, "="'] & U^2 \arrow[d, "\mu"] \\
& U
\end{tikzcd}$$

commutes. This is one theory law. Setting W = I instead, gives another theory law.

For the commutation of the square:

$$\begin{tikzcd}
U^3 \arrow[r, "\mu U"] \arrow[d, "U\mu"'] & U^2 \arrow[d, "\mu"] \\
U^2 \arrow[r, "\mu"'] & U
\end{tikzcd}$$

expand, using the 3-approximation. Thus we need to prove that, for all W, W', W" in D:

$$\begin{tikzcd}
WW'W'' \arrow[rr, "\zeta_W\zeta_{W'}\zeta_{W''}"] \arrow[dd, "\zeta_W\zeta_{W'}\zeta_{W''}"'] & & U^3 \arrow[d, "\mu U"] \\
& & U^2 \arrow[d, "\mu"] \\
U^3 \arrow[r, dash, "U\mu"'] & U^2 \arrow[r, "\mu"'] & U
\end{tikzcd}$$

commutes. But it is

readily verified that either way round the square is $\zeta_{WW'W''}$. Thus $\underline{U}$ is a theory.

All that remains to do is to show that it is the coproduct of $\underline{T}_1$ and $\underline{T}_2$.

Define theory morphisms $t_i : \underline{T}_i \to \underline{U}$ by

$t_i = \zeta_{T_i}$ for $i = 1,2$.

These are theory morphisms because, for $i = 1,2$:

$$\begin{tikzcd} & I \arrow[dl, "\eta_i"'] \arrow[dr, "\eta"] & \\ T_i \arrow[rr, "t_i"'] & & U \end{tikzcd}$$

commutes as ζ_W is a cone on D, and

$$\begin{tikzcd} T_i^2 \arrow[rr, "\zeta_{T_i}\zeta_{T_i}"] \arrow[d, "\mu_i"'] \arrow[drr, "\zeta_{T_i^2}"] & \text{(1)} & U^2 \arrow[d, "\mu"] \\ T_i \arrow[rr, "\zeta_{T_i}"'] & \text{(2)} & U \end{tikzcd}$$

commutes because (1) is the definition of

and (2) commutes because ζ_W is a cone on D.

Now suppose that $\underline{V} = (V, \eta', \mu')$ is another theory, with theory morphisms, for $i = 1,2$

$v_i : \underline{T}_i \to \underline{V}$.

Construct a cone on D by:

$\nu_I = \eta' : I \to V$

$\nu_W = v_W \cdot \mu'_W : W \to V$ for all $W \neq I$

where v_W is the horizontal composition of v_1's and v_2's obtained by substituting v_1 for T_1 and v_2 for T_2 in the word W and μ'_W is defined by,

$$\mu'_V = i_V$$
$$\mu'_{V^2} = \mu'$$
$$\mu'_{V^n} = \mu'_{V^{n-2}} . \mu'_{V^{n-1}}$$

Then by the 1-existence lemma, there is a unique $u : U \to V$ such that, for all W,

$$\begin{tikzcd} W \arrow[r, "\zeta_W"] \arrow[dr, "\nu_W"'] & U \arrow[d, dashed, "u"] \\ & V \end{tikzcd}$$

commutes.

This u is a theory morphism as:

$$\begin{tikzcd} & I \arrow[dl, "\eta"'] \arrow[dr, "\eta'"] & \\ U \arrow[rr, "u"'] & & V \end{tikzcd}$$

commutes from the definition of u and

$$\begin{tikzcd} U^2 \arrow[r, "u^2"] \arrow[d, "\mu"'] & V^2 \arrow[d, "\mu'"] \\ U \arrow[r, "u"'] & V \end{tikzcd}$$

commutes by expanding as a 2-approximation:

$$\begin{tikzcd}
WW' \arrow[rr, "\zeta_W\zeta_{W'}"] \arrow[d, "\zeta_W\zeta_{W'}"'] & & U^2 \arrow[d, "\mu"] \\
 & & U \arrow[d, "u"] \\
U^2 \arrow[r, "u^2"'] & V^2 \arrow[r, "\mu'"'] & V
\end{tikzcd}$$

This commutes for all W and W' in D as both ways round it is $\zeta_{WW'}$.

Thus we have proved the universality of $\underline{U}$, that is, $\underline{U}$ is the coproduct of the theories $\underline{T}_1$ and $\underline{T}_2$. $\square$

We now turn our attention to coequalisers of theories.

Theorem

The category of monadic theories, whose endofunctors are cocontinuous, on an (arbitrarily) cocomplete category $\underline{C}$ has coequalisers of pairs of morphisms.

(The cocontinuity condition can be weakened somewhat to require only that epis, ω-colimits and the coequalisers and pushouts below are preserved.)

Proof

Let $\underline{S} = (S,\eta_S,\mu_S)$ and $\underline{T} = (T,\eta_T,\mu_T)$ be two cocontinuous theories on a cocomplete category $\underline{C}$ and let

$f,g : \underline{S} \to \underline{T}$

be two theory morphisms.

We may suspect that, like the coequalisers of monadic signatures, coequalisers of theories are created by those in $\underline{C}^{\underline{C}}$. However, the coequaliser of

$$\begin{tikzcd}
S \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & T
\end{tikzcd}$$

in $\underline{C}^{\underline{C}}$ does not respect the composition in the theories. It is an equivalence and not a congruence. We build the required congruence iteratively using pushouts and take an ω-colimit to construct the coequalising theory. Iterated pushouts and the associated ω-colimits are used elsewhere in category theory to construct colimits. For instance Adamek and Koubek [1980] have used them to construct colimits of algebras. There seems to be a general understanding of this construction - iterated pushouts and the associated ω-colimit are used to obtain the "closure" of an object under an operation.

Define inductively a sequence of functors, U_n, and natural transformations, $u_n : U_n \dot{\to} U_{n+1}$ $n \geq 0$, by $U_0 = T$, and u_0 is given by the coequaliser:

$$\begin{tikzcd} S \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & T \arrow[r, dashed, "u_0"] & U_1 \end{tikzcd}$$

Inductively define U_{n+2} and u_{n+2} by the pushout:

$$\begin{tikzcd} U_n^2 \arrow[r, "u_n^2"] \arrow[d, "k_n"'] & U_{n+1}^2 \arrow[d, dashed, "k_{n+1}"] \\ U_{n+1} \arrow[r, dashed, "u_{n+1}"'] & U_{n+2} \end{tikzcd}$$

where $k_0 = \mu_T . u_0$. Notice that u_0 is an epi because it is a coequaliser, thus as each U_n preserves epis, u_n is an epi.

Now consider the chain

$$\begin{tikzcd} U_0 \arrow[r, "u_0"] & U_1 \arrow[r, "u_1"] & U_2 \arrow[r, "u_2"] & \cdots & U_n \arrow[r, "u_n"] & U_{n+1} \arrow[r, "u_{n+1}"] & \cdots \end{tikzcd}$$

Call it Δ.

Let its colimit be $\zeta_n : U_n \to U$ for all $n \geq 0$. Then

$$\begin{tikzcd} U_n \arrow[rr, "u_n"] \arrow[dr, "\zeta_n"'] & & U_{n+1} \arrow[dl, "\zeta_{n+1}"] \\ & U & \end{tikzcd}$$

commutes.

We make U into the endofunctor of a theory $\underline{U} = (U, \eta, \mu)$ by defining a η and a μ:

$\eta : I \dot{\to} U$ is given by $\eta = \eta_T \cdot \zeta_0$

μ again arises from the 2-existence lemma, but because Δ is an ω-chain, we can use the stronger form to say that μ is defined to be the unique morphism such that, for all $n \geq 0$,

$$\begin{tikzcd} U_n^2 \arrow[r, "\zeta_n^2"] \arrow[d, "k_n"'] & U^2 \arrow[d, dashed, "\mu"] \\ U_{n+1} \arrow[r, "\zeta_{n+1}"'] & U \end{tikzcd}$$

commutes.

To verify the theory properties of $\underline{U}$ is rather routine.

For instance, we need to show that

$$\begin{tikzcd} U \arrow[r, "U\eta"] \arrow[dr, "="'] & U^2 \arrow[d, "\mu"] \\ & U \end{tikzcd}$$

commutes. Using the 1-approximation lemma, it suffices to show that:

$$\begin{tikzcd}
U_i \arrow[rrrr, "\zeta_i"] \arrow[dr, "U_i\eta_T"'] \arrow[dddd, "\zeta_i"'] & \text{(1)} & & & U \arrow[dl, "U\eta_T"'] \arrow[dd, "U\eta"] \\
& U_iT \arrow[rr, "\zeta_iT"] \arrow[dr, "U_iu_i^*"'] & \text{(3)} & UT \arrow[dr, "U\zeta_0"] & \text{(2)} \\
\text{(6)} & & U_i^2 \arrow[rr, "\zeta_i^2"'] \arrow[d, "k_i"'] & & U^2 \arrow[dd, "\mu"] \\
& & U_{i+1} \arrow[dll, "\zeta_{i+1}"'] \arrow[drr, "\zeta_{i+1}"] & \text{(4)} & \\
U \arrow[rrrr, "="'] & & \text{(5)} & & U
\end{tikzcd}$$

where $u_i^* = u_0u_1 \ldots\ldots u_i$. Here, (1) commutes by the naturality of η_T, (2) by definition of η and (3) is

$$\begin{tikzcd}
U_iT \arrow[rr, "\zeta_iT"] \arrow[dd, "U_iu_i^*"'] & & UT \arrow[dl, "Uu_i^*"'] \arrow[dd, "U\zeta_0"] \\
& UU_i \arrow[dr, "U\zeta_i"] & \\
U_i^2 \arrow[ur, "\zeta_iU_i"] \arrow[rr, "\zeta_i^2"'] & & U^2
\end{tikzcd}$$

which commutes, (4) is the definition of μ and (5) is trivial.

Only (6) remains - we need an inductive argument:

We want to show that for all $n \geq 0$ the following commutes.

$$\begin{tikzcd}
U_i \arrow[r, "U_i\eta_T"] \arrow[d, "\zeta_i"'] & U_iT \arrow[r, "U_iu_i^*"] & U_i^2 \arrow[d, "k_i"] \\
U & & U_{i+1} \arrow[ll, "\zeta_{i+1}"]
\end{tikzcd}$$

If $n = 0$, this reduces to

$$\begin{tikzcd}
T \arrow[r, "T\eta_T"] \arrow[dd, "\zeta_0"'] & T^2 \arrow[r, "="] & T^2 \arrow[d, "\mu_T"] \\
& & T \arrow[d, "u_0"] \\
U & & U_1 \arrow[ll, "\zeta_1"]
\end{tikzcd}$$

which commutes by a theory property of $\underline{T}$ and the fact that ζ_n is a cone on Δ. Assume that the required square commutes for a fixed n, then show that it commutes for n+1. We have the situation:

$$\begin{tikzcd}
T \arrow[r, "u_i^*"] \arrow[drr, "T\eta_T"'] \arrow[ddd, "\zeta_0"'] & U_i \arrow[r, "U_i\eta_T"] & U_iT \arrow[r, "U_iu_i^*"] & U_i^2 \arrow[ddd, "k_i"] \\
& & T^2 \arrow[u, "u_i^*T"] \arrow[ur, "u_i^{*2}"'] \arrow[d, "\mu_T"'] & \\
& & T \arrow[dll, "\zeta_0"'] \arrow[dr, "u_{i+1}^*"] & \\
U & & & U_{i+1} \arrow[lll, "\zeta_{i+1}"]
\end{tikzcd}$$

Each inner piece commutes so the whole square commutes.

The other theory laws are established by a similar reduction to an inductive argument.

If we define h : $\underline{T}$ -> $\underline{U}$ by h = ζ_0, then it is indeed a theory morphism. Moreover, we now show that it is the coequaliser of f and g.

Let β : $\underline{T}$ -> $\underline{V}$ be another theory morphism such that $f\beta = g\beta$. We prove that there is a unique γ such that

$$\begin{tikzcd}
\underline{T} \arrow[r, "h"] \arrow[dr, "\beta"'] & \underline{U} \arrow[d, dashed, "\gamma"] \\
& \underline{V}
\end{tikzcd}$$

commutes. Construct γ inductively.

Define γ_0 to be the unique morphism such that

$$\begin{tikzcd}
T \arrow[r, "u_0"] \arrow[dr, "\beta"'] & U_1 \arrow[d, dashed, "\gamma_0"] \\
& V
\end{tikzcd}$$

commutes (using the colimiting property of the coequaliser). Now suppose that we have γ_{i-1} and γ_i such that

$$\begin{tikzcd}
U_i \arrow[r, "u_i"] \arrow[dr, "\gamma_{i-1}"'] & U_{i+1} \arrow[d, "\gamma_i"] \\
& V
\end{tikzcd}
\qquad
\begin{tikzcd}
U_i^2 \arrow[r, "\gamma_{i-1}^2"] \arrow[d, "k_i"'] & V^2 \arrow[d, "\mu_V"] \\
U_{i+1} \arrow[r, "\gamma_i"'] & V
\end{tikzcd}$$

both commute. Construct a γ_{i+1} as the unique morphism making the following diagram commute,

$$\begin{tikzcd}
U_i^2 \arrow[rr, "u_i^2"] \arrow[dr, "\gamma_{i-1}^2"'] \arrow[ddd, "k_i"'] & & U_{i+1}^2 \arrow[dl, "\gamma_i^2"] \arrow[ddd, "k_{i+1}"] \\
& V^2 \arrow[d, "\mu_V"'] & \\
& V & \\
U_{i+1} \arrow[ur, "\gamma_i"] \arrow[rr, "u_{i+1}"'] & & U_{i+2} \arrow[ul, dashed, "\gamma_{i+1}"']
\end{tikzcd}$$

(using the colimiting property of the pushout).

Now, by definition, the γ_i's form a cone on the chain Δ. So there is a unique $\gamma : U \dot{\to} V$ such that

$$\begin{tikzcd}
 & U_i \arrow[dl, "\zeta_i"'] \arrow[dr, "\gamma_i"] & \\
U \arrow[rr, "\gamma"'] & & V
\end{tikzcd}$$

commutes. This is the required morphism. We need only show that it is a theory morphism and has the required uniqueness.

Set i = 0 for one condition and for the commutation of

$$\begin{tikzcd}
U^2 \arrow[r, "\gamma^2"] \arrow[d, "\mu"'] & V^2 \arrow[d, "\mu_V"] \\
U \arrow[r, "\gamma"'] & V
\end{tikzcd}$$

use the 2'-approximation lemma. Thus it suffices that the following commutes.

$$\begin{tikzcd}
U_i^2 \arrow[r, "\zeta_i^2"] \arrow[d, "\zeta_i^2"'] \arrow[dr, "k_i"'] \arrow[rr, bend right, "\gamma_{i-1}^2"'] & U^2 \arrow[r, "\gamma^2"] & V^2 \arrow[d, "\mu_V"] \\
(2) & U_{i+1} \arrow[d, "\zeta_i"'] \arrow[r, phantom, "(4)"] \arrow[dr, "\gamma_i"] & {} \\
U^2 \arrow[r, "\mu"'] & U \arrow[r, "\gamma"'] & V
\end{tikzcd}$$

(1) (3)

But (1) commutes by the definition of the γ_i's, (2) by the definition of μ , (3) by the definition of γ and (4) by a property of the γ_i. Thus γ is a theory morphism as required. The necessary uniqueness follows from the uniqueness of γ above.

This completes our proof of the existence of coequalisers. $\square$

Notice now that the trivial theory on $\underline{C}$, (I,i,i), where I is the identity functor on $\underline{C}$ and i is the identity natural transformation, is initial in $\underline{Th}(\underline{C})$. Finite cocompleteness now follows from a general theorem (page 36).

We now turn to the factorisations in the category of theories, thus verifying Axiom 3 for monadic theories.

FACTORISATIONS IN THE CATEGORY OF THEORIES

Factorisations in categories are of use in formalising data abstraction mechanisms, in particular the DERIVE operation of CLEAR [Burstall,Goguen 1980b]. We have not so far dealt with factorisations but it seems that we can "compute" them by similar techniques (including lifting and pointwise evaluation) to those used for computing colimits. To complete our discussion of the axioms for theories (page 103), we include here a result about factorisations in the category of monadic theories.

Unlike colimits of theories, certain factorisations of theories are easy - they lift from functor categories where they are computed pointwise.

First several definitions:

Definition

We say that e is an extremal-epi if e is an epi such that the following holds: If e = fm with m a mono then m is an isomorphism.

Definition

Let $\underline{C}$ be a category and $\mathcal{E}$ the class of its extremal-epis and $\mathcal{M}$ its monos, then $\underline{C}$ is said to have extremal-epi,mono factorisations if every morphism, f, in $\underline{C}$ can be expressed as

f = em with e in $\mathcal{E}$ and m in $\mathcal{M}$.

Now several propositions (proofs may be found in [Schubert 1972]).

1. A sufficient condition for a category to have extremal-epi mono factorisations is that the category is cocomplete and co-wellpowered. The proof is constructive.

2. If fe is an extremal-epi, so is e.

3. If $\underline{C}$ is co-wellpowered and cocomplete then extremal-epi,mono factorisations are natural. That is if f = em and f' = e'm' are such factorisations and if

$$\begin{tikzcd} a \arrow[r, "e"] \arrow[d, "i"'] & d \arrow[r, "m"] & b \arrow[d, "j"] \\ a' \arrow[r, "e'"'] & d' \arrow[r, "m'"'] & b' \end{tikzcd}$$

commutes, then there is a unique g : d -> d' which fills in the diagram commutatively. Consequently, these factorisations are unique to within an isomorphism.

With these results, we may proceed. We show first that extremal-epi,mono factorisations in functor categories are computed pointwise.

Let $\underline{C}$ be cocomplete and co-wellpowered - it therefore has extremal-epi,mono factorisations. If $\underline{A}$ is any other category, consider the functor category, $\underline{C}^{\underline{A}}$. Let α : F -> G be a morphism in this category. For each a in $\underline{A}$, aα: aF -> aG factorises uniquely (to within an isomorphism) as,

aF $\xrightarrow{e}$ d $\xrightarrow{m}$ aG with e in $\mathcal{E}$, m in $\mathcal{M}$.

This factorisation is functorial, that is we can define a functor D in $\underline{C}^{\underline{A}}$ such that, on objects, aD = d and, on morphisms (using the naturality of the factorisation):

$$\begin{tikzcd}
aF \arrow[r, "e"] \arrow[d, "fF"'] & aD \arrow[r, "m"] \arrow[d, dashed, "fD"] & aG \arrow[d, "fG"] \\
bF \arrow[r, "e'"'] & bD \arrow[r, "m'"'] & bG
\end{tikzcd}$$

This diagram says that the e, e' and m, m' may be considered as natural transformations which we will call ϵ and ν. Thus in the category of functors, we have the factorisation:

$\alpha = F \xrightarrow{\epsilon} D \xrightarrow{\nu} G.$

ϵ is an extremal epi in $\underline{C}^A$ and ν is a mono, as required.

We now turn to factorisations in the category of theories.

<u>Theorem</u>

If $\underline{C}$ is cocomplete and co-wellpowered (hence has extremal-epi,mono factorisations) and $\underline{Th}^*(\underline{C})$ is the full subcategory of theories on $\underline{C}$ whose endofunctors preserve extremal_epis and monos, then $\underline{Th}^*(\underline{C})$ has extremal-epi,mono factorisations.

<u>Proof</u>

Let $\underline{S} = (S,\eta,\mu)$ and $\underline{T} = (T,\eta',\mu')$ be theories in $\underline{Th}^*(\underline{C})$ and $\alpha : \underline{S} \to \underline{T}$. By the previous discussion there is a functor $U : \underline{C} \to \underline{C}$ and a factorisation in $\underline{C}^{\underline{C}}$:

$\alpha = S \xrightarrow{\epsilon} U \xrightarrow{\nu} T.$

First, it can be verified that this functor preserves extremal-epis and monos. We now show that it is the endofunctor of a theory and that ϵ and ν are then theory morphisms.

Define $\eta'' : I \overset{\bullet}{\to} U$ by $\eta'' = \eta.\epsilon$ and μ'' as the unique fill-in (using the fact that S, T, U preserve extremal-epis and monos):

$$\begin{tikzcd}
S^2 \arrow[r, "\epsilon^2"] \arrow[d, "\mu"'] & U^2 \arrow[r, "\nu^2"] \arrow[d, dashed, "\mu''"'] & T^2 \arrow[d, "\mu'"] \\
S \arrow[r, "\epsilon"'] & U \arrow[r, "\nu"'] & T
\end{tikzcd}$$

It can readily be verified that (U, η'', μ'') really is a theory. This completes our demonstration. $\square$

Notes

Factorisation in a category and factorisation systems were described in [Mac Lane 1948].

The connection between factorisation of theory morphisms and data abstraction was noted in [Goguen and Burstall 1978].

CHAPTER ELEVEN

MANY-SORTED THEORIES AND MORE

The monadic theories which we have investigated so far are adequate to handle only theories with a single sort. It is true that we could use the cross-product of categories to describe many-sorted (or heterogeneous) theories - that is, we coerce the sorts into one sort by taking the product of the sorts. The operations then take such product sorts as arguments. Besides being clumsy, this method fails to generalise sufficiently for our purposes. It cannot, for instance, handle the, so called, order-sorted theories. We will, in this chapter, show how to generalise monadic theories to handle these cases. This categorical formulation of order-sorted theories means that we should be able to translate work in order-sorted theories into standard category theory. As an example of this, we show that the 'fill-out' operation of [Goguen 1978] arises from an adjunction.

First we discuss several extensions of the many-sorted or heterogeneous theories.

ORDER-SORTED AND CATEGORY-SORTED THEORIES

The idea of "coercions" has proved useful in programming whenever one sort is to be thought of as included in another sort. For instance, we may wish that integers were, in some sense, included in the reals. This facility is available in many programming languages and is allied to the notion of "overloading" of operations. Overloading is simply the use of the same operation name for operations with different functionalities. This is necessary if coercions are to be introduced. For instance, an addition on reals should, under the above coercion, become an addition on integers. All this is formalised in the paper [Goguen 1978]. Notice that many sorted theories give us no means for describing such relations between the sorts. What is required is an ordering upon the sorts such that if there is a coercion of 'int' into 'real' then int $\leq$ real. This assumes that there is only one coercion of 'int' into 'real' and that there is no coercion of 'real' into 'int'. With this example, this is evidently the case, but several authors have advanced [Reynolds

1978] the more general idea of a category of sorts which allows more than one coercion between sorts and coercions in both directions between sorts.

So far we have made no mention of "errors" or "exception cases" but for a correct formulation of many data types within universal algebra we must introduce means of handling "errors". For instance, the standard presentation of 'stack' is:

```
procedure Stack(X : Triv) =
  let A =
     data  sorts stack
           opns  nil : stack
                 push : element,stack -> stack
  in
     enrich A by
           opns  pop : stack -> stack
                 top : stack -> element
           eqns  pop(push(n,s)) = s
                 top(push(n,s)) = n
                                         end
```

Of course 'pop(nil)' is not a stack in the normal sense. We cannot push onto this stack. Goguen [1978] has shown that, by introducing "error-sorts", we may present this example correctly, and furthermore that this may be formulated within these order-sorted theories. The order on the sorts arises from the fact that we would like some relation between, say, 'error-stack' and 'ok-stack', both being included in a sort 'stack'. Further details and an algebraic formulation may be found in the paper just alluded to.

An Aside

This does not end the discussion on handling errors in data types. For one thing, the idea of introducing error-sorts and subsuming these theories with errors within order-sorted theories leads to rather unwieldy presentations of data types and the interpretation of the error-sorts and error-operations is often not at all obvious. Moreover, whilst we often need many different errors of different sorts so that we can handle error messages and different types of error recovery, often we would wish to eschew the mentioning of errors altogether. Yet we would still need the presentation of 'stack' above to be correct - which it evidently is not. A case in question is the synthesis of programs from specifications. We would hardly expect errors to be needed in programs synthesised correctly from specifications.

We seem to need "errors" because our language for describing the domains of our operations is too limited (essentially to the product of sorts and, in higher-order theories, to their exponent). For instance in the case of 'stack' we would wish to say that the domain of 'pop' was restricted to non-empty stacks - those that satisfy the sentence:

$\exists$ n: element, $\exists$ s: stack : x = push(n,s)   (1)

For another example let us look at a presentation of (small) categories:

```
sorts object, morphism

opns  source, target : morphism -> object
      identity : object -> morphism
      compose : morphism, morphism -> morphism

eqns  source(identity(n)) = n
      target(identity(n)) = n
      source(compose(m1,m2)) = source(m1)
      target(compose(m1,m2)) = target(m2)
      compose(identity(source(m1)),m1) = m1
      compose(m1,identity(target(m1))) = m1
      compose(compose(m1,m2),m3) = compose(m1,compose(m2,m3))
```

Again this is not correct - it does not describe what we want. Composition is not defined on all pairs of morphisms, only on those pairs (m1,m2) satisfying the equation:

target(m1) = source(m2)   (2)

We give a brief outline of what might be done in this direction. Notice first that both the restricted domains above, those defined by sentences (1) and (2) have a categorical interpretation. If, for the moment, we work with Lawvere theories this will become clear. In the case of equation (1), 'push' will be a morphism in the category and the equation then says that we want a unique factorisation of this morphism (exactly which sort of factorisation may depend upon context). We may denote the sort obtained from the factorisation (intuitively, the sort of non-empty stacks) by 'push' and write:

    pop : push -> stack.

Likewise for the case of small categories, equation (2) denotes a pullback:

$$\begin{tikzcd}
\text{target \# source} \arrow[r, dashed] \arrow[d, dashed] & \text{morphism} \arrow[d, "\text{source}"] \\
\text{morphism} \arrow[r, "\text{target}"'] & \text{object}
\end{tikzcd}$$

We can call the new sort, using standard category theory notation, 'target # source', and write

    compose : target # source -> morphism.

It should be possible to extend the usual product (or coproduct) in Lawvere theories to these more general domain constructors. Likewise monadic theories should encompass theories with operations of this form. We do not intend that we should be able to define every domain (though we could considerably extend the domain building operations above) but we may not need to. For instance we should not ask for a domain of acyclic graphs for a computation of transitive closure. There is a difference between the failure of a program to terminate and the undefinedness of, say, 'pop(nil)'. The former is correct - acyclic graphs have an infinity of paths - the latter has no meaning.

However, it should be noted that this extension of algebraic theories does not include the theory of fields, unlike the error-theories of [Goguen 1978]. Fields are not well-behaved (for example, they are not monadic and free fields do not, in general, exist) whereas stacks and small-categories are well-behaved (at least in the sense given). The behaviour of theories with partially defined operations is discussed in [Broy,Wirsing 1980].

Notice that when we define restricted domains for operations, these restrictions are in terms of previously given operations. For instance, the declaration of "pop" is in terms of "push". We assume that "push" is declared before "pop". We are thus thinking of a hierarchy of operations, each defined in terms of those 'below' it. Freyd [1972] discusses theories of this sort in the case when the domains are given by equations, calling these theories 'essentially algebraic'.

MANY-SORTED THEORIES AND ORDER-SORTED THEORIES AS MONADIC THEORIES

We now show how to extend monadic theories to include these many-sorted, order-sorted and even category-sorted theories. In the Lawvere approach to theories this extension from single-sorted theories to many-sorted theories can be accomplished by choosing a different category of "arities" as described in [Wagner,Wright,Goguen,Thatcher 1978].

We deal here only with category-sorted theories as the others are special cases. Let $\underline{S}$ be a category of categories and S an object of $\underline{S}$. We are to think of $\underline{S}$ as the category of "sort sets" and S as a "sort set", i.e. a category - possibly the discrete category.

Let $\underline{C}$ be any category (we usually think of $\underline{C}$ as the category of sets or possibly continuous partial orders), and F a functor:

$F : S \to \underline{C}$.

An S-sorted theory will be a theory on the category of these functors, $\underline{C}^S$. So far, so good. What about theory morphisms? Theory morphisms have to include now not only a map on operations which preserves the equations (which, as we know, is handled by a natural transformation between endofunctors) but also a map between the sorts. Theory morphisms may change the sorts. Thus, we see that theory morphisms may map theories on one base category to theories on a different base category. This is where an extension of the definition of monadic theories is required.

Notice that, with the categories $\underline{S}$ and $\underline{C}$, we can form a new category whose objects are categories of the form $\underline{C}^S$, for S an object of $\underline{S}$, and whose morphisms are all functors between these categories. Moreover, a map on sorts - a morphism in $\underline{S}$ - of the form $h : S \to S'$, gives rise to a functor,

$\underline{C}^h : \underline{C}^{S'} \to \underline{C}^S$, from the category of S'-sorted $\underline{C}$-objects to the category of S-sorted $\underline{C}$-objects.

What is a category of theories now going to look like? Theories will still be monads on some category of the form $\underline{C}^S$. On the other hand, theory morphisms from $\underline{T} = (T, \eta, \mu)$ on $\underline{C}^S$ to $\underline{T}' = (T', \eta', \mu')$ on $\underline{C}^{S'}$ will be pairs (h, α) where $h: S \to S'$ and $\alpha : \underline{C}^h T \to T'\underline{C}^h$ is a natural transformation. The diagram below may help:

$$\begin{tikzcd}
\underline{C}^{S'} \arrow[r, "\underline{C}^h"] \arrow[d, "T'"'] & \underline{C}^S \arrow[d, "T"] \arrow[dl, Rightarrow, "\alpha"] \\
\underline{C}^{S'} \arrow[r, "\underline{C}^h"'] & \underline{C}^S
\end{tikzcd}$$

The identity morphism on $\underline{T}$ is $(I_S, i_{\underline{C}}S)$ and compositions of morphisms is given by, if $\underline{T}'' = (T'', \eta'', \bar{\mu}'')$ is a theory on $\underline{C}^{S''}$ and $(h', \alpha') : \underline{T}' \to \underline{T}''$ then

$$(h, \alpha).(h', \alpha') = (hh', \underline{C}^{h'}\alpha \cdot \alpha'\underline{C}^h)$$

defines the composition.

We call this category $\underline{\text{Theory}}(\underline{S},\underline{C})$ and by 'forgetting' the composition in theories we get, by the same construction, a category of signatures, $\underline{\text{Signature}}(\underline{S},\underline{C})$. Again, there is a forgetful functor,

$$U : \underline{\text{Theory}}(\underline{S},\underline{C}) \to \underline{\text{Signature}}(\underline{S},\underline{C}).$$

More notation: S-$\underline{\text{Sig}}(\underline{C})$ is the category of S-sorted signatures for an S in $\underline{S}$, likewise for theories.

This looks rather awkward but it can be cast into an elegant general form as these theory morphisms behave rather like natural transformations. We can define a 'horizontal' composition as well as the above 'vertical' composition. See Appendix Two for details.

PROPERTIES OF MANY-SORTED AND ORDER-SORTED MONADIC THEORIES

We now look at properties of many-sorted and order-sorted theories in this monadic setting. We will show that the previous results, which we have proved in the case of the single-sorted theories, extend to these many-sorted, order-sorted and category-sorted theories.

The idea is to choose a suitable category in which the construction for the single-sorted case can be carried out and then show that the construction is universal in the larger category of theories, Theory($\underline{S}$,$\underline{C}$). We use the notation of the previous section and sketch briefly what the constructions look like in this category of theories.

For the free theory on an S-sorted signature with respect to the functor, U, we notice that the construction takes place entirely within the category $\underline{C}^S$, so, provided that $\underline{C}$ has ω-colimits and that the endofunctor of the signature preserves these colimits, the construction goes through as before. To establish the universality of the theory so constructed, we need that for each h : S -> S', the functor $\underline{C}^h : \underline{C}^{S'} \to \underline{C}^S$ preserves ω-colimits. This is a corollary of a general result concerning colimits in functor categories (see [Herrlich,Strecker 1973]). Now the construction of the mediating morphism goes through as before replacing, of course, single-sorted theory morphisms by these more general morphisms.

What about finite cocompleteness of the category Theory*($\underline{S}$,$\underline{C}$), the full subcategory of Theory($\underline{S}$,$\underline{C}$) consisting of those theories whose endofunctors are cocontinuous? If we assume that $\underline{C}$ is (arbitrarily) cocomplete and that $\underline{S}$ is finitely cocomplete (as a subcategory of the category of categories) then Theory*($\underline{S}$,$\underline{C}$) is finitely cocomplete.

The initial object in Theory*($\underline{S}$,$\underline{C}$) has no sorts - it has the initial "sort-set", Φ. There is a unique functor H : $\Phi \to \underline{C}$ and the initial theory is then (I,i,i) where I is the identity functor on the one object category containing H, and i is the identity natural transformation from I to I.

To construct the coproduct of a pair of objects in $\underline{\text{Theory}}^*(\underline{S},\underline{C})$ let $\underline{T}_1 = (T_1, \eta_1, \mu_1)$ be an S_1-sorted theory and $\underline{T}_2 = (T_2, \eta_2, \mu_2)$ be an S_2-sorted theory (S_1 and S_2 being objects of $\underline{S}$). Let S_1+S_2 be the coproduct of the categories S_1 and S_2. By universality of the coproduct a functor $H : S_1+S_2 \to \underline{C}$ splits into functors $H_1 : S_1 \to \underline{C}$ and $H_2 : S_2 \to \underline{C}$. Now consider the functors, H_1T_1 and H_2, these give a functor $H' : S_1+S_2 \to \underline{C}$ and then this passage from H to H' is functorial, the functor being the endofunctor of a theory $\underline{T}'_1$ on $\underline{C}^{S_1+S_2}$. Likewise construct a theory $\underline{T}'_2$. Then the coproduct of $\underline{T}_1$ and $\underline{T}_2$ is the coproduct of $\underline{T}'_1$ and $\underline{T}'_2$ calculated in the category $\underline{C}^{S_1+S_2}$. The universality of this construction can readily be checked.

A similar treatment will provide the coequaliser of a parallel pair of theory morphisms in $\underline{\text{Theory}}^*(\underline{S},\underline{C})$, using the coequaliser of "sort-sets" in $\underline{S}$. Thus $\underline{\text{Theory}}^*(\underline{S},\underline{C})$ is finitely cocomplete.

The other constructions that we have given for the single-sorted monadic theories should generalise to the category, $\underline{\text{Theory}}^*(\underline{S},\underline{C})$, for suitable $\underline{S}$ and $\underline{C}$. This has yet to be checked.

FREE SIGNATURES AND KAN EXTENSIONS

Part of the aim of this work is to bring the universal concepts of category theory to bear upon our understanding of programming. This is not anything new. For some years now the descriptive power of initiality and adjunctions and of colimits has been recognised. However adjunctions and colimits are not the only possible universal concepts in category theory. Whilst more such concepts may be formulated it appears that they all are 'equivalent' to each other. For instance, the definition of a colimit can be cast into a special case of that of an adjunction and adjunctions can become colimits via a formal existence theorem for adjunctions.

We introduce here another, less familiar, universal concept - that of a Kan extension. We need this new universal concept to describe and extend an operation on signatures to be found in [Goguen 1978]. In fact, in investigating this, I inadvertently "rediscovered" the important existence theorem for Kan extensions (page 201).

Signatures and Order-sorted Theories

What do signatures for order-sorted theories look like? To incorporate order-sorted theories into universal algebra an arbitrary set of operations will no longer serve as a signature. We need to consider how the order upon the sorts is reflected in the signature.

For instance, if {int, real} is our set of sorts and there is a coercion 'int -> real' , and if we had an operation in our signature:

$* : \text{real} \to \text{int}$

we would expect also the operations

$*_1 : \text{int} \to \text{int}$

$*_2 : \text{real} \to \text{real}$

$*_3 : \text{int} \to \text{real}$

which arise from the coercion by restricting the domain or enlarging the range of *.

It should be clear however that given an arbitrary set of operations and their arities and an order upon the sorts we can introduce extra operations as above to 'fill-out' the signature to an order-sorted signature. This filling-out operation is introduced in [Goguen 1978].

We show that this operation arises from a Kan extension and thereby give an interpretation of 'filling-out' for category-sorted signatures as well. In fact, there is a sense in which the fill-out of a signature is free upon the original signature. We first define the rather unfamiliar idea of a Kan extension then show how it relates to the construction of free signatures.

Kan extensions

Consider the following diagram of categories and functors

$$\begin{tikzcd}
\underline{A} \arrow[r, "F"] & \underline{B} \arrow[d, "G"] \\
 & \underline{C}
\end{tikzcd}$$

There is an H : A -> $\underline{C}$ such that:

$$\begin{tikzcd}
\underline{A} \arrow[r, "F"] \arrow[dr, dashed, "H"'] & \underline{B} \arrow[d, "G"] \\
 & \underline{C}
\end{tikzcd}$$

commutes. H is given by the composition of F and G. Now we are looking for a way of inverting this. That is, given a diagram:

$$\begin{tikzcd}
\underline{A} \arrow[r, "F"] \arrow[dr, "H"'] & \underline{B} \\
 & \underline{C}
\end{tikzcd}$$

we are looking for a functor, G, such that:

$$\begin{tikzcd} \underline{A} \arrow[r, "F"] \arrow[dr, "H"'] \arrow[r, phantom, "(1)", shift right=20] & \underline{B} \arrow[d, dashed, "G"] \\ & \underline{C} \end{tikzcd}$$

commutes. In general no such G exists. Suppose, however, that we weaken our requirement. Instead of insisting that (1) commutes we merely ask that there is a natural transformation α : H -> FG. In general, there are many such G's. Now suppose that α is universal in the following sense:

For any other G' : $\underline{B}$ -> $\underline{C}$ and α': H -> FG' there is a unique θ : G -> G' such that

$$\begin{tikzcd} H \arrow[r, "\alpha"] \arrow[dr, "\alpha'"'] & FG \arrow[d, dashed, "F\theta"] \\ & FG' \end{tikzcd}$$

commutes.

Then we say that G is the left Kan extension of H along F. Right Kan extensions arise if we insist on the requirement of a natural transformation α : FG -> H with a similar universal property.

Now suppose that for a fixed F, every H has a left Kan extension then we can cast the definition into that of an adjunction. Indeed the passage from an H to such a G is left adjoint to the "composition" functor,

$\underline{C}^F$: $\underline{C}^{\underline{B}}$ -> $\underline{C}^{\underline{A}}$ which is defined by:

$\underline{C}^F$(G) = FG.

We denote this left adjoint to $\underline{C}^F$, if it exists, by,

$\underline{C}_F : \underline{C}^{\underline{A}} \rightarrow \underline{C}^{\underline{B}}$.

Conversely an adjunction can be considered as a special case of a Kan extension. Further details of this and the interrelation of other universal concepts may be found in [Mac Lane 1971].

The existence theorem for Kan extensions, already alluded to, goes as follows.

Theorem (Existence of left Kan extensions)

If $\underline{C}$ is cocomplete then for any $F : \underline{A} \rightarrow \underline{B}$, $\underline{C}^F$ has a left adjoint.

Sketch of Proof

Given any $H : \underline{A} \rightarrow \underline{C}$ we construct a $G : \underline{B} \rightarrow \underline{C}$ with the given universal property.

Let b be an object of $\underline{B}$, $(\underline{B},b)$ be the comma category of morphisms into b and

$P^b : (\underline{B},b) \rightarrow \underline{B}$

the projection functor.

We define a diagram in $\underline{A}$ as the functor E^b given by the following pullback square in the category of categories.

$$\begin{tikzcd}
\underline{J}^b \arrow[r, dashed] \arrow[d, dashed, "E^b"'] & (\underline{B},b) \arrow[d, "P^b"] \\
\underline{A} \arrow[r, "F"'] & \underline{B}
\end{tikzcd}$$

Pullbacks of categories exist.

Now consider, for the given $H : \underline{A} \rightarrow \underline{C}$, the diagram in $\underline{C}$ given by the composition of functors:

$$\begin{tikzcd}
\underline{J}^b \arrow[r, "E^b"'] & \underline{A} \arrow[r, "H"'] & \underline{C}
\end{tikzcd}$$

The colimit of this diagram gives rise to a functor, G, which takes b onto the apex of the colimiting cone. I.e. there is a $G : \underline{B} \rightarrow \underline{C}$ such that the colimit of (2) is given by

$$j\zeta^b : jH^b \rightarrow bG \text{ for } j \text{ in } \underline{J}^t.$$

The morphism part of G arises from the universality of the colimit. This G is the free object on H with respect to $\underline{C}^F$. $\square$

We now state a general lemma about adjunctions which we shall need in the next section.

Lemma

Let $(K, L, \nu, \epsilon) : \underline{A} \rightarrow \underline{B}$ and $(T, U, \eta, \pi) : \underline{C} \rightarrow \underline{D}$ be any two adjunctions. Then the following functor

$$L \_ T : \underline{C}^{\underline{A}} \rightarrow \underline{D}^{\underline{B}} \text{ is left adjoint to}$$

$$K \_ L : \underline{D}^{\underline{B}} \rightarrow \underline{C}^{\underline{A}}$$

The meaning of these functors should be obvious : $L \_ T$ takes any $H : \underline{A} \rightarrow \underline{C}$ into $LHT : \underline{B} \rightarrow \underline{D}$.

Proof omitted.

Free Signatures

Here we show the connection between the rather obvious "fill out" operation (referred to above) on signatures (which is, for finite signatures, computable) and the rather unfamiliar Kan extensions.

Let $\underline{S}$ be a category of sorts and $h : S \rightarrow S'$ a morphism in the category. Let $\underline{C}$ be a cocomplete category and $\underline{C}^{\underline{S}}$ the category of $\underline{S}$-sorted $\underline{C}$-objects. Then

$$\underline{C}^h : \underline{C}^{S'} \rightarrow \underline{C}^S$$

has a left adjoint. This is simply the existence theorem for the left Kan extension. Let us denote this free functor by

$$\underline{C}_h : \underline{C}^S \rightarrow \underline{C}^{S'}$$

Notice that the $h : S \rightarrow S'$ induces a "forgetful" functor on signatures,

$$U : S'\text{-}\underline{\text{Sig}}(\underline{C}) \rightarrow S\text{-}\underline{\text{Sig}}(\underline{C}) \text{ given by}$$

$(\Sigma', \sigma')U = (\underline{C}_h \Sigma' \underline{C}^h, \underline{C}_h \sigma' \underline{C}^h)$ on objects and $\alpha U = \underline{C}_h \alpha \underline{C}^h$ on morphisms. For instance, any order-sorted signature is a ordinary (set-sorted) signature.

The above proposition about adjunctions together with the existence of a left adjoint for Kan extensions now give a left adjoint to U:

$T : S\text{-}\underline{Sig}(\underline{C}) \to S'\text{-}\underline{Sig}(\underline{C})$

given by $(\Sigma, \sigma)T = (\underline{C}^h \Sigma \underline{C}_h, \underline{C}^h \sigma \underline{C}_h)$ on objects and $\alpha T = \underline{C}^h \alpha \underline{C}_h$ on morphisms. Thus we have constructed an adjunction on signatures.

Now suppose that $\underline{C} = \underline{Set}$ and that S' is an ordered set of sorts and S its underlying set (forgetting the order). Then we have an inclusion

$h : S \to S'$.

This induces a functor $\underline{C}^h$, which takes an S'-sorted set to an S-sorted set by forgetting the order among the sets of variables. By the Kan extension existence theorem, there is a left adjoint to $\underline{C}^h$. This fills out an S-sorted set according to the ordering on S'. For example, let S' = {int, real} with int < real. Then S = {int, real} with no order upon it. Each S'-sorted set is automatically an S-sorted set by forgetting the order 'int < real'. This is $\underline{C}^h$.

Now suppose that {n,m : int, x,y : real} is an S-ordered set. It can be filled out to an S'-ordered set by including all the variables of sort "int" into those of sort "real". That is, the left adjoint of $\underline{C}^h$, $\underline{C}_h$, takes

$\underline{C}_h$: {n,m : int, x,y : real} -> {n,m : int, n,m,x,y : real}

The adjunction between signatures arising from h : S -> S' gives exactly the same operation but between sorted sets of terms. That is, it is the "filling out" operation of [Goguen 1978].

Notes

The ADJ group gave a mathematical formulation of heterogeneous (many-sorted) theories [Goguen, Thatcher, Wagner and Wright 1975]. The idea of extending these theories to order-sorted and category-sorted theories came from several authors (e.g. Wadge (unpublished letter) and

Reynolds (1978)). Our presentation is an attempt to capture within the framework of monadic theories the work of [Goguen 1978].

CHAPTER TWELVE
CONCLUSION

The programming of category theory was an experiment both in translating mathematical proofs (relevant to programming theory) into code and using high level functionals for 'combinatorial' programming (as advocated by Backus [1978]).

We have shown how constructive proofs in category theory can be encoded as programs and the routines arising from these proofs can be used to construct a program specifically concerned with category theory. What we have not shown is the usefulness of these routines for general programming problems. This is the task that we set ourselves in the attempt to systematically develop programs from specifications using category theoretic techniques. Unlike Backus' work there is a mathematical connection between our high level functionals and specifications of programs. This may possibly indicate that these functionals have the 'correct' degree of generality for general programming tasks. Evidence for this conjecture can arise both theoretically (the connection between structure in the category of algebras of a specification (e.g. colimits) and programs which implement the specification) and through experience with translating programming tasks into a suitable categorical framework. For example, can we develop algorithms for sorting a list of totally-ordered objects into an ordered list (e.g. quicksort, bubblesort etc.)? Possibly we can. It should be said that this approach to program development was an unexpected bonus of our exercise in programming category theory.

We gather together some speculations more or less loosely based on our experience with categorical programming. These can be seen as possible directions for future work.

1. The conjecture in the paragraph above is correct.

2. Constructions of colimits of algebras (e.g. those of Linton (1969) and Adamek and Koubek (1980)) are of some importance to programming. Unlike limits of algebras

(which are created by carriers) colimits of algebras require elaborate and non-trivial constructions. We have seen that co-continuous functors may be computed by colimit recursion as long as a construction of colimits of algebras of the resultant data type is available.

3. Adding the power of universal contraints and of partially defined operations (as described in the text) to (possibly parameterised) equational specifications gives an adequate and natural means of program description which draws specification and implementation closer. Examples of the unorthodox use of universal constraints may be found in chapter five. Theoretical considerations show us that universal constraints can model unbounded existential quantifiers. The vague claim of adequacy could be substantiated by examining cases where equational specification of itself proved inadequate (e.g. [Majster 1979]).

4. Proving the correctness of categorical programs is essentially a task of proving theorems in category theory. We have looked at proof rules for colimits (chapter seven) - there are other such rules (for instance, associated with monos and epis). Could these rules be automated and along with a theorem-prover give a semi-automatic

   program-proving system for categorical programming? It may be of some interest to see a specification (in CLEAR) of the properties of such a categorical program - for instance that of a colimit program:

```
constant Category =
        sorts object, morphism
        opns  domain : morphism -> object
              range :  morphism -> object
              identity : object -> morphism
              compose : range # domain -> morphism
        eqns
              domain(identity(o)) = o
              range(identity(o)) = o
              domain(compose(m,n)) = domain(m)
              range(compose(m,n)) = range(n)
              compose(identity(o),n) = n
              compose(m,identity(o)) = m
              compose(compose(l,m),n) =
                      compose(l,compose(m,n))
                                                end

constant Graph =
        sorts node, edge
        opns  source : edge -> node
              target : edge -> node
                                 end

procedure Diagram(C : Category) =
    enrich Graph by
        opns obj_map : node -> object
             mor_map : edge -> morphism
        eqns
             domain(mor_map(e)) =
                        obj_map(source(e))
             range(mor_map(e))  =
                        obj_map(target(e))
                                       end

procedure Colimit(C : Category) =
  enrich Diagram(C) by
    data opns apex : object
              sides : node -> morphism
         eqns domain(sides(n)) = obj_map(n)
              range(sides(n)) = apex
              compose(mor_map(e),sides(target(e)))
                        = sides(source(e))
                                             end
```

Notice how the universality of the colimit-cone has become a data constraint.

Turning now to the constructions concerning monadic theories, we have shown that monadic theories are sufficiently well-behaved from the point of view of program specification. However the ad hoc nature of the constructions is unsatisfactory. In the case of the free theory construction we now understand the place of ω-colimits in the construction of free theories, initial algebras etc. Is there a similar understanding of the place of the other constructions? At several points in the text we have noted that the constructions resemble those of other authors. Is there a general synthesis - a few general constructions which can be used as (almost) all-purpose tools for constructing universally defined objects in certain categories? If such a synthesis exists, how does it impinge upon programming?

APPENDIX ONE

COLIMITS IN FUNCTOR CATEGORIES

We show how to program the lifting of colimits to functor categories. The relevant theorem is:

<u>Theorem</u> (Colimits in Functor Categories.)

If $\underline{B}$ is (finitely) cocomplete then so is $\underline{B}^{\underline{A}}$.

<u>Proof</u>

Consult [Schubert 1972].

The computation of this lifting of colimits goes as follows.

The application of a diagram and a diagram morphism (and a cone) of functors to an object of the source category give the following functions.

```
    dec applydo : Diagram( Functor(o,m,o1,m1),
                           Nat_transform(o,m,o1,m1)) # o
                 -> Diagram(o1,m1)
    dec applydm :
            Cat(o,m) ->
        (Diagram(Functor(o,m,o1,m1),Nat_transform(o,m,o1,m1)) # m
                  -> Diagram_Mor(o1,m1))
    dec applyco :
            Cat(o1,m1) ->
        (Cone(Functor(o,m,o1,m1),Nat_transform(o,m,o1,m1)) # o
                  -> Cone(o1,m1) )
    --- applydo(diagram(g,fo,fm),a) <=
              diagram( g,
                   (lambda n => fo(n) ofo a),
                   (lambda e => fm(e) of a) )
    --- applydm(cat(s,t,_,_)) <=
          (lambda d & diagram(_,fo,fm), m =>
              diagram_mor( applydo(d,s(m)),
                           ident,
                           ident,
                           (lambda n => fo(n) ofm m),
                           applydo(d,t(m))) )
    --- applyco(B) <=
            lambda Fcone,s =>
               let ds == applydo(base(Fcone),s) in
                    cone(B)
                        ( ds,
                          (lambda n => sides(Fcone)(n) of s),
                          apex(Fcone) ofo s)
```

The colimiting object, a functor, of a diagram of functors, is computed 'pointwise':

```
dec colimit_obj : Cat(o,m) # Colimit_Cat(o1,m1) ->
        (Diagram(Functor(o,m,o1,m1),Nat_transform(o,m,o1,m1))
                    -> Functor(o,m,o1,m1))

--- colimit_obj(A & cat(s,t,_,_),cB & colimit_cat(B,_)) <=
      (lambda D  =>
           let dcatB == cat_of_diagrams(B) in
           let dcat ==
                cat_of_diagrams(cat_of_functors(A,B)) in
      let omap ==                ! object part of colimit functor
        (lambda a =>
           colimit_object(cB)(applydo(D,a))) in
      let mmap ==                ! morphism part of colimit functor
        (lambda m =>
           let suniv ==
                universal_part(cB)(applydo(D,s(m))) in
           let tcone == colimit_cone(cB)(applydo(D,t(m))) in
           let pretendcone ==
                left_compose(dcatB,dcatB,I(dcatB))
                            (applydm(A)(D,m),tcone) in
         apex_morphism(suniv(pretendcone)) ) in
    functor(omap,mmap) )                ! colimit functor
```

Here the lambda-expressions in objects and morphisms of the source category, A, are the 'pointwise' computations. The whole of the colimit - the colimiting cone and the universal part - then arise as follows:

```
    dec lift_colimit : Cat(o,m)#Colimit_Cat(o1,m1) ->
           Colimit(Functor(o,m,o1,m1),Nat_transform(o,m,o1,m1))

    --- lift_colimit(A, cB & colimit_cat(B,_) ) <=
          (lambda D & diagram(_,Fo,Fm) =>
               let dcatB == cat_of_diagrams(B) in
               let fcat == cat_of_functors(A,B) in
       let F == colimit_obj(A,cB)(D)        in
       let gamma ==
             (lambda a =>
                sides(colimit_cone(cB)(applydo(D,a))) ) in
       let colimcone ==
             cone(fcat)( D,
                         (lambda n =>
                            nat_transform( Fo(n),
                                           (lambda a =>
                                               gamma(a)(n)),
                                           F )),
                         F ) in
       let univ ==
             (lambda a =>
                 universal_part(cB)(applydo(D,a)) ) in
       let universal ==
             (lambda pcone =>
                cone_mor(fcat)
                        (colimcone,
                         nat_transform(F,
                                       (lambda a =>
                                           apex_morphism(
                                              univ(a)
                                              (applyco(B)
                                              (pcone,a)))),
                                           apex(pcone)),
                         pcone) ) in
       (colimcone,universal) )
```

An inspection of the steps of an elementary proof of the theorem will make this text comprehensible.

Finally, as in the case of comma categories, we can use this 'lifting' routine to define a colimit category by declaring:

```
    dec colimit_cat_of_functors :
            (Cat(o,m)#Colimit_Cat(o1,m1)) ->
        Colimit_Cat(Functor(o,m,o1,m1),Nat_transform(o,m,o1,m1))

    --- colimit_cat_of_functors(A,cB & colimit_cat(B,_)) <=
            colimit_cat(cat_of_functors(A,B),lift_colimit(A,cB))
```

We could use this to compute colimits of diagrams for instance, or of signatures in a monadic form (page 123).

APPENDIX TWO

GENERALISED TRANSFORMATIONS AND A GENERAL CATEGORY OF THEORIES

We introduce here a generalisation of natural transformations of which the morphisms of many-sorted theories are an example.

Consider the (not necessarily commuting) square of categories and functors:

$$\begin{tikzcd}
\underline{A}' \arrow[r, "F"] \arrow[d, "S'"'] & \underline{A} \arrow[d, "S"] \\
\underline{B}' \arrow[r, "G"'] & \underline{B}
\end{tikzcd}$$

We say that α is an F,G -transformation from S to S', denoted by

α : S $\overset{\cdot}{\Rightarrow}$ S' (with F and G understood)

if α is a natural transformation, α : FS $\overset{\cdot}{\to}$ S'G.

There are two compositions of these general transformations. Suppose we have

α : S $\overset{\cdot}{\Rightarrow}$ S' an F,G -transformation

β : S' $\overset{\cdot}{\Rightarrow}$ S" an F',G'-transformation

γ : T $\overset{\cdot}{\Rightarrow}$ T' a G, H -transformation

with the following (not necessarily commuting) diagrams:

$$\begin{tikzcd}
\underline{A}'' \arrow[r, "F'"] \arrow[d, "S''"'] & \underline{A}' \arrow[r, "F"] \arrow[d, "S'"'] & \underline{A} \arrow[d, "S"] \\
\underline{B}'' \arrow[r, "G'"'] & \underline{B}' \arrow[r, "G"'] & \underline{B}
\end{tikzcd}
\qquad
\begin{tikzcd}
\underline{A}' \arrow[r, "F"] \arrow[d, "S'"'] & \underline{A} \arrow[d, "S"] \\
\underline{B}' \arrow[r, "G"] \arrow[d, "T'"'] & \underline{B} \arrow[d, "T"] \\
\underline{C}' \arrow[r, "H"'] & \underline{C}
\end{tikzcd}$$

The vertical composition $\alpha.\beta : S \overset{\bullet}{\Rightarrow} S''$, an F'F,G'G-transformation, is defined by $\alpha.\beta = F'\alpha.\beta G$ where the composition on the right-hand side is that of natural transformations.

The horizontal composition $\alpha \circ \gamma : ST \overset{\bullet}{\Rightarrow} S'T'$, an F,H -transformation, is defined by $\alpha \circ \gamma = \alpha T.S'\gamma$, again the right-hand side is the composition of natural transformations. Notice that this composition is no longer given as the diagonal of a commuting square - the functionalities are wrong.

These generalised transformations, with their compositions, form a double category (in the sense of [Ehresmann 1965]) in that the interchange law holds -

$$(\alpha.\beta)\circ(\alpha'.\beta') = (\alpha\circ\alpha').(\beta\circ\beta')$$

whenever either side is defined. It is not a two-category - the identities do not behave correctly.

<u>A Generalised Category of Theories</u>

Let $\underline{V}$ be a category whose objects are categories and whose morphisms are functors. The objects of $\underline{V}$ are to be the base categories of theories.

Suppose that $\underline{C}$ and $\underline{C}'$ are objects of $\underline{V}$ and that $\underline{T} = (T,\eta,\mu)$ and $\underline{T}' = (T',\eta',\mu')$ are theories on $\underline{C}$ and $\underline{C}'$ respectively. Let $H : C' \to \underline{C}$ be a functor in $\underline{V}$, then the square of categories and functors above reduces to:

$$\begin{tikzcd}
\underline{C}' \arrow[r, "H"] \arrow[loop below, "T'"'] & \underline{C} \arrow[loop below, "T"]
\end{tikzcd}$$

We now introduce the category <u>Theory</u>($\underline{V}$) - a generalised category of theories. Its objects are to be monadic theories on the objects (categories) in $\underline{V}$. Its morphisms are given by the following: Let $\underline{C}$

and $\underline{C}$' be categories in $\underline{V}$, a theory morphism (H,α) : $\underline{T}$ -> $\underline{T}$', where $\underline{T}$ and $\underline{T}$' are theories as above, is a functor in $\underline{V}$, H : $\underline{C}$' -> $\underline{C}$ together with an H,H-transformation (which we abbreviate to an H-transformation), $\alpha : T \overset{\bullet}{=}> T'$ where satisfies the following two equations:

$$\eta.\alpha = i.\eta' \text{ and } (\alpha o \alpha).\mu' = \mu.\alpha$$

Here the equations are of generalised transformations and their compositions. 'i' is the identity H-transformation and we have made the implicit coercion of natural transformations to I-transformations, I being the identity functor on a category. Notice how the form of the equations has been preserved form the case of ordinary monadic theories and their morphisms.

Composition of generalised theory morphisms is given by the generalised vertical composition. This completes our construction of the category Theory($\underline{V}$).

By 'forgetting' the composition, μ, in theories we get a category, Signature($\underline{V}$), whose objects are signatures and whose morphisms are generalised signature morphisms.

Furthermore, just as before, there is a 'forgetful' functor, U, from theories to signatures given by: (T,η,μ)U = (T,η) on objects and the identity on morphisms.

In this general setting, provided there are suitable restrictions on the category, $\underline{V}$ and the theories, the work which we have done with monads should generalise.

APPENDIX THREE

HOPE : AN EXPERIMENTAL APPLICATIVE LANGUAGE

An experimental applicative language called HOPE has been developed by Prof. Rod Burstall, David MacQueen and Don Sannella. The following brief informal description of the language is an extract from [Burstall, MacQueen and Sannella 1980].

Data Declarations

Conceptually, all data in HOPE is represented as terms consisting of a data constructor applied to a number of subterms, each of which in turn represents another data item. The tips of this tree are nullary data constructors or functional objects. An example is succ(succ(0)) in which succ is a unary constructor and 0 is a nullary one (i.e. a constant). Constructor functions are uninterpreted; they just construct.

A data declaration is used to introduce a new data type along with the data constructors which create new elements of the type. For example, the data declaration for natural numbers would be

```
    data Num == 0 ++ succ(Num)
```

defining a data type called Num with data constructors 0 and succ. So the elements of Num are 0, succ(0), succ(succ(0)), ...; that is, 0, 1, 2, ...

Types may be parameterised. To define a type of lists of elements where all the elements are to have the same (but arbitrary) type, we declare a type variable:

```
    typevar alpha
```

which when used in a type expression denotes any type (including second- and higher-order types). A general definition of 'list' as a recursive parametric type is now possible:

```
    data List(alpha) == nil ++ cons(alpha,List(alpha))
```

Notice that List is not a type but a unary type constructor.

There is another means of defining types. We may define a new type by a type expression in other types. For example:

```
    type Pair(alpha) == alpha # alpha
```

where # is an (inbuilt) type constructor. If n and m are of type Num then an object of type Num # Num (and so of Pair(Num)) is (n,m). Notice that a data declaration introduces a new type in terms of new type constructors whereas a type declaration as above simply gives a name to a type expression.

Expressions

The simplest expressions of HOPE are constants (i.e. data constructors and functions - the 'usual' concept of a constant is just the class of nullary functions and data constructors) and variables.

An application may be formed by simply juxtaposing two expressions:

```
    (f x) y
```

The first expression, (f x), is taken to be a function which is applied to the second, y.

It is possible to use function symbols as infix, postfix, outfix and distributed-fix if they are declared and given a precedence. For example:

```
    infix + : 8
```

HOPE has a conditional expression:

```
    a if b else c
```

Lambda expressions (denoting functions) are formed as described in the next section.

Local variables may be introduced and asssociated with values using either of the equivalent forms:

```
    e where p == g
```

or

```
    let p == g in e
```

where p is an expression formed by application of data constructors to a number of distinct variables (this is called a pattern). For example:

```
    a + b where cons(a,cons(b,l)) == f(t)
```

Upon evaluation, f(t) is expected to yield a value which matches the pattern 'cons(a,cons(b,l))'. The corresponding subterms in the value of f(t) are then bound to a, b and l. Notice that the binding to l is not used in the evaluation of the above expression. We therefore may allow patterns to have "holes" in them (denoted by an underscore) where we do not wish a matching to take place. Then the above expression would become:

```
    a + b where cons(a,cons(b,_)) == f(t)
```

(yielding the same value but with less work).

"Multilevel patterns" are also catered for: If x is a variable and P is a pattern, then x & P is also a pattern. The effect of this is that x is matched to the value matching the entire pattern P whilst variables within P are bound as before. For example, if l is a list, then the following expression:

```
    let cons(e1, t & cons(e2,_)) == l in f(e1,t,e2)
```

binds e1 to the head of l, t to the tail of l and e2 to the head of the tail of l. Evidently the above expression could be written without an & but less concisely as:

```
    let cons(e1,t) == l in
    let cons(e2,_) == t in f(e1,t,e2)
```

Note that & binds more tightly that comma (tupling) but less tightly than function application.

Defining Functions

Before a function is defined, its type must be declared. For example:

```
    dec reverse : List(alpha) -> List(alpha).
```

HOPE is a strongly typed language, and the HOPE system includes a polymorphic type checker (a modification of that in [Milner 1978]) which is able to detect all type errors at compile time. Function symbols may be overloaded - declared several times with different types. When this is done the typechecker is able to determine which function definition belongs to each instance of the function symbol.

Functions are defined by a sequence of one or more equations, where each equation specifies the function over some subset of possible argument values. This subset is described by a pattern on the left-hand side of the equation. For example:

```
    --- reverse(nil) <= nil
    --- reverse(cons(a,l)) <= reverse(l) <> cons(a,nil)
```

(the symbol <> is infix append). This defines the (top-level) reverse of a list.

The set of equations defining a function should exhaust the possibilities given in the data statement introducing the argument types. Nullary functions may also be defined:

```
    dec 2 : Num
    --- 2 <= succ(succ(0))
```

(Actually, numbers as well as lists, sets, characters and truth-values are built into the HOPE system.)

Lambda expressions are defined similarly. For example, a function to compute the conjunction of truth-values:

```
    lambda true,true => true
         | false,_   => false
         | _,false   => false
```

Notice the use of case analysis on patterns which is allowed within a lambda expression. Lambda expressions are often used in the definition of higher order functions (functionals):

```
    typevar alpha, beta, gamma

    dec compose : (alpha -> beta) # (beta -> gamma)
                          -> (alpha -> gamma)

    --- compose(f,g) <= lambda x => f(g(x))
```

HOPE has several features which we shall not explain as they do not occur in our programming. For instance, there is a simple data abstraction facility, there is lazy evaluation (for lists) and several iterators are available for sets and lists. Comments in HOPE are indicated by an exclamation mark.

APPENDIX FOUR

CLEAR : A SHORT TUTORIAL

The following outline of the CLEAR specification language is a summary of the pertinent parts of the following three papers: Burstall and Goguen [1977], [1980a] and [1980b]. The reader should consult these papers for a more thorough explanation of the language and its semantics.

A specification in CLEAR is built from algebraic theories using operations to modify or combine specifications in various ways. CLEAR allows us to build specifications in a modular manner.

Thus a CLEAR specification may simply be an algebraic theory in the form of a set of sorts, of operations and of equations. For example:

```
    constant SemiGroup =
            sorts element
            opns  mult : element, element -> element
            eqns
                  mult(a,mult(b,c)) = mult(mult(a,b),c)
                                                      end
```

Any semigroup is an algebra of the above specification. There are occasions (when we are dealing with data types) when we need to constrain the possible interpretations of a specification to those which are initial or, more generally, free in some sense. The DATA operation of CLEAR allows us to do this. For instance:

```
    constant Simple Bool =
       data  sorts  bool
             opns   true  : bool
                    false : bool
                                end
```

The keyword "data" here means that the only algebra of this specification is the initial algebra (in this case the initial algebra has only two elements in the carrier). In fact "data" also introduces an explicit equality on the sorts constructed.

It ought to be mentioned that CLEAR is not restricted to equational theories. More general theories (e.g. those with general predicate calculus expressions instead of equations) can be used only we must ensure that when a "data" occurs the relevant initial or free algebras exist. In the case of equational theories this is automatically the case, in general it is not so.

We can add extra sorts, operations and equations to a specification by the ENRICH operation:

```
    constant Bool =
       enrich Simple Bool by
            opns not : bool -> bool
                 and : bool, bool -> bool
            eqns
                 not(true) = false
                 not(false) = true
                 b and true = b
                 b and false = false
                                     end
```

Note that the equation 'x and y = y and x' does not follow by equational deduction from the equations in Bool. However because Simple Bool is to be interpreted initially the equation is valid in this specification (it follows by case analysis on the elements in the carrier of the initial algebra).

We now turn to the fixing together of specifications to build large specifications in a modular manner. Suppose we specify "relations" as follows:

```
    constant Relation =
       enrich Simple Bool by
            sorts element
            opns  rel : element,element -> bool
                                              end
```

We may want to add to this theory the further operations on booleans available in the theory Bool (for instance to specify equivalence relations). We could do this by an enrichment but since we have the theory of Bool at hand we use the CLEAR operation COMBINE (written as an infix "+"):

```
    constant Full Relation = Relation + Bool end
```

The theory Full Relation has the sorts and operations of both Relation and Bool but has only one "copy" of the theory Simple Bool - it being a common subtheory. In general COMBINE gives the disjoint union of the sorts and operations (with the associated equations) except where there is a common subtheory in which case only one copy of the sorts and operations in the common subtheory are retained. For instance in Full Relation there is only one operation "true" and one "false". This prevents a proliferation of sorts and operations which ought to be identified. It is handled semantically by introducing environments in which theories are defined.

CLEAR allows procedural (or parameterised) specifications. For instance we may specify lists of objects of an arbitrary type:

```
    procedure Lists(X : Triv) =
       enrich X by
            data sorts list
                 opns  nil : list
                       cons : element,list -> list
                                                    end
```

Notice that this procedure needs not only a formal parameter, X, but also a "constraint" (called a metasort) upon this parameter. The metasort itself is a theory. In this case the constraint upon X is merely that there is at least one sort in X so that we may build lists of objects of a particular sort. Thus the metasort is the theory Triv:

```
    constant Triv =
            sorts element
                      end
```

In general the metasort is more elaborate. For instance if we were defining a procedure to produce ordered lists of elements then instead of Triv we would need the theory of total-orders as a metasort.

To apply a procedure, we need to show how the actual parameter "fits" the metasort. For a theory of lists of booleans we would write:

```
    constant Bool Lists = List(Bool[element is bool]) end
```

showing that the sort 'element' in Triv is bound to the sort 'bool' in Bool. If the metasort has operations as well as sorts we need to show how these operations fit those in the actual parameter and that the fitting respects the equations which are present (that is, this fitting is a theory morphism).

The final operation supplied in CLEAR is a data abstraction facility. It is often convenient to build up a large specification and then abstract from it certain sorts and operations (and the equations associated with them) and "hiding" the rest. The CLEAR operation DERIVE allows us to do this. For instance, we might want to represent numbers as binary numbers using bit-strings from the theory Bool Lists:

```
    constant Numbers =
       derive sorts num
              opns  0 : num
                   succ : num -> num
             using Bool
             from enrich Bool Lists by
                     opns zero : list
                          add_one : list -> list
                     eqns zero = cons(false,nil)
                          add_one(cons(false,l)) = cons(true,l)
                          add_one(cons(true,nil)) =
                             cons(false,cons(true,nil))
                          add_one(cons(true,cons(x,l))) =
                             cons(false,add_one(cons(x,l)))
             by num is list
                0 is zero
                succ is add_one
                                                          end
```

Thus the representation of a number is a list of booleans with the least significant figure at the head. Notice that the operations 'nil' and 'cons' are not available in Numbers - they are "hidden".

The foregoing informal description of CLEAR is made precise with a formal semantics in [Burstall and Goguen 1980a]. Several reasonably substantial specifications have been written in CLEAR including a "garbage collector" [Burstall and Goguen 1980b].

INDEX OF DATA TYPES AND FUNCTIONS

BIBLIOGRAPHY

Aho A.V., Hopcroft J.E. and Ullman J.D. (1974) The Design and Analysis of Computer Algorithms. Addison Wesley.

Adamek J. (1979) Construction of Free Ordered Algebras. Faculty of Electrical Engineering. Fel Cvut, Suchbatrarova 2. 166 27 Praha 6. Czechoslovakia.

Adamek J. and Koubek V. (1980) Are Colimits of Algebras Simple To Construct? J. Algebra, 66, 226-250.

Adamek J. and Trnkova V. (1978) Varietors and Machines. Comp. and Inform. Science Technical Report, 78-6, University of Massachusetts at Amherst.

Backus J. (1978) Can programming be liberated from the Von Neumann style? Comm. A.C.M 21. 8. pp 613-641

Barr M. (1970) Coequalisers and Free Triples. Math. Z. 2. Vol 116. pp 307-332.

Birkhoff G. (1938) Structure of Abstract Algebras. Proc. Cambridge Phil. Soc. 31 pp 433-454.

Broy M. and Wirsing M. (1980) Initial versus Terminal Algebra Semantics for Partially Defined Abstract Types. Institut Fur Informatik, Technische Universitat Munchen.

Burstall R.M. (1972) An Algebraic Description of Programs with Assertions, Verification and Simulation. Proc A.C.M. Conf. on Proving Assertions about Programs. New Mexico.

Burstall R.M. (1980) Electronic Category Theory. Proc. Ninth International Symposium on The Mathematical Foundations of Computer Science, Rydzyua, Poland.

Burstall R. M. and Darlington J. (1977) A transformation system for developing recursive programs. J.A.C.M. 24 44-67.

Burstall R. M. and Goguen J. A. (1977) Putting Theories Together To Make Specifications. In Proc. of the Fifth International Joint Conference on Artificial Intelligence, Boston.

Burstall R.M. and Goguen J. A. (1980a) The Semantics of Clear, a Specification Language. Proc. 1979 Copenhagen Winter School on Abstract Software Specification.

Burstall R.M. and Goguen J.A. (1980b) An Informal Introduction to Specifications using CLEAR. Dept. of Computer Science, University of Edinburgh.

Burstall R.M. and Goguen J.A. (1980c) CAT, a System for the Structured Elaboration of Correct Programs from Correct Structured Specifications. Research Report, SRI International, Menlo Park, Calif.

Burstall R.M. and Landin P.J. (1969) Programs and Their Proofs: An Algebraic Approach. Machine Intelligence 4, Edinburgh: University Press. 17-44.

Burstall R.M., MacQueen D.B. and Sannella D.T. (1979) HOPE: An Experimental Applicative Language. Comp. Sci. Dept. Report, University of Edinburgh.

Burstall R.M. and Rydeheard D.E. (1979) The Free Algebraic Theory on a Signature. Unpublished Report. Dept. of Artificial Intelligence, Univ. of Edinburgh.

Burstall R.M. and Thatcher J.W. (1974) The Algebraic Theory of Recursive Program Schemes. Proc. of First International Symp: Category Theory Applied to Computation and Control. University of Amherst.

Darlington J. (1978) A Synthesis of Several Sorting Algorithms. Acta Informatica 11, 1-30.

Dijkstra E.W. (1968) Goto statement considered harmful. Comm. A.C.M. 11 pp 147-148.

Dubuc E.J. (1974) Free Monoids. J. Algebra, 29, 208-228.

Dungan D.M. (1979) Bibliography on Data Types. SIGPLAN Notices 14, 11, 31-59.

Ehresmann Ch., (1960) Categorie des foncteurs types. Rev. Un. Mat. Argentina, 20 pp 194-209.

Ehresmann Ch., (1963) Categories doublees et categories structurees. C.R. Acad. Sci. Paris. 256 pp 1198-1201.

Ehrig H., Kreowski H-J., Maggiolo-Schettini A., Rosen B. and Winkowski J. (1977) Deriving Structures from Structures. Research Report. IBM Research Centre, Comp. Sci. Dept., Yorktown Heights, N.Y.

Ehrig H., Pfender M., and Schneider H. (1973) Graph Grammars: An Algebraic Approach. Proc. 14th. Ann. IEEE Symp. on Switching and Automata Theory, pp 167-180.

Ehrig H. and Rosen B. (1980) Mathematics of Record Handling. SIAM J. Comput. Vol. 9, No. 3.

Eilenberg S. and Moore J.C. (1965) Adjoint functors and Triples. Ill. J. Math. 9 pp 381-398.

Ehrich H.-D. (1978) Extensions and Implementations of Abstract Data Type Specifications. Proc. 7th. Symp. Math. Foundations of Comp. Science. Zakopane, Poland. Springer-Verlag.

Freyd P. (1972) Aspects of Topoi. Bull. of Australian Math. Soc. Vol. 7, 1-76.

Godement R. (1958) Theorie des Faisceaux. Hermann.

Goguen J. A. (1978) Order-Sorted Algebras: Exception and Error Sorts, Coercions and Overloaded Operators. Unpublished Report, UCLA Los Angeles.

Goguen J.A. and Burstall R.M. (1978) Some Fundamental Properties of Algebraic Theories : A Tool for the Semantics of Computation. Dept. of Artificial Intelligence Report No. 53., University of Edinburgh.

Goguen J. A., Thatcher J. W. and Wagner E. G. (1978) An initial algebra approach to the specification, correctness and implementation of Abstract Data Types. Current Trends in Programming Methodology, IV, Data Structuring. Prentice Hall pp 80-149.

Goguen J.A., Thatcher J.W., Wagner E.G. and Wright J.B. (1975) An Introduction to Categories, Algebraic Theories and Algebras. I.B.M. Research Rep. RC 5369. T.J. Watson Research Centre, Yorktown Heights, N.Y.

Goguen J.A., Thatcher J.W., Wagner E.G. and Wright J.B. (1977) Initial Algebra Semantics and Continuous Algebras. Jour. A.C.M. 24. 1. pp 68-95.

Guttag J.V., Horowitz E. and Musser D.R. (1978) Abstract Data Types and Software Validation. Comm. A.C.M. 21, No. 12, pp 1048-1063.

Herrlich H. and Strecker G. E. (1973) Category Theory. Allyn and Bacon.

Hoare C.A.R. (1974) Proofs of Correctness of Data Representations. Acta Inform. 1. pp 271-281.

Kan D. M. (1958) Adjoint functors. Trans. American Math. Soc. 87 pp 294-329.

Kaphenst H. and Reichel H. (1971) Algebraische algorithmentheorie. VEB Robotron, Zentrum fur Forschung und Technik, Dresden, WIB.

Klaeren H. (1980) An Abstract Software Specification Technique Based on Structural Recursion. SIGPLAN Notices 15, 28-34.

Kleisli H. (1965) Every Standard Construction is induced by a pair of Adjoint Functors. Proc. Am. Math. Soc. 16. pp 544-546.

Landin P.J. (1966) The Next 700 Programming Languages. Comm. A.C.M. 9, 3, pp 157-166.

Landin P.J. (1969) A Program Machine Symmetric Automata Theory. Machine Intelligence 5, Edinburgh University Press.

Lawvere F.W. (1963a) Functorial Semantics of Algebraic Theories. Proc. Nat. Acad. of Science. 50 pp 869-872

Lawvere F.W. (1963b) Functorial Semantics of Algebraic Theories. Ph.D. Thesis. Columbia University.

Lehmann D.J. and Smyth M.B. (1977) Data Types. Theory of Computation Report No. 19. University of Warwick.

Linton F.E.J.(1969) Coequalisers in Categories of Algebras. Lecture Notes in Mathematics. No. 80. Springer-Verlag.

Liskov B.H. and Zilles S.N. (1974) Programming with Abstract Data Types. Proc. A.C.M. SIGPLAN Sym. on Very High Level Languages. SIGPLAN Notices 9 pp 50-59.

Mac Lane S. (1948) Groups, categories and duality. Proc. Nat. Acad. Sci. U.S.A. 34 pp 263-267.

Mac Lane S. (1965) Categorical Algebra. Bull. of the Amer. Math. Soc. 71 pp 40-106

Mac Lane S. (1971) Categories for The Working Mathematician. Springer-Verlag.

Mac Lane S. (1975) Sets, Topoi and Internal Logic in Categories. Logic Coll., Bristol. 1973. North Holland.

Majster M.E. (1979) Data Types, Abstract Types and Their Specification Problem. Theor. Comp. Sci. 8 pp 89-113.

Manes E.G. (1976) Algebraic Theories. Springer-Verlag, New York.

Manna Z. and Waldinger R. (1980) A deductive approach to program synthesis. A.C.M. Trans. on Prog. Lang. and Sys. Vol. 2, No. 1.

Meseguer J. (1978) Completions, Factorisations and Colimits for ω-posets. Semantics and Th. of Computation

Report No. 13. U.C.L.A.

Milner R. (1978) A Theory of Type Polymorphism in Programming. J. of Compn. and System Sci. 17 No. 3 pp 348-375.

Morris F. L. (1973) Advice on Structuring Compilers and Proving Them Correct. A.C.M. Symp. on the Principles of Programming Languages. pp 144-152.

Morris J.H. (1973) Types are not Sets. A.C.M. Sym. on the Principles of Programming Languages. pp 120-124.

Nourani F. (1980) On Induction for Programming Logic : Syntax, Semantics and Inductive Closure. Bulletin of EATCS. Vol. 3. No. 13. pp 51-64.

Parsaye-Ghomi K. (1981) Higher Order Data Types. Submitted for PhD. Comp. Science Dept. U.C.L.A.

Reichel H. (1980) Initially Restricting Algebraic Theories. Proc. of Mathematical Foundations of Computer Science, Vol. 88, Springer-Verlag, Berlin. pp 504-514.

Reynolds J.C. (1978) Category Sorted Algebras. Manuscript, Syracuse University.

Robinson J. A. (1965) A machine - oriented logic based on the resolution principle. J.A.C.M. 12. 1. 23-41.

Sannella D. (1981) A New Semantics for Clear. Dept. of Comp. Sci. University of Edinburgh. Unpublished Report.

Schubert H. (1972) Categories. Springer-Verlag.

Schwartz J. T. (1980) Ultracomputers. A.C.M. Trans. on Prog. Lang. and Sys. Vol. 2 No. 4.

Smyth M.B. and Plotkin G.D. (1977) The Category-Theoretic Solution of Recursive Domain Equations. Proc. of Foundations of Comp. Sci.

Thatcher J.W., Wagner E.G. and Wright J.W. (1981) More on Advice on Structuring Compilers and Proving Them Correct. Theoretical Comp. Sci. 15 pp223-249.

Wagner E., Wright J., Goguen J.A. and Thatcher J. W. (1978) Some Fundamentals of Order-Algebraic Semantics. Research Report, IBM Laboratories, Yorktown Heights, N.Y.

Wand M. (1979) Final Algebra Semantics and Data Type Extensions. Jour. of Compn. and System Sci. 19. 27-44.

Warshall S. (1962) A Theorem on Boolean Matrices.
Jour. A.C.M Vol. 9. No. 1. pp 11-12.