HOPE: AN EXPERIMENTAL APPLICATIVE LANGUAGE

R.M. Burstall
D.B. MacQueen[1]
D.T. Sannella

Department of Computer Science
University of Edinburgh
Edinburgh, Scotland

ABSTRACT: An applicative language called HOPE is described and discussed. The underlying goal of the design and implementation effort was to produce a very simple programming language which encourages the construction of clear and manipulable programs. HOPE does not include an assignment statement; this is felt to be an important simplification. The user may freely define his own data types, without the need to devise a complicated encoding in terms of low-level types. The language is very strongly typed, and as implemented it incorporates a typechecker which handles polymorphic types and overloaded operators. Functions are defined by a set of recursion equations; the left-hand side of each equation includes a pattern used to determine which equation to use for a given argument. The availability of arbitrary higher-order types allows functions to be defined which 'package' recursion. Lazily-evaluated lists are provided, allowing the use of infinite lists which could be used to provide interactive input/output and concurrency. HOPE also includes a simple modularisation facility which may be used to protect the implementation of an abstract data type.

1. INTRODUCTION

As hardware gets cheaper, people become increasingly aware of the high cost of software and particularly of those components of the cost which are due to bugs in programs and to inflexible programs. A simple yet powerful programming language in which one has a good chance of avoiding mistakes and in which programs are transparent enough to be easily altered would be an effective means of countering this trend. We believe this goal is worth working towards even if it requires radical changes in our programming habits.

In this paper we will first list some aspects of programming languages which contribute to accuracy and flexibility. We illustrate them by describing an applicative language, called HOPE, which we have developed and implemented. This is followed by an example of a complete HOPE program. Finally, we discuss the strengths and weaknesses of HOPE.

---

[1]Current address: USC Information Sciences Institute, Marina del Rey, California

2. DESIRABLE PROGRAMMING LANGUAGE FEATURES

The following points seem important in a language for writing correct and flexible programs. They are listed rather briefly but they are illustrated in the description of HOPE which follows.

No assignment: referential transparency

Applicative languages which work in terms of expressions and their values, using recursion instead of loops, seem much clearer and less error-prone. The expressions are transparent in the sense that their value depends only on their textual context and not on some notion of computational history. Each variable is given a value just once where it is declared. Assignments which alter data structures are particularly prone to cause bugs; disallowing them greatly simplifies the language although it does slow down execution. Backus [2] makes this case strongly.

Maximum use of user-defined types

The user should define his own types whenever possible; thus type 'age' rather than type 'integer'. The machine must check these types. The language should allow polymorphic types so that code can be as general as possible; for example, 'list of alphas' rather than specifically 'list of numbers', where alpha is a type variable which can be instantiated to any type. We suspect that a great many errors are caused by the complications introduced when encoding data in terms of the commonly-supplied low-level types; the provision of a simple and powerful facility for defining types should greatly simplify the programmer's task.

Overloaded operators

It is convenient to use common operation symbols such as **+** or **eval** with a variety of meanings, according to the type of the arguments. The implementation should be able to disambiguate such 'overloaded' operators (this requires some subtlety for polymorphic types).

Exhaustive case analysis

Each data type will have a set of disjoint subtypes, each with a different constructor function; e.g., lists are made with **cons** or **nil** It should be easy to do case analysis with respect to these constructors and easy for a compiler to check whether the analysis is exhaustive. This

avoids mistakes like forgetting to include a test for the empty list.

Avoiding unnecessary names

It should not be necessary to introduce a large number of different names for functions to construct data structures (e.g. **cons**), to test which subtype it is (e.g. the predicate **null**), or to select components (e.g. **car** and **cdr**). We can use just the constructor name and by using pattern matching syntax let it denote testing and selecting as the context dictates.

Abolishing explicit recursion as far as possible

Given that we want an assignment-free language, we will use recursion instead of loops. However, undisciplined use of recursion is not at all desirable. It is much better to use standard constructions related to the data structures; for example, $\{ x^2 \mid x \in S \}$ as in mathematics (and SETL [12]) instead of recursion over the elements of S. In a language which allows higher-order functions (that is, procedures taking procedures as parameters), we can easily define useful higher-order functions which 'package' recursion to save writing it explicitly. An example is **mapcar** in LISP.

Making sequences into data objects

In traditional imperative-style programming, sequences are sometimes represented as successive values of a variable; thus a sequence of integers is represented as values of an integer variable. It is also possible to represent them by an array or list, but this has the disadvantage that all the elements of the sequence must be in memory at the same time. An idea aptly called 'lazy evaluation' by Henderson and Morris [18] (due originally to Wadsworth [29]) enables us to deal with such sequences as lists but have them evaluated one element at a time as needed. Amongst other things, this enables one to achieve the same effect as coroutines in an imperative language but in a more perspicacious way, by passing a lazy list as the result of a function. It also enables one to handle interactive input/output and concurrency in an applicative language.

Modularity and data abstraction

The main trick in writing large bug-free programs is to construct them in smallish pieces, each as self-contained as possible, with the pieces communicating with each other in a disciplined, explicit manner. An important technique for achieving this is data abstraction (see for example Hoare in chapter 2 of [10]). This has been explored as an aid to good programming style in such recent languages as ALPHARD [31], CLU [25] and ADA [20], which may be thought of as developing one aspect of the SIMULA class concept [11].

An abstract data type, such as 'set' or 'queue', is defined by a collection of procedures which create elements of the type and operate on them. The implementations of these procedures and the representation of the type itself are hidden from the user; his code relies solely on the specified properties of the abstract type.

Conventional block structure scope rules make it difficult to maintain the desired separation of use and implementation, so these languages adopt different rules for identifier scopes.

3. THE HOPE PROGRAMMING LANGUAGE

A programming language called HOPE which illustrates the features mentioned above has been designed and implemented at Edinburgh University. A brief informal description of HOPE follows; full details can be found in [9]. A precursor, NPL, was described in [5].

The aim throughout was to design a programming language which was very simple and encouraged clarity and manipulability of programs. Major influences in the design of HOPE have been LISP and Landin's ISWIM [24]. We were not trying to be original; we sought a judicious selection of well-understood ideas. HOPE seeks some blend of LISP power with the discipline of strong typing and modularity. It bears some resemblance to a number of other languages, including PROLOG [30], ML [17], SASL [28], OBJ [15], SCRATCHPAD [22], and languages by Burge [4] and Backus [2].

HOPE (in its present form) is an experiment in language design and a means of testing certain ideas in programming methodology rather than the ultimate in programming languages. It is still somewhat incomplete and lacks such conveniences as sensible input/output facilities (but see section 3.5).

Part of the HOPE implementation was carried out by Michael Levy and we express our appreciation of his efforts.

3.1. 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 elements of that 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, ...

To define a type 'tree-of-numbers' we could say

```
data numtree == empty ++ tip(num)
                      ++ node(numtree#numtree)
```

(the sign **#** gives the cartesian product of types). One of the elements of **numtree** is:

```
node(tip(succ(0)),
     node(tip(succ(succ(0))),tip(0)))
```

But we would like to have trees of lists and trees of trees as well, without having to define them all separately. So 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 **tree** as a parametric type is now possible:

```
data tree(alpha) == empty ++ tip(alpha)
                  ++ node(tree(alpha)#tree(alpha))
```

Now **tree** is not a type but a unary type constructor -- the type **numtree** can be dispensed with in favour of **tree(num)** .

Another example of a data declaration is

```
data graph == mkg(set vertex
                  # (vertex#vertex->truval))
```

This says that a graph is (the data constructor **mkg** applied to) a set of vertices together with a binary relation which tells if there is an edge between any two vertices.

HOPE currently comes equipped with the data types **num** , **truval** , **char** , **list** , and **set** .

3.2. 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:

```
factorial 6
```

For functions of several arguments we use tuples, formed with commas; thus 3,4 is a 2-tuple. Parentheses are used for grouping, for example:

```
g (3,4)
```

In the expression

```
(f x) y
```

the subexpression **f x** would have to produce a function; thus the types would be

$$f : T_1 \rightarrow T_2 \rightarrow T_3$$

with $x : T_1$ and $y : T_2$.

It is possible to use function symbols as infix or postfix operators if they are declared and given a precedence; for example:

```
infix +, - : 8
```

A similar form is used to assign a precedence to a prefix symbol.

Some convenient notations have been implemented for built-in types; thus $e_1::(e_2:: \ldots ::nil)$ is abbreviated $[e_1,e_2, \ldots]$, ['a','b', ...] is "ab..." and sets are written $\{e_1,e_2, \ldots\}$. Note that we write **cons** as infix :: .

There are two equivalent forms of conditional expression:

$$e_1 \text{ if } c \text{ else } e_2$$

and

$$c \text{ then } e_1 \text{ else } e_2$$

(in many languages written **if** c **then** e_1 **else** e_2).

Lambda-expressions (denoting functions) are formed as described in section 3.3.

Local variables may be introduced and associated with values using either of the equivalent forms

$$e_1 \text{ where } p == e_2$$

or

$$\text{let } p == e_2 \text{ in } e_1$$

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 a::(b::l) == f(t)
```

Upon evaluation, **f(t)** is expected to yield a value which 'matches' the pattern **a::(b::l)** . The corresponding subterms in the value of **f(t)** are then bound to **a** , **b** , and **l** while evaluating **a+b** .

3.3. DEFINING FUNCTIONS

Before a function is defined, its type must be declared. For example:

```
dec reverse : list(alpha) -> list(alpha)
```

HOPE is a very strongly-typed language, and the HOPE system includes a polymorphic typechecker (a modification of the algorithm in [27]) which is able to detect all type errors at compile time. Function symbols may be overloaded. 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 the possible argument values. This subset is described by a pattern (see section 3.2) on the left-hand side of the equation. For example:

```
--- reverse(nil) <= nil                          (1)
--- reverse(a::l) <= reverse(l) <> [a]           (2)
```

(the symbol <> is infix **append**). This defines the (top-level) reverse of a list; for example:

```
reverse(1::(2::nil)) = reverse(2::nil) <> [1]
                     = (reverse(nil) <> [2]) <> [1]
                     = (nil <> [2]) <> [1]
```

So **reverse** [1,2] = [2,1] (by two applications of equation 2 followed by a single application of equation 1). The left-hand-side patterns will normally be disjoint and be related to the structure of the type definition:

```
data list alpha == nil ++ alpha :: list alpha
```

The set of equations defining a function should exhaust the possibilities given in the **data**-statement introducing the argument types. For example, a definition of the Fibonacci numbers:

```
dec fib : num -> num
--- fib(0) <= 1
--- fib(succ(0)) <= 1
--- fib(succ(succ(n))) <= fib(succ(n)) + fib(n)
```

In this case the three patterns **0** , **succ(0)** , and **succ(succ(n))** exhaust the set of values belonging to **num** . The pattern **1** may be used as shorthand for **succ(0)** .

Nullary 'functions' may also be defined; for example:

```
dec pi : rational
--- pi <= mkrational(22,7)
```

which assumes that the type **rational** has been defined.

Lambda-expressions are defined similarly. For example, a function to compute the conjunction of two truth values (already available as the function **and**):

```
lambda true,p => p
       | false,p => false
```

Another example of a **lambda**-expression occurs in the definition of functional composition:

```
typevar alpha,beta,tau
dec compose : (alpha->beta)#(beta->tau)
                          -> (alpha->tau)
--- compose(f,g) <= lambda x => f(g(x))
```

3.4. MODULES

Any sequence of statements may be made into a module by surrounding it with the statements

```
module name
```

and

```
end
```

Data types defined in a module may be referred to outside only if a statement

```
pubtype tname
```

is included in the module. Similarly, constants (including data constructors) may be referenced only if a statement

```
pubconst cname
```

is included.

Nothing defined outside a module may be referenced within it, unless the module includes the statement

```
uses mname
```

In this case, all of the types and constants declared as public to the indicated module are available. In addition, certain global types and constants (**num** , **truval** , **char** , **list** , and **set** , together with some primitive operations) may be referenced within any module.

This is an effective tool for the encapsulation of data abstractions; if the primitive constructors and low-level operations on the data representation are not declared public, then the implementation of the abstraction is hidden from the rest of the program.

3.5. LAZY EVALUATION AND INPUT/OUTPUT

Lazy evaluation was mentioned briefly in section 2; there we described it as a desirable language feature. In HOPE, lazy evaluation has thus far only been implemented for lists, which makes available most of the power of the mechanism. Lazy lists are created using the special constructor **lcons** in place of the usual :: . This constructs a list which is identical to a normal list (in particular, the pattern **a::l** will match **lcons(c,d)**) except that the arguments to **lcons** remain unevaluated until their values are required during subsequent use of the list.

Consider the function which produces the (potentially) infinite list consisting of a number and all its successors:

```
dec allsuccs : num -> list num
--- allsuccs(n) <= lcons(n,allsuccs(n+1))
```

Now if we define a 'lazy **mapcar**' function

```
typevar alpha, beta
dec <*> : (alpha->beta)#list alpha -> list beta
infix <*> : 6

--- f <*> nil <= nil
--- f <*> (a::al) <= lcons(f(a),(f <*> al))
```

then the infinite list of the squares of all numbers is just

```
square <*> allsuccs(0)
```

We can use lazy lists to provide interactive input/output in HOPE. The first step would be for the system to provide two new functions, **input** and **output** (not yet available). **Input** takes an argument of type **device** specifying an input device, where

```
data device == tty ++ file(list(char)) ++ ...
```

and produces a lazy list where new items are read from the device when needed. A special character (control/Z) signals end of input, and consequently the end of the lazy list. **Output** is a function of type **device->(list(alpha)->void)** . Given a list, **output**(dev) evaluates each element and produces it as output on the indicated device.

As a simple example, here is a program to read a number from the terminal, output its square, and then repeat (until control/Z is typed):

```
output("tty") (square <*> input("tty"))
```

Since **input("tty")** in this expression is of type **list(num)** , the teletype will accept as input only expressions of type **num** .

This is a special example of the general problem

of communication between concurrent processes. Lazy lists provide a means of implementing communication channels, but at present the notation of HOPE is not adequate to specify an arbitrary network of communicating processes. We have not yet given much thought to this problem; however, notations exists which are sufficient and which could be adapted for our purposes (see for example [26]).

3.6. IMPLEMENTATION NOTES

The HOPE system consists of a compiler (from HOPE programs to code for an abstract stack machine) and an implementation of the target machine. The system is written in POP-2, and currently runs in approximately 55K words (plus a 15K shareable segment) on a DEC KI-10. The compiled code should be quite portable, as the abstract machine simulator could easily be written for a new machine. The compiler itself is less portable since it requires availability of POP-2.

Our preliminary tests indicate that a program written in HOPE runs approximately 9 times slower than the same algorithm coded in LISP running under the Rutgers/UCI interpreter (and 23 times slower than compiled LISP). A large program run recently ran more slowly, presumably because of page thrashing. A machine code implementation of the interpreter should run a lot faster.

4. EXAMPLES

An example of a complete HOPE program is given in figure 1. This illustrates how we can use HOPE to implement a data type (ordered trees), and then how that type can be used in a program for treesort.

Ordered trees

The first module contains an implementation of the abstract type ordered-tree-of-numbers (data type **otree** in the program). An **otree** is defined to be either **empty** , a **tip** (containing a number), or a **node** containing two **otrees** and a number. The special property of **otree** is that for any term **node**(t1,n,t2) , all numbers contained in t1 are less than n , which is in turn less than or equal to all numbers contained in t2 . We define three public constants:

empty the empty **otree**

insert adds a number to an **otree** , preserving the 'orderedness' of the **otree**

flatten inorder traversal of an **otree**

Ordinarily an abstract data type would have a few more operations; we have only included those which are used in the remainder of the program.

Note that the data constructor **node** is not public. Consequently, the only functions available to the 'outside world' for constructing and modifying **otrees** are **empty** and **insert** . Both of these preserve the properties of **otrees**, so the integrity of the implementation is assured. However, **insert** is not a data constructor, and hence may not be used in patterns.

```
module ordered_trees
  pubtype otree
  pubconst empty, insert, flatten

  data otree == empty ++ tip(num)
                      ++ node(otree#num#otree)

  dec insert : num#otree -> otree
  dec flatten : otree -> list num

  --- insert(n,empty)  <= tip(n)
  --- insert(n,tip(m))
              <= n<m then node(tip(n),m,empty)
                     else node(empty,m,tip(n))
  --- insert(n,node(t1,m,t2))
              <= n<m then node(insert(n,t1),m,t2)
                     else node(t1,m,insert(n,t2))

  --- flatten(empty)  <= nil
  --- flatten(tip(n)) <= [n]
  --- flatten(node(t1,n,t2))
                <= flatten(t1) <> (n::flatten(t2))

end


module list_iterators
  pubconst *, **

  typevar alpha, beta

  dec * : (alpha->beta)#list alpha -> list beta
  dec ** : (alpha#beta->beta)#(list alpha#beta)
                                          -> beta

  infix *, ** : 6

  --- f * nil     <= nil
  --- f * (a::al) <= (f a)::(f * al)

  --- g ** (nil,b)   <= b
  --- g ** (a::al,b) <= g ** (al,g(a,b))

end


module tree_sort
  pubconst sort
  uses ordered_trees, list_iterators

  dec sort : list num -> list num

  --- sort(l) <= flatten(insert ** (l,empty))

end
```

Figure 1: A HOPE Program

List iterators

This module defines two second-order functions which apply a given function to every element of a list and collect the results. These two functions are representatives of a group of functions which are widely used in HOPE programs in an attempt to eliminate explicit recursion as far as possible.

The first function, * , is identical to **mapcar** in LISP. It produces a list containing the results of applying the function supplied to each element of the given list. This operation is not actually used in the example. (A version of this function, producing a lazy list, was given in section 3.5.)

The function ** is slightly more complicated. When supplied with a function **g** of type **alpha#beta -> beta** , a list of **alpha**s, and an 'initial' **beta**-object, it applies **g** to each element of the list, beginning with the given **beta**-object as a second argument and subsequently recycling the result of the previous application. This operation is analogous to the 'reduction' operator of APL [21]; an example of its use would be to compute the union of a list of sets:

```
union ** (setlist,emptyset)
```

In this case, the **module** facility is used as a means of packaging a number of related functions rather than as a device for protecting a delicate abstraction. However, if one of the operations requires an auxiliary function which has no utility of its own, then it might be desirable to keep this function local to the module.

Tree sort

A function for sorting a list of numbers is now defined using the primitives developed in the preceding modules. The ** operation from **list_iterators** is used to successively **insert** the list elements into an initially empty **otree** . The result is then **flatten**ed to produce the final answer.

A number of nontrivial HOPE programs (most of them much larger than this example) have been produced. These include the following:

- Compiler for a simple language (translation of a program in [13])
- Graph isomorphism
- Implementation of the denotational semantics of a simple language (TINY from [16])
- Several abstract data types (bags, maps, ...)
- Colimits of finite diagrams in a cocomplete category, and in a kind of comma category (discussed in [6])
- Telegram problem (in [13]; problem from [19])
- Text formatter (in [13]; problem from [23])

The last two of these are written in an earlier (and syntactically incompatible) version of HOPE called NPL; translation would be tedious but straightforward.

5. DISCUSSION

Some desirable features for programming languages were discussed in section 2. Since HOPE was designed with these in mind, it not surprisingly displays them all, although some are exhibited better than others (an exception is set expressions [12], still to be implemented). But more important is the question of whether the resulting language provides a better interface with the computer than the one presented by existing languages.

Probably the most pleasing result of our efforts has been the observation that it really is significantly easier to construct programs in HOPE than in any other programming language we know. In particular, we have found that it is rather easy to write programs which are absolutely correct the first time they are run. It seems quite difficult to commit an error which remains undiscovered for long -- the simple errors are caught during compilation by the typechecker, while the more fundamental errors (stemming usually from an insufficient understanding of the problem) display themselves glaringly during even a casual test. Since testing and debugging account for more than half the cost of many software projects, this could yield enormous savings.

An important aim of language design is to make it easier to verify that a program meets a given specification. In this respect applicative languages such as HOPE seem to offer considerable advantages; the absence of assignment statements and the consequent replacement of iteration by recursion gives programs a simple and easy to analyse form. Powerful verification systems for applicative languages have been written by Boyer and Moore [3] and by Aubin [1].

Another advantage of an applicative language is the fact that programs lend themselves very well to the technique of program transformation [7], whereby a simple but inefficient program is transformed into an acceptably efficient one by steps which maintain its correctness. A very simple example of program transformation would be the production of the following linear-time program for generating Fibonacci numbers from the equivalent program in section 3.3 which requires exponential time.

```
dec g : num -> num#num
--- g(0) <= 1,1
--- g(succ(n)) <= (a + b),a   where a,b == g(n)

dec fib' : num -> num
--- fib'(0) <= 1
--- fib'(1) <= 1
--- fib'(succ(succ(n)))
          <= a + b   where a,b == g(n)
```

Feather [13] has produced a system for transforming large programs, which is connected to an earlier version of the HOPE system.

A very high-level language such as HOPE pays penalties of inefficiency because it is remote from the machine level. It could be thought of as a specification language in which the specifications are 'walkable' (if not 'runnable'), or as a

language for making a first try at a programming project. But the program transformation approach discussed above gives us some hope that we can produce tolerably efficient programs with less effort than in a conventional language.

In addition, there is another (as yet unmentioned) advantage of applicative languages which may come to our rescue: applicative languages are not so tightly bound to the notion of a sequential machine as are imperative languages. The value of the function application

$$e_0(e_1, \ldots, e_n)$$

is independent of the order of evaluation of the expressions $e_0, \ldots, e_n$ (if parameters are passed 'by value'); this is guaranteed by the absence of an assignment statement. If a parallel machine is available, $e_0, \ldots, e_n$ may be evaluated simultaneously. Not only that, but if $e_0, \ldots, e_n$ are themselves function applications, then their arguments may all be evaluated simultaneously. With the rapidly falling cost of hardware, this is feasible, although we would have to build the machine ourselves.

HOPE has faults, too; one is illustrated in figure 1. The sorting program will only sort a list of numbers, because **otree** is 'ordered-tree-of-numbers'. We want a more general sorting program, and this depends on a more general definition of ordered trees; we would like to define 'ordered-tree-of-alphas'. The **data** declaration is easy to generalise. But to generalise **insert** to type

alpha#otree(alpha)->otree(alpha)

we must have a more general order relation than < , which is defined only for numbers.

One solution to this problem would be to require **otree** users to supply the appropriate order relation explicitly when dealing with an **otree** . It could be added as an extra argument to **insert** , or alternatively it could be built into the **otrees** themselves (supplied as an argument to **empty** , and propagated to new tips and nodes by **insert**). Unfortunately, a 'bad' order relation (for example, one which is not transitive) would violate the integrity of the data type, causing unpredictable results.

Another possibility would be to define a general order relation as a new HOPE primitive. This was our solution to the analogous problem involving the equality relation = . Two data values are equal if they have the same representation as terms of constructors, except that for a newly-defined data type the user may provide a nonstandard equality which is automatically incorporated into the standard system equality. For example, we might want to define equality for **otrees** so that **tip(n) = node(empty,n,empty)** . Allowing an order to be associated with each type would be a satisfactory solution to our present difficulties. But how many more such operations should be associated with a type?

The best solution is of course to associate a collection of operations with each data type (so types become algebras instead of simply sets). Rather than generalising to **otree(alpha)** we could generalise to **otree(alpha[<])** , requiring an order relation to exist on the parameter type. This is the approach taken in CLU [25] and in the specification language CLEAR [8]. We really want HOPE modules to have parameters, a collection of types and operators, just as CLU clusters have parameters.

As a further example, refer again to figure 1 and note that the module **tree_sort** does not depend on the fact that **otrees** are trees, but just on certain properties of **insert** and **flatten** . We may substitute a module **ordered_lists** for **ordered_trees** , where **empty** becomes **nil** , **insert** becomes the obvious order-preserving insertion in an ordered list, and **flatten** is the identity function. Essentially, **tree_sort** is a parametrised module which may be 'applied' to any module satisfying certain (nontrivial) properties.

The CLU cluster parameter, however, is just an explicit list of types and operators,

T with < : T#T -> truval

In CLEAR such an entity (called a theory and provided with axioms) can be named, thus 'Ordered-Set'; what is more various operations for building such theories are provided. We would like to extend HOPE to have parameterised modules where the parameters and the interfaces between modules are nameable and manipulable as in CLEAR. Further work in this direction is being carried out in collaboration with J.A. Goguen [14]. We hope it will contribute to a better understanding of the structure of large programs and their development.

Acknowledgements

We would like to thank the Science Research Council and Edinburgh University for supporting this work. Michael Levy did part of the implementation. Robin Milner and Michael Gordon showed us how to handle polymorphic types. John Darlington helped develop NPL, the precursor of HOPE. Peter Landin's influence was pervasive.

REFERENCES

1. Aubin, R. Strategies for Mechanizing Structural Induction. Proc. 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts, August, 1977, pp. 363-369.

2. Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Comm. ACM 21, 8 (August 1978), 613-641.

3. Boyer, R.S. and Moore, J.S. A Computational Logic. Academic Press, 1980.

4. Burge, W.H. Recursive Programming Techniques. Addison-Wesley, 1975.

5. Burstall, R.M. Design Considerations for a Functional Programming Language. Infotech State of the Art Conference: The Software Revolution, Copenhagen, October, 1977.

6. Burstall, R.M. Electronic Category Theory. Proc. 9th Int. Symp. on Mathematical Foundations of Computer Science, Rydzyna, Poland, September, 1980.
7. Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs. J. ACM 24, 1 (January 1977), 44-67.
8. Burstall, R.M. and Goguen, J.A. Putting Theories Together to Make Specifications. Proc. 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts, August, 1977, pp. 1045-1058.
9. Burstall, R.M. and Sannella, D.T. HOPE User's Manual. In preparation.
10. Dahl, O-J., Dijkstra, E.W. and Hoare C.A.R. Structured Programming. Academic Press, 1972.
11. Dahl, O-J., Myhrhaug, B. and Nygaard, K. The SIMULA 67 Common Base Language. Publication S22, Norwegian Computing Centre, Oslo, 1970.
12. Dewar, R.B.K., Grand, A., Liu, S-C. and Schwartz, J.T. Programming by Refinement, as Exemplified by the SETL Representation Sublanguage. ACM Trans. Programming Languages and Systems 1, 1 (July 1979), 27-49.
13. Feather, M.S. A System for Developing Programs by Transformation. Ph.D. Th., University of Edinburgh, 1979.
14. Goguen, J.A. and Burstall, R.M. CAT, a System for the Structured Elaboration of Correct Programs from Structured Specifications. In preparation.
15. Goguen, J.A. and Tardo, J.J. An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications. Specifications of Reliable Software Conf. Proc., Cambridge, Massachusetts, April, 1979.
16. Gordon, M.J.C. The Denotational Descripion of Programming Languages. Springer-Verlag, 1979.
17. Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey, M. and Wadsworth, C. A Metalanguage for Interactive Proof in LCF. Proc. 5th ACM Symp. on Principles of Programming Languages, Tucson, Arizona, 1978.
18. Henderson, P. and Morris, J. A Lazy Evaluator. Proc. 3rd ACM Symp. on Principles of Programming Languages, Atlanta, Georgia, 1976, pp. 95-103.
19. Henderson, P. and Snowdon, R. An Experiment in Structured Programming. BIT 12, 1 (1972), 38-53.
20. Ichbiah, J.D. et al. Preliminary ADA Reference Manual. SIGPLAN Notices 14, 6A (June 1979).
21. Iverson, K. A Programming Language. John Wiley and Sons, 1962.
22. Jenks, R.D. The SCRATCHPAD Language. Proc. Symp. on Very High Level Languages, April, 1974.
23. Kernighan, B.W. and Plauger, P.J. Software Tools. Addison-Wesley, 1976.
24. Landin, P.J. The Next 700 Programming Languages. Comm. ACM 9, 3 (March 1966), 157-166.
25. Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. Abstraction Mechanisms in CLU. Comm. ACM 20, 8 (August 1977), 564-576.
26. Milne, G. and Milner, R. Concurrent Processes and Their Syntax. J. ACM 26, 2 (April 1979), 302-321.
27. Milner, R. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17, 3 (December 1978), 348-375.
28. Turner, D.A. SASL Language Manual. University of St. Andrews, 1979.
29. Wadsworth, C.P. Semantics and Pragmatics of the Lambda Calculus. Ph.D. Th., Programming Research Unit, Oxford University, 1971.
30. Warren, D.H.D, Pereira, L.M. and Pereira, F.C.N. PROLOG -- The Language and Its Implementation Compared With LISP. Proc. ACM Symp. on Artificial Intelligence and Programming Languages, Rochester, New York, August, 1977.
31. Wulf, W.A., London, R.L. and Shaw, M. An Introduction to the Construction and Verification of Alphard Programs. IEEE Trans. on Software Eng. SE-2, 4 (December 1976), 253-265.