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. Piling 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. C, Mesa, Modula2, ADA and MI 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 Lamport 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 Lamport and myself which is fully defined in Burstall and Lamport (1984). Indeed the present paper is mainly an attempt to motivate and give a summary account of the joint work with Lamport 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 Revi 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 Constable 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, Module 2, Mesa and ADA. (See Appel and Rezn 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;
function mkpoint(x: real; y: real): point;
procedure rotate(p: point; theta: real);

function mkpoint;
var p: point;
begin new(p);
  p.xcoord := x; p.ycoord := y;
  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
  xcoord: real;
  ycoord: real;
end;
function mkpoint(x: real; y: real): point;
procedure rotate(p: point; theta: real);

<implementation of point>
```

**Figure 2-2: Points, lines and pictures**

The point interface of Fig. 1 is in fact rather too informative. It was realized long ago that in order to maintain some abstraction about the notion of point the interface should not show the details of how the points are made of. The data type point should be opaque (Fig. 2-3), making point into an abstract data type whose implementation details are not in public view.

```
POINT

function mkpoint(x: real; y: real): point;
procedure rotate(p: point; theta: real)

function mkpoint;

procedure rotate;

```

**Figure 2-3: Points as an abstract data type**

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

```
POINT

function mkpoint(x: real; y: real): point;
procedure rotate(p: point; theta: real):

```

**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 fulfill this requirement is just an implementation. Thus we can use the usual parameter notation when writing modules. (This insight appeared in Meiss and in Clear.)

To see this let us look at an example where we
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 lessq(e1,e2:element): boolean
```

**implementation NUMERIC_ORD:ORDERING**

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

**implementation STRING_ORD:ORDERING**

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

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,8>, <grass,3>, <rain,1> would be lexically ordered <frog,3>, <grass,3>, <rain,1>, <rain,4>, <water,8>. The parameters P and Q are just required to be ORDERINGS, it is no longer necessary to list the individual types and procedures required as we did in Fig. 2-4.

**module LEXICAL_ORD(P,ORD:ORDERING)**

```
type element=record z:P.element; y:Q.element
end
function lessq(p1,p2:element):boolean
  begin if P.lessq(p1,z.p2,z then...else...
end
```

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

**interface SORTING**

```
type element
procedure sort(var array of element; m: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. 2-8). 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**


**figure 2-9: A program outline**

This is best expressed as a picture with interfaces at 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:

```plaintext
define QuSortINS=QUICKSORT(  
   LEXICAL_ORD(NUMERIC_ORD,SPRING_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 Mes 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 (Hurstall 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 endeavor 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 a function
  - or an n-tuple
  - or 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

```plaintext
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

```plaintext
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 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
\[ z = 3, \ p \sim \text{true} \]

has as its type the declaration

\[ z: \text{integer} \times p:\text{boolean} \]

(We shall see later that as well as the usual notions of \( \sim \) and \( \approx \), we will need dependent versions in the above definitions.)

Another example: the binding

\[ \text{element} \sim \text{integer}, f \sim \text{factorial} \]

has as its type the declaration

\[ \text{element: type} \times f(\text{integer} \sim \text{integer}) \]

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 a 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 ‘\( \lambda \)’, thus \( \lambda \ : \text{int} \ldots \) As usual we permit ourselves to write \( f(z:\text{int}) \cdot \ldots \) instead of the more primitive \( f \sim \lambda z: \text{int} \ldots \). Here are some examples of manipulating bindings and declarations

\[
\begin{align*}
\text{let } x: \text{int} & \sim 1 \text{ in} \\
\text{let } z: \text{type} & \sim (y: \text{int} \times p: \text{boolean}) \text{ in } \{ \text{note: } d \text{ is a decl} \\
\text{let } b : d & \sim (y \sim z + 3, p \sim \text{true}) \text{ in} \\
\text{[note: } b \text{ is a binding} \\
\text{let } b & \text{ in} \\
\text{if } p \text{ then } y \text{ else } y + 1
\end{align*}
\]

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

\[
\begin{align*}
\text{let } f(b:d) & \sim \text{let } b \text{ in} \\
\text{if } p \text{ then } y & \text{ else } y + 1
\end{align*}
\]

Here \( f \) takes an argument \( b \) 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

\[ f(b:d) = \text{if } (\text{let } b \text{ in } p) \text{ then } (\text{let } b \text{ in } y) \text{ else } (\text{let } b \text{ in } y) + 1 \]

Here \( b \text{ in } p \) 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 \( \text{stack.empty} \).

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

\[
\begin{align*}
\text{ORDERING} \\
& \text{type element} \\
& \text{function lesseq(e1, e2: element): boolean}
\end{align*}
\]

We would like to write this interface as

\[ (\text{element: type}) \times (\text{lesseq: element} \times \text{element} \rightarrow \text{boolean}) \]

When we write \((z:x) \times (y:f)\) we mean to declare the types of \( x \) and \( y \) and assume that we already know the values of \( x \) and \( f \) (which should of course be types). But here the use of \text{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 1989, Scott}
108. Martin-Lof 1973, see Constable and Zucker 1984 for a recent formulation. It was also observed by Constable and Sace 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 \( x \) and \( \to \) respectively (\( 3 \) and \( 1 \) in Martin-Lof's notation). interfaces need the first of these, dependent types.

To distinguish the dependent types from the usual ones we write it with \( \equiv \). Before \( x \) we may write not just a type but the declaration of a name which may be used after the \( \equiv \). Thus we can write

\[
\text{element} : \text{type} \equiv (\text{has} : \text{element} \times \text{element} \to \text{boolean})
\]

The general form is

\[
n : T \equiv T_2(n)
\]

where \( n \) is a name

\( T_1 \) is an expression denoting a type

\( T_2 \) is an expression denoting a type and

(only optionally) using the name \( n \).

This dependent type denotes a type whose elements are pairs, the first component being an element of type \( T_1 \) 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

\[
\begin{align*}
(\text{int}) & \equiv (\text{array}[1..n] \text{ of real}) \\
\text{elements} & [4.71260.85] \\
[3.73.51.7]
\end{align*}
\]

\[
\begin{align*}
(\text{int}) & \equiv (\text{list}() \\
\text{elements} & [\text{int}, [6,(1,2,3)]] \\
[\text{bool}, \text{true}, \text{false}]] \\
[\text{list} \times \text{int}, [(1,2), (1,3,2), (4,1)]]
\end{align*}
\]

\[
\begin{align*}
(\text{int} \to \text{type}) & \equiv (\text{int} \to \text{int}) \\
\text{element} & [\text{list}, \text{CAR}]
\end{align*}
\]

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

If \( a \) has type \( T_1 \) and \( b \) has type \( T_2(a) \)

then \( a . b \) has type \( n : T_1 \equiv T_2(n) \)

Thus in example (ii) above

\[
\begin{align*}
\text{int} & \text{ has type type} \\
[6,(1,2,3)] & \text{ has type} \times \text{list} \times \text{int} \\
\text{so } [\text{int}, [6,(1,2,3)]] & \text{ has type} (\text{int} \times \text{list}()) \equiv (\text{int} \times \text{list}())
\end{align*}
\]

Dependent cross types are not altogether unfamiliar. A textbook definition of a finite automaton might be

"An automaton is a 5-tuple \( (I, \Sigma, \delta, \gamma, S) \) where \( I \) is a set of inputs, \( \Sigma \) a set of states, \( \delta \) a set of outputs, \( \Sigma \to \delta \) is a function and \( \gamma : S \to \delta \) is a function."

We can imitate this in our typed language by the dependent cross type

\[
(\text{int} \times \text{type} \times \text{int}) \equiv ((\text{int} \to \delta) \times (\delta \to \text{int}))
\]

Here the second component is a product type, but to represent the 5-tuple definition more closely we can make it into a declaration

\[
(\text{int} \times \text{type} \times \text{int}) \equiv ((\text{int} \to \delta) \times (\delta \to \text{int}))
\]

Part of the motivation for our investigation of more expressive type systems came from our experience of coding up category theory as programs (BurSELL 1980, Rydeheard 1982, Rydeheard and Burstall 1983). Burd McDowell and I spent considerable effort converting our category programs into a more sophisticated type system; indeed we regarded 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 \( \to \) type constructor, as used for example in integer \( \to \text{boolean} \). We write it \( \rightarrow \). It enables us to introduce polymorphism, a facility which is available in some sophisticated type systems, notably the ML language (Gordon, Milner and Wadsworth 1976, Milner 1980, 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, \text{TransposeIntegerArray} \text{ and } \text{TransposeRealArray} \text{. We would like to have a single procedure } \text{TransposeArray} \text{ 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 } \text{TransposeArray} \text{ 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 } \lambda \text{ for types, analogous to } \lambda \text{ for values.)}

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

\[
\text{let at: array of int ~ TransposeArray(int)(a)}
\]

But what is the type of this function \text{TransposeArray}? Its argument is a type \text{int} \text{ and its result is a function of type array of } \text{int} \rightarrow \text{array of}
1. So we might write

\[ \text{type} \rightarrow (\text{array of } t \rightarrow \text{array of } t) \]

but then the name \( t \) is not declared. So instead we use a dependent arrow type:

\[ t.:\text{type} \rightarrow (\text{array of } l \rightarrow \text{array of } t) \]

The declaration before the \( \rightarrow \) introduces the name \( t \) which may then be used after the \( \rightarrow \).
(Reynolds would write this as \( \text{at.} (\text{array of } t \rightarrow \text{array of } t) \)).

The general form is

\[ n.T1 \rightarrow T2(n) \]

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) \( (n: \text{integer}) \rightarrow (\text{array [1..n] of real}) \)
- 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) \( (t.:\text{type}) \rightarrow (\text{list} (t.) \rightarrow \text{list} (t.)) \)
- 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 \( e \) of type \( T1 \),
  - \( f(a) \) has type \( T2(a) \)
  - then \( f \) has type \( n.T1 \rightarrow T2(n) \)

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 \( \text{TranspeseArray} \) is not given an extra parameter. Its type is

\[ \text{array of } t \rightarrow \text{array of } t \]

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

\[ (\forall t)(\text{array of } t \rightarrow \text{array of } t) \]

Now in ML \( \text{TranspeseArray} \) may be applied directly to an integer array (for example) and a type inference system is used to determine that there \( t \) 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 \( \text{ORDERING} \) becomes a declaration, the implementations \( \text{NUMERIC_ORD} \) and \( \text{STRING_ORD} \) become bindings, the module \( \text{LEXICAL_ORD} \) becomes a function from bindings to bindings, and so on.

Pebble primitive bindings have the form \( \langle \text{name} : \langle \text{type} \rangle \rightarrow \langle \text{expression} \rangle \rangle \) but the \( \text{type} \) may be omitted. Bindings to functions may include parameter names and use is instead of \( \rightarrow \). Bindings are connected by \( ; \) (sequential) or by \( | \) (simultaneous). \( f \text{element} \) is equivalent to \( let \ p \ in \ element \), except that an error occurs if \( p \) does not bind element. This example uses \( = \) but not \( \rightarrow \), 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 Module 2 or Ada, which attach the implementation to the interface by using the same name, this is difficult. In Meta or Pebble it is straightforward using the parameter mechanism, similarly in CBJ (Goguen, Meseguer and Plotz 83) 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 x element -> bool);
NUMERIC_ORD: ORDERING ~
  (element: ~ int;
   lesseq: (k: int x t: int) -> bool is ...)
end

STRING_ORD: ORDERING ~
  (element: ~ string,
   lesseq: (a: string x a: string) -> bool is ...)
end

LEXICAL_ORD: (P-ORDERING x Q-ORDERING) -> ORDERING is
  (element: ~ (a: P$element x b: Q$element);
   lesseq: (e: P$element x e: Q$element) -> bool is ...)
end

SORTING: ~ element: type =
  sort: array(element) -> array(element);
SHELLSORT: (P-ORDERING) -> SORTING is ...
end

QUICKSORT: (P-ORDERING) -> SORTING is ...
end

(LEXICAL_ORD(NUMERIC_ORD, STRING_ORD))

Figure 7-1: Sorting example in FEBBLE

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 x 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 FEBBLE we can express this by introducing
an extra parameter and parameterising vector and
matrix (this device derives from Meta).

field: type ~ ...
vector: field -> type ~ ...
PARAMETERISED INTERFACE
matrix: field -> type ~ ...
PARAMETERISED INTERFACE
linearalgebra: P:field = (vectorover(P)
  x matrixover(P)) -> whatever
MODULE

A typical call of linear algebra might be

Real:field ~ ...
IMPLEMENTATION
Vectorlower(F:field -> Vectorover(F)) ~ ...
MODULE
Matrixlower(F:field -> Matrixover(F)) ~ ...
MODULE
Linearalgebra(Real, Vectorover(Real),
Matrixover(Real))

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

(integer x integer) x (k: integer x i: integer)

A binding of this type is a \([(i=1, j=2),
(k=3, j=4)]\)

But we might add an equality constraint

(integer x integer) x (k: integer x i: 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: field x ...
matrix: field x ...
linearalgebra: (vector x matrix) where
  (r$vector) = (m$matrix) -> whatever,
Real:field ~ ...
VECTOR: vector ~ [u$field x Real,...]
MATRIX: matrix ~ [m$field x Real,...]
LINEARALGEBRA(VECTOR, MATRIX)

(We have presented MacQueen's idea as a
simple extension of FEBBLE 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 (Furthall and
Goguen 61) in which each theory (interface) kept a
record of where its components came from, thus in
terms of FEBBLE notation

(integer x integer) x (integer x integer)
would not involve sharing, but
(t x integer) x (t x 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 raises 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 Lamport (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 intentional 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. MacQueen and Sethi (1982) give a denotational semantics for Reynolds' form of polymorphic lambda calculus (see Fortune et al. 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 is both 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 testing implementions 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 Buller Lamport 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


Corden, M., Milner, R. & Wedgeworth, C. (1979)
Edinburgh LCP Lecture Notes in Computer Science 76, Springer-Verlag.


