PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

PROGRAMMING WITH MODULES AS TYPED FUNCTIONAL PROGRAMMING

Rod Burstall

Department of Computer Science, University of Edinburgh,

King's Buildings, Mayfield Rd.,

Edinburgh EH9 3JZ, Scotland U.K.

Abstract

In the 1970's languages were provided with module facilities for building large programs. These notations can be understood and refined by translating them into a typed functional language which provides 'dependent types'. In such a language we can denote a large program by an expression whose operators are modules. This paper tries to give a readable introduction to the way in which recent ideas about types may be used for the study of modularity features in programming languages.

1. Introduction

Our programming ambitions are bounded by the size of programs which we can debug and maintain. When a program exceeds a certain size no-one in the world understands it. But we hope that each part of it will be understood by at least one person and that someone will understand how the parts fit together. Fitting the parts together has been called 'programming in the large', as opposed to 'programming in the small' which is the business of forming the parts by writing assignments, **while** statements and procedures. Pascal, LISP and Prolog permit programming in the small. CLU, Mesa, Modula2, ADA and ML add facilities for programming in the large, variously called 'clusters', 'modules', 'packages' and 'abstract data types'.

When we combine modules to perform some task we need to look at their 'interfaces'. The interface tells us what data types and procedures the module requires and what it produces. To ensure that we are combining the modules in a sensible way we check that their various input and output interfaces agree with each other.

Unfortunately, modules and interfaces form an extra layer of complexity in recent programming languages, yet more features for the programmer to wrestle with. We would like to find a uniform framework which would 'explain' these extra features as well as the ones which are familiar from programming in the small, explaining modules as well as procedures. Butler Lampson and I have worked to show that such a framework is provided by a typed functional language. Calling a collection of data type and procedure definitions an 'implementation' we regard modules as functions from implementations to implementations and interfaces as the types of the implementation. Conventional languages can be defined by translation into such a simpler 'Kernel' language.

We hope that our work will show how to reduce the apparent complexity of programming languages in the facilities they provide for dealing with modules. This rationalisation should give us tools to design more uniform and powerful facilities for programming in the large.

The attempt to use a typed functional language for programming with modules shows up some of the limitations of the type system usually employed, and it points to the need to adopt a more flexible and expressive system using 'dependent types'. These were originally proposed in order to deal with the logic of constructive mathematics (Martin-Lof 1973), but recently a number of people have become aware of their importance for programming languages.

The approach here is based on the 'Pebble' functional language of Lampson and myself which is fully defined in Burstall and Lampson (1984). Indeed the present paper is mainly an attempt to motivate and give a summary account of the joint work with Lampson described in that paper. The topic addressed seems to be generating considerable interest recently; in particular we have exchanged ideas with David MacQueen at Bell Laboratories and with Gordon Plotkin at Edinburgh, and they have collaborated with Ravi Sethi, John Mitchell and Jim Hook. The inspiration for this work came from a long collaboration with Joe Goguen and from the sophisticated module facilities provided in the Xerox PARC Mesa language. The theory of polymorphic types was developed by Reynolds and by Milner.

The other stream of work which has contributed is in the logic of constructive mathematics and influential work here has been due to Martin-Lof, to Girard, and to Consteble and his collaborators at Cornell.

2. Building programs out of modules

Let us review the current state of the art. We start off with some building blocks for programs, ones which are familiar in the context of languages such as UCSD Pascal, Modula 2, Mesa and ADA. (See Appelbe and Ravn 1984 for a recent critique of modularity features of several such languages.)

This will fix ideas on current practice and establish some terminology, namely *implementation*, *interface* and *module*.

Consider for example a typical piece of program to enable us to do computations with points, as shown in Fig. 2-1.

```
POINT
type point = ↑ record xcoord:real;
                      ycoord:real
               end;                                   INTERFACE
function mkpoint(x:real; y:real):point;
procedure rotate(p:point; theta:real);
------------------------------------------
function mkpoint;
  var p:point;
  begin new(p);                                       IMPLEMENT-
        p↑.xcoord:=x; p↑.ycoord:=y;                       ATION
        mkpoint:=p
  end;
procedure rotate;
  begin ..... end
```

Figure 2-1: Points

The interface introduces names such as *point*, *mkpoint* and *rotate* and associates a type with each of them, but no executable code; this code is supplied by the implementation. A program consists of a number of such pieces, referring to each other (Fig. 2-2).

```
POINT
type point = ↑ record...end;
function mkpoint(x:real; y:real):point;
procedure rotate(p:point; theta:real);

<implementation of point>
```

```
LINE
USES POINT
type line = ↑ record...end
function mkline(p:point; q:point):line
function intersects(l1:line; l2:line):point

<implementation of line>
```

```
PICTURE
USES POINT, LINE
type pic = record...end
function mkpic(p:point; al:array of line):pic;
procedure display(p:pic; var d:array of boolean)

<implementation of picture>
```

Figure 2-2: Points, lines and pictures

The point interface of Fig. 1 is in fact rathe too informative. It was realised long ago that i order to maintain some abstraction about th notion of point the interface should not show wha points are made of. The data type point should b *opaque* (Fig. 2-3), making point into an *abstrac datatype* whose implementation details are not i public view.

```
POINT
type point                                            INTERFACI
function mkpoint(x:real; y:real):point;
procedure rotate(p:point; theta:real)
------------------------------------------
type point = ↑ record xcoord:real;
                      ycoord:real
               end;                                   IMPLEMENT-
function mkpoint;                                          ATIO
  ...
procedure rotate;
  ...
```

Figure 2-3: Points as an abstract data type

The next good idea was to separate th implementation from the interface (Fig. 2-4).

```
POINT
type point                                            INTERFACE
function mkpoint(x:real; y:real):point;
procedure rotate(p:point; theta:real);
```

```
POINT
type point = ↑record...end
function mkpoint;                                     IMPLEMENT-
  ...                                                      ATION
procedure rotate;
  ...
```

Figure 2-4: Points with separate implementation

After all there could be several implementations of the same interface, different ways of representing and computing with points. This means that the conventional way of connecting an implementation to an interface, simply giving them both the same name *POINT*, is not adequate in general. We shall look later at an alternative way of making this connection.

So far we have seen a box which produces an implementation. Sometimes this implementation is not fixed but depends on some parameters, as in a general *sort* function which requires to be supplied with the type of the elements to be sorted and an appropriate ordering; we shall call this a module, (*SORT* in Fig. 2-5).

But the requirement is rather familiar, indeed it is just another example of an interface. What has to be supplied to fulfil this requirement is just an implementation. Thus we can use the usual parameter notation when writing modules. (This insight appeared in Mesa and in Clear.)

To see this let us look at an example where we

```
SORT
Requires(type element;
         function lesseq(e1,e2:element):boolean)

 procedure sort(var a:array of elem)
```

```
MAIN
USES STRINGS, SORT(string; stringlesseq)
 ...
```

Figure 2-5: Parameters for *SORT*

define some interfaces, implementations and modules.

First we have an interface *ORDERING* and two different implementations of it, one using numbers and one using strings (Fig. 2-6).

```
interface ORDERING
type element
function lesseq(e1,e2:element):boolean
```

```
implementation NUMERIC_ORD:ORDERING
type element = integer
function lesseq(i1,i2:integer):boolean
  begin if i1<i2 then...else...
  end
```

```
implementation STRING_ORD:ORDERING
type element = string
function lesseq(s1,s2:string):boolean
  begin for j=1 to .....
  end
```

Figure 2-6: Orderings

Then we have a module for lexically ordering elements, (Fig. 2-7), producing a new ordering on pairs of elements from two given orderings. Thus if words are ordered alphabetically and numbers in increasing order. The pairs <frog,3>, <rain,4>, <water,0>, <grass,3>, <rain,1> would be lexically ordered <frog,3>, <grass,3>, <rain,1>, <rain,4>, <water,0>. The parameters P and Q are just required to be *ORDERING*s; it is no longer necessary to list the individual types and procedures required as we did in Fig. 2-4.

```
module LEXICAL_ORD(P:ORDERING,Q:ORDERING)
                                :ORDERING
type element=record x:P.element; y:Q.element
              end
function lesseq(p1,p2:element):boolean
   begin if P.lesseq(p1.x,p2.x) then...else...end
```

Figure 2-7: Lexical Ordering

We can now define a sorting interface, two different ways of realising it and a main program which uses all this (Fig. 2-8).

```
interface SORTING
type element
procedure sort(var a:array of element; n:integer)
```

```
module SHELLSORT(P:ORDERING):SORTING
  .....
```

```
module QUICKSORT(P:ORDERING):SORTING
  .....
```

```
MAIN
  .....
```

Figure 2-8: Sorting

Mesa has most of this capability, with rather complicated syntax, but it does not distinguish between implementations and modules. Logically we can write modules with no parameters instead of the implementations, doing without implementations in the programming language. But then the idea of implementation is just explanatory and semantic; you cannot write down an implementation. I believe that this makes it harder to understand what is going on.

If we forget about what is in the boxes and look just at the lines on top of them we get a very important idea: a program plan (Fig. 2-9). This is where 'thinking big' happens. It gives us an overview of the whole program structure, omitting the details. But it is precise and capable of further refinement, not just informal hand waving.

```
interface ORDERING, SORTING
implementation NUMERIC_ORD:ORDERING;
               STRING_ORD :ORDERING
module LEXICAL_ORD(P,Q:ORDERING):ORDERING;
       SHELLSORT(P:ORDERING):SORTING;
       QUICKSORT(P:ORDERING):SORTING
```

Figure 2-9: A program outline

This is best expressed as a picture with interfaces as the nodes and implementations and modules on the arrows, (Fig. 2-10).

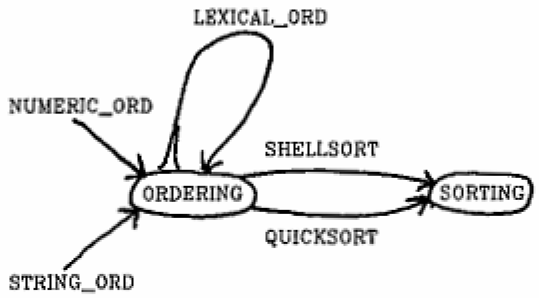

Figure 2-10: Diagram for program outline

We can use this picture to have fun devising various programs (rather as the 'railroad diagrams' for Pascal Syntax allow you to devise programs syntactically). For example:

define QuSortNS=QUICKSORT(
  LEXICAL_ORD(NUMERIC_ORD,STRING_ORD))

So now a program looks like an expression in terms of implementations and modules, which can be *type-checked* by checking compatibility of interfaces. It is really nice to be able to write expressions to denote big programs.

It is possible to base a module linking language on these ideas, and indeed this has been done at Xerox PARC (Schmidt 82, Lampson and Schmidt 83). Schmidt designed and implemented a 'System Modelling' language, based on an early version of Pebble, to express the linking of large collections of Mesa modules.

In the next sections we will describe the formal basis for this view of large scale programming, a basis which has been embodied in the functional kernel language Pebble (Burstall and Lampson 84).

3. Implementations, interfaces and modules

We would now like to remove the special character of the constructs which we have encountered in our examples of modular programming, so as to treat them just like such old friends as integers and procedures.

The analogy which we seek to establish is

implementation	— value
interface	— type
module	— function

If we can make sense of this analogy, 'programming in the large' with implementations, interfaces and modules will become just typed functional programming. This endeavour motivated the design of Pebble, one of a number of recently proposed typed functional languages which extend traditional ideas of types in the light of work in the theory of constructive mathematics.

Let us see how we can regard implementations as values. Based on the examples we have looked at we may say that:

an implementation is

either a name bound to a basic value

(0,1,2,...,true,false,...)

or to a function

or to an n-tuple

or to a type

or an n-tuple of implementations

This suggests that we define our value space as follows:

a value is

either a basic value (0,1,2,...,true,false,...)

or a function

or an n-tuple of values

or a type

or a binding

where

a binding is either a name bound to a value

or a tuple of bindings

Notice that we have replaced the word *implementation* by *binding*. This is because the idea of a collection of names bound to some kind of values is already familiar in most functional languages and need not be restricted to the particular context of 'programming in the large'. In fact a binding is what is often described after let, for example

let x=u+2 *and* p=*not* q *or* r
in if p *then* x-1 *else* x-1

We think of the expression after *let* being evaluated to give a binding, and prefer to think of it as a pair of elementary bindings, which we write using '~' rather than the overworked sign '='. Thus

let [x ~ u+2, p ~ *not* q *or* r]
in if p *then* x-1 *else* x-2

In a situation where u is 1, q is false and r is true, the expression after let evaluates to the binding [x ~ 3, p ~ *true*].

Bindings will do very well to represent implementations.

Similarly

an interface is

either a name with a given type

or a 'product' of interfaces

So since an interface is to be the type of an implementation we need

a type is a basic type (integer,boolean,...)

or type → type

or type × type

or a declaration

where

a declaration is

either a name with a given type

or a declaration × declaration

Now a declaration is the type of a binding. For example the binding

$[x \sim 3,\ p \sim true]$

has as its type the declaration

$x{:}integer \times p{:}boolean$

(We shall see later that as well as the usual notions of × and → we will need *dependent* versions in the above definitions.)

Another example: the binding

$[element \sim integer,\ f \sim factorial]$

has as its type the declaration

$element{:}type \times f{:}(integer \to integer)$

Here *type* is a special type, namely the type of all types (including itself). Introducing such a type lays us open to the possibility of paradoxes (Girard 1972). We believe however that the system may be appropriate for computing purposes.

Declarations will do very well to represent interfaces.

We can now summarise three of the major decisions taken in designing Pebble.

- We take *types* as *values*
- We take *bindings* as *values*
- We take *declarations* as *types*

From a theoretical point of view Pebble is just typed lambda calculus with some syntactic sugar (notably the let construction) and the three extensions above plus a fourth, *dependent types*, which we will discuss shortly. We designed it with two aims in mind

- to act as a kernel language into which we could translate the much richer and more *ad hoc* syntax of PARC's Cedar language so as to give a precise definition of Cedar. (Cedar is a successor to Mesa.)
- to enable us to extend the Cedar language sufficiently to act as its own module linking language.

4. Bindings and declarations

Let us see how we can play with bindings and declarations. Bindings naturally appear after let, as we have seen. Declarations naturally appear in formal parameter lists i.e. after 'λ', thus λ *x:int*... . As usual we permit ourselves to write *f(x:int)*~... instead of the more primitive *f* ~ λ*x:int*..... Here are some examples of manipulating bindings and declarations

```
let x:int ~ 1 in
let d:type ~ (y:int × p:bool) in {note: d is a decl}
let b:d ~ [y ~ x+3, p ~ true] in
                                  {note: b is a binding}
let b in
   if p then y else y+1
```

We may also define a function f which takes a binding as argument. After the definition of d above we could write

```
let f(b:d) ~ let b in
               if p then y else y+1
```

Here *f* takes as an argument a binding *b* whose type is given by a declaration *d*. Notice that we can typecheck the function *f* without knowing the value of *b*, but we must know the value of the declaration *d* since this tells us that *p* is a boolean and *y* is an integer. We could rewrite the definition of *f* thus

```
let f(b:d) = if (let b in p) then (let b in y)
                              else (let b in y) + 1
```

Here *let b in p* means *the value of the name p in the binding b*, and it is closely equivalent to the form *b.p* in Modula 2 (for example), meaning the identifier *p* defined in module *b*. A familiar example would be *stack.empty*.

Now we can say that

- implementations are bindings
- interfaces are declarations
- modules are functions from bindings to bindings

5. Dependent cross types

If we try to use bindings for implementations and declarations for interfaces, we come across a difficulty. Consider the interface

```
ORDERING
type element
function lesseq(e1,e2:element):boolean
```

We would like to write this interface as

$(element{:}type) \times (lesseq{:}element \times element \to boolean)$

When we write $(x{:}s) \times (y{:}t)$ we mean to declare the types of *x* and *y* and assume that we already know the values of *s* and *t* (which should of course be types). But here the use of *element* after *lesseq*: does not refer to some pre-existing type *element*; it refers to the *element* declared by *element:type*, and we will not know what type *element* is meant until we get the binding to correspond to this declaration. Fortunately this situation was encountered in the type theory of constructive mathematics (Howard 1969, Scott

1970, Martin-Lof 1973, see Constable and Zlatin 1984 for a recent formulation). It was also observed by Constable and Bates(1984) in work on program verification and by Demers and Donahue in designing their language Russell (Demers and Donahue 1980) . We need what is known as a *dependent type*, of which there are two kinds, corresponding to × and to → respectively (∃ and ∀ if one uses a more logical notation, $\sum$ and $\prod$ in Martin-Lof's notation). Interfaces need the first of these, dependent cross.

To distinguish the dependent cross from the usual one we write it with $\times\!\!\times$. Before the $\times\!\!\times$ we may write not just a type but the declaration of a name which may be used after the $\times\!\!\times$. Thus we can write

$$(element{:}type) \times\!\!\times (lesseq{:}element \times element \rightarrow boolean)$$

The general form is

$n{:}T1 \times\!\!\times T2(n)$

where n is a name

$T1$ is an expression denoting a type

$T2$ is an expression denoting a type and

(optionally) using the name n.

This dependent cross denotes a type whose elements are pairs, the first component being an element of type T1 and the second an element whose type depends on the value of the first component. Thus *the type of the second component depends on the value of the first.*

Some examples will help

(i) $(n{:}int) \times\!\!\times (array[1..n] \ of \ real)$

elements [4, 7.1 1.2 6.0 8.5]
[3, 7.2 1.5 1.7]

(ii) $(t{:}type) \times\!\!\times (t \times list(t))$

elements [*int* , [6, (1,2,3)]]
[*bool* , [*true*,(*true*,*false*)]]
[*list*(*int*) , [(1,2),((1,3,2),(4,1))]]

(iii) $(\varphi{:}type \rightarrow type) \times\!\!\times (\varphi(int) \rightarrow int)$

element [*list* , *CAR*]

The type-checking rule for dependent cross types may be written thus:-

If a has type T1
and b has type T2(a)
then [*a*,*b*] *has type* $n{:}T1 \times\!\!\times T2(n)$

Thus in example (ii) above

int has type type
and [6,(1,2,3)] *has type int* × *list*(*int*)
so [*int*,[6,(1,2,3)]] *has type* $(t{:}type) \times\!\!\times (t \times list(t))$

Dependent cross types are not altogether unfamiliar. A textbook definition of a finite automaton might be
"An automaton is a 5-tuple (I,S,O,δ,γ) where I is a set of inputs, S a set of states, O a set of outputs, δ: I×S→S is a function and γ: S→O is a function."
We can imitate this in our typed language by the dependent cross type

$$(I{:}type \times S{:}type \times O{:}type) \times\!\!\times ((I{\times}S \rightarrow S) \times (S \rightarrow O))$$

Here the second component is a product type, but to reproduce the informal definition more closely we can make it into a declaration

$$(I{:}type \times S{:}type \times O{:}type) \times\!\!\times ((\delta{:}I{\times}S \rightarrow S) \times (\gamma{:}S \rightarrow O))$$

Part of the motivation for our investigation of more expressive type systems came from our experience of coding up category theory as programs (Burstall 1980, Rydeheard 1982, Rydeheard and Burstall 1983). David MacQueen spent considerable effort rewriting our category programs using a more sophisticated type system; indeed we regard them as something of a challenge for language designers. This contributed to the development of MacQueen's proposals for a module system for ML (MacQueen 1984).

6. Dependent arrow types

The other form of dependent type can be regarded as a generalisation of the → type constructor, as used for example in *integer* → *boolean*. We write it →>. It enables us to introduce *polymorphism*, a facility which is available in some sophisticated type systems, notably the ML language (Gordon, Milner and Wadsworth 1979, Milner 1983 , Mycroft and O'Keefe 84), but not in Pascal and its descendants. In Pascal if we want to write a procedure to transpose a two-dimensional array, so that a_{ij} becomes a_{ji}, we have to know whether it is an integer array or a real array. Although the code is the same in either case we have to write two separate procedures, *TransposeIntegerArray* and *TransposeRealArray*. We would like to have a single procedure *TransposeArray* which could handle an array with any type of element. To achieve this in Pebble we use the approach proposed by Reynolds (1974), that is we give *TransposeArray* an extra parameter, namely the type of the elements in the array. Since we have already decided to treat types as values there is no particular difficulty in passing a type as a parameter. (Reynolds who maintained the usual distinction between types and values had to introduce a new abstraction operator Λ for types, analogous to λ for values.)

Thus we can write *TransposeArray* as a function which takes a type, *t*, as argument and delivers as its result a function; this resulting function takes an array of *t* as its argument and delivers a transposed array of *t* as its result. More briefly *TransposeArray* given a type *t* delivers the transposition function for arrays of type *t*. (Such function producing functions are permissible in LISP or ML but not in Pascal.) For example we may use *TransposeArray* on an integer array *a* thus

let a1: array of int ~ *TransposeArray*(*int*)(*a*)

But what is the type of this function *TransposeArray*? Its argument is a type *t* and its result is a function of type *array of t* → *array of* *t*. So we might write

type → (*array of t* → *array of t*)

but then the name *t* is not declared. So instead we use a dependent arrow type:-

t:type →> (*array of t* → *array of t*)

The declaration before the →> introduces the name *t* which may then be used after the →>. (Reynolds would write this as Δt.(*array of t* → *array of t*)).

The general form is

n:*T*1 → *T*2(*n*)

where n is a name

T1 is an expression denoting a type

T2(n) is an expression denoting a type and optionally using the name n.

The elements of this type are functions whose argument is of type T1. *The type of the result depends on the value of the argument.*

Other examples are

(*i*) (*n*:*integer*) →> (*array* [1..*n*] *of real*)

elements - *the function which for any integer* $n \geq 0$ *produces an array of zeros of length n*

- *the function which for any integer* $n \geq 0$ *produces the array of square roots of the first n integers.*

(*ii*) (*t*:*type*) →> (*list*(*t*) → *list*(*t*))

elements - *the function which for any type t produces the identity function for lists of elements of type t*

- *the function which for any type t produces the reverse function for lists of elements of type t.*

The type-checking rule for dependent arrow types may be written thus:-

If for an arbitrary a of type T1

f(a) has type T2(a)

then f has type n:T1 →> *T2(n)*

In Pebble we take a symbolic value for *a* and perform symbolic evaluation at type checking time. This is a computationally oriented approach, rather than a semantically oriented one.

A somewhat different approach to polymorphism is taken in ML. There *TransposeArray* is not given an extra parameter. Its type is

array of t → *array of t*

where t is a type variable by global convention rather than explicit declaration. This can be written more explicitly with a universal quantifier

(*for all t*)(*array of t* → *array of t*)

Now in ML *TransposeArray* may be applied directly to an integer array (for example) and a *type inference system* is used to determine that here *t* must be an integer. This inference system uses *unification*, as in resolution theorem proving or in Prolog, but now at the type level (Milner 1978). A system with static type checking using such quantified types is described by MacQueen and Sethi (1982); they give a denotational semantics and type inference rules. The unification method is more elegant and convenient than passing the type as an explicit parameter. However it does not extend to higher order type variables, and if these are allowed we have to resort to explicit parameters.

7. The sorting example in Pebble

We are now able to show the sorting example of Figures 2-6, 2-7 and 2-8 written in Pebble (Figure 7-1). The interface *ORDERING* becomes a declaration, the implementations *NUMERIC_ORD* and *STRING_ORD* become bindings, the module *LEXICAL_ORD* becomes a function from bindings to bindings, and so on.

Pebble primitive bindings have the form <name>:<type> ~ <expression> but the <type> may be omitted. Bindings to functions may include parameter names and use *is* instead of ~. Bindings are connected by ';' (sequential) or by *and* (simultaneous). *P$element* is equivalent to *let p in element*, except that an error occurs if *p* does not bind *element*. This example uses ⇒ but not →>, since there are no polymorphic functions.

8. Sharing implementations

Our general aim has been to provide a flexible means of building large programs. It is sometimes useful to have more than one implementation of a given interface, for example one implementation might be small but slow, another faster but requiring more space, perhaps an optimised version of the former. In languages such as Modula 2 or Ada, which attach the implementation to the interface by using the same name, this is difficult. In Mesa or Pebble it is straightforward using the parameter mechanism, similarly in OBJ (Goguen, Meseguer and Plaisted 82) and in MacQueen's ML module proposal (MacQueen 84).

It is perhaps worth noting that the distinction between 'classical' algebra and 'modern' algebra can be seen as the difference between one implementation of an interface and multiple implementations. Classical algebra studied individual structures such as real numbers, complex numbers and matrices over these; think of these as implementations and their axiomatisation as the interfaces (with some specifications added to the declarations). Modern algebra studies axiom systems such as 'group' (an interface) which may be interpreted by many different structures (implementations).

However if an interface can have multiple implementations a new problem arises. How do we

```
let
ORDERING:type ~
  (element:type) ⋉
  (lesseq:element × element → bool);
NUMERIC_ORD:ORDERING ~
  (element: ~ int,
   lesseq:(i1:int × i2:int) → bool is ...)
and
STRING_ORD:ORDERING ~
  (element: ~ string,
   lesseq:(s1:string × s2:string) → bool is ...)
and
LEXICAL_ORD:(P:ORDERING × Q:ORDERING) →
                         ORDERING is
  (element: ~ (x:P$element × y:Q$ element);
   lesseq:(eq:element × e2:element) → bool is ...)
and
SORTING: ~ element:type ⋉
          sort:array(element) → array(element);
SHELLSORT:(P:ORDERING) → SORTING IS ...
and
QUICKSORT:(P:ORDERING) → SORTING IS ... ;
QuSortNS:sorting ~ QUICKSORT(
   LEXICAL_ORD(NUMERIC_ORD,STRING_ORD))
```

Figure 7-1: Sorting example in PEBBLE

indicate that a module with two parameters requires them to be satisfied by implementations which share a common subimplementation. Suppose for example we have a linear algebra module, it might have two parameters whose interfaces are 'vectors' and 'matrices' respectively.

Linear algebra: vectors × matrices → whatever

But this only makes sense if the vectors and matrices are over the same kind of element, say both over reals or both over complexes (technically they have to be over the same field).

In Pebble we can express this by introducing an extra parameter and parameterising vector and matrix (this device derives from Mesa).

```
field:type ~ ...                              INTERFACE
vectorover:field → type ~ ...
                          PARAMETERISED INTERFACE
matrixover:field → type ~ ...
                          PARAMETERISED INTERFACE
linearalgebra: F:field ⋉ (vectorover(F)
                  × matrixover(F)) → whatever
                                           MODULE
```

A typical call of linear algebra might be

```
Real:field ~ ...                         IMPLEMENTATION
Vector1over:(F:field →> Vectorover(F)) ~ ...
                                                MODULE
Matrix1over:(F:field →> Matrixover(F)) ~ ...
                                                MODULE
Linearalgebra(Real,Vector1over(Real),
              Matrix1over(Real))
                                        IMPLEMENTATION
```

This might be called 'sharing by parameterisation'. MacQueen's proposal for a module facility in ML (MacQueen 84) suggests another approach which we might call 'sharing by equations'. The idea is to allow one to add equations to an interface. Consider the declaration

$(i{:}integer \times j{:}integer) \times (k{:}integer \times l{:}integer)$

A binding of this type is a $[[i=1,j=2], [k=3,j=4]]$.

But we might add an equality constraint

$(i{:}integer \times j{:}integer) \times (k{:}integer \times l{:}integer)$ *where* $i=k$

Then the above binding would not be of this type. But $[[i=1,j=2], [k=1,j=4]]$ would be.

Now a component of a binding can itself be a binding, so we can make the implementation of the field be a component of the implementation of vector and similarly of matrix

```
field:type ~ ...
vector:type ~ vfield:field × ...
matrix:type ~ mfield:field × ...
linearalgebra:(v:vector × m:matrix where
       (r$vfield) = (m$mfield)) → whatever;
Real:field ~ ...
Vector1:vector ~ [vfield ~ Real,...]
Matrix1:matrix ~ [mfield ~ Real,...]
Linearalgebra(Vector1,Matrix1)
```

(We have presented MacQueen's idea as a simple extension of Pebble rather than explain his notation).

Introducing such equations requires that the typechecker be able to verify that they are satisfied. According to MacQueen this can be done in a straightforward way without any general theorem proving.

Yet a third approach to sharing was embodied in the specification language Clear (Burstall and Goguen 81) in which each theory (interface) kept a record of where its components came from, thus in terms of Pebble notation

(integer × integer) × (integer × integer)

would not involve sharing, but

(t × integer) × (t × integer) where t ~ integer

would demand the same value for the first and third components. (This use of *where* does not permit substitution, and it might better be written *sharing*.)

9. Typechecking and semantics

One of our design decisions in Pebble was to take types as values. This gives a certain economy of notions, for example we need only one kind of lambda abstraction. However one may maintain the distinction between types and values as do Reynolds and Macqueen in the work already referred to. Abolishing the distinction and maintaining it raise somewhat different problems for typechecking and semantics and the respective merits of these two options are not yet clear.

Typechecking may be described either by inference rules, which allow us to deduce types for terms, or by an algorithm which evaluates the type of a term. Burstall and Lampson (1984) presents inference rules for typing Pebble, and since they are deterministic they may be re-interpreted as an algorithm. MacQueen and Sethi (1982) present inference rules for types in their language, but it is not obvious how to produce an algorithm.

Pebble is defined using the inferential form of operational semantics (Plotkin 1981), but we lack a denotational semantics. We do not have an implementation of Pebble, but we suggest how one might be obtained from the operational semantics. MacCracken (1979) gives a denotational semantics for Reynolds' form of polymorphic lambda calculus (see Fortune et al 1983 for further investigations). MacQueen and Sethi (1982) give a denotational semantics for their language with dependent types, extended by MacQueen, Plotkin and Sethi (1984). Bruce and Meyer (1984) give a denotational semantics which seems close to what one might expect for Pebble. Hook (1984) gives an operational semantics for a kernel language for Russell.

10. Conclusion

The modularity facilities of current programming languages form an extra layer of complication on top of the base language. We would like as much simplicity and flexibility as possible in tools for creating large programs in an intelligible manner. Hence we try to understand the top level by re-expressing it in a simple functional language with dependent types, capable of treating implementations as values. It is satisfying to see the fruitfulness here of ideas developed elsewhere for the logic of constructive mathematics. Theoretical insights help us to understand design possibilities for conventional programming languages and for special system building languages.

11. Acknowledgements

This paper is based largely on joint research with Butler Lampson and any merit in the ideas presented is mainly due to him. I am most grateful for the support of a BP Venture Research Fellowship during the preparation of this paper. The work on Pebble was supported mainly by Xerox Palo Alto Research Center, also by SERC, BP and DEC. I am grateful to a number of colleagues, notably to Joseph Goguen and David MacQueen for lengthy collaboration, to Gordon Plotkin, for ideas and discussions and to Eleanor Kerse for expert typing and Scribing.

References

Appelbe, W., & Ravn, A. (1984) Encapsulation constructs in system programming languages, ACM TOPLAS 6,2, 129-158.

Bruce, K. & Mayer, A. (1984) The semantics of second order polymorphic lambda calculus, in Symp. on Semantics of Data Types, Springer LNCS 173, 131-144

Burstall, R.M. (1980) Electronic Category Theory. Proc. of International Symposium on Mathematical Foundations of Computer Science(ed. Dembinski), Springer LNCS 88.

Burstall, R.M. and Goguen, J.A. An informal introduction to specification using Clear, in The Correctness Problem in Computer Science (ed. Boyer and Moore) 185-214, Academic Press.

Burstall, R.M. & Lampson, B. (1984) A kernel language for abstract data types and module,. in Symposium on Semantics of Data Types, Sophia Antipolis, (ed. Kahn et al) Springer LNCS 173, 1-50

Constable, R. & Bates, J. (1984) The nearly ultimate PRL, CS Tech. Report TR-83-551, Cornell University NY

Constable, R.L. & Zlatin, D.R. (1984) The type theory of PL/CV3. ACM Transactions on Programming Languages and Systems, 6, 94-117.

Demers, A. & Donahue, J. (1980) Datatypes, parameters and type checking. 7th ACM Symposium on Principles of Programming Languages, Las Vegas, 12-23.

Fortune, S., Leivant, D. & O'Donnell, M. (1983) The expressiveness of simple and second order type structures), JACM 30,1,151-185

Girard, J-Y. (1972) Interpretation Fonctionelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur, These de Doctorat d'etat, University of Paris.

Goguen, J., Meseguer, J. & Plaisted, D. (1982) Programming with parameterised abstract objects in OBJ, in Theory and Practice of Software Technology (ed. Ferrari et al) 163-193, N. Holland.

Gordon, M., Milner, R. & Wadsworth, C. (1979)

Edinburgh LCF. Lecture Notes in Computer Science 78, Springer-Verlag.

Hook, J. (1984) Understanding Russell - a first attempt, in Symposium on Semantics on Semantics of Data Types, Springer LNCS 173, 69-86.

Howard, W.A. (1969) The formulae-as-types notion of construction, privately circulated, published (1980) in To H.B.Curry: Essays on Combinatory Logig, Lambda Calculus and Formalism (ed Seldin and Hindley), Academic Press

Lampson, B. & Schmidt, E. (1983) Practical use of a polymorphic applicative language. ACM 10th Symposium on Principles of Programming Languages, Austin, Texas.

MacQueen, D. (1984) Modules for standard ML. ACM Symp. on LISP and Functional Prog. Austin TX 198-207

MacQueen, D., Plotkin, G. & Sethi, R. (1984) An ideal model for recursive polymorphic types. Proc. 11th ACM Symp on Princ. of Prog. Lang.

MacQueen, D. & Sethi, R. (1982) A higher order polymorphic type system for applicative languages. Symposium on Lisp and Functional Programming, Pittsburgh, PA, 243-252.

Martin-Lof, P. (1973) An intuitionistic theory of types: predicative part, in Logic Colloquium '73 (eds. H.E. Rose & J.C. Shepherdson) North-Holland, 73-118.

MacCracken, N. (1979) An investigation of a programming language with a polymorphic type structure, Ph. D. dissertation, Syracuse University, Syracuse NY

Milner, R. (1978) A theory of type polymorphism in programming. J. Computer and System Sciences, 17, 348-375.

Milner, R. (1983) A proposal for standard ML. CSR-157-83, Dept. of Computer Science, University of Edinburgh.

Mitchell, J., Maybury, W., & Sweet, R. (1979) Mesa language Manual, Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto CA

Mycroft, A. & O'Keefe, R.A. (1984) A polymorphic type system for Prolog, Art. Intell. J, 23, 295-308

Plotkin, G. (1981) A structural approach to operational semantics. Tech. Report Computer Science Dept. Aarhus Universit[illegible]enmark

Reynolds, J.C. (1974) Towards a theory of type structure. In Lecture Notes in Computer Science 19: Coll. sur la programmation, Springer-Verlag, 408-423.

Rydeheard, D.E. (1981) Applications of category theory to programming and program specification. Ph.D. thesis, University of Edinburgh.

Rydeheard, D.E. & Burstall, R.M. (1983) Draft book on Computational Category Theory, Dept. of Computer Science, University of Edinburgh.

Scott, D. (1970) Constructive validity, in Symp. Automatic Demonstration, Springer Lect. Notes in Maths 125, 237-275

Schmidt, E. (1982) Controlling Large Software Development in a Distributed Environment, Report CSL-82-7, Xerox Palo Alto Research Center, Palo Alto, CA.