Technical Report

UCAM-CL-TR-99
ISSN 1476-2986

Number 99

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

An overview of the Poly programming language

David C.J. Matthews

August 1986

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

ISSN 1476-2986

An Overview of the Poly Programming Language

David C.J. Matthews

12 August 1986

Poly is a general purpose programming language based on the idea of treating types as first-class values. It can support polymorphic operations by passing types as parameters to procedures, and abstract types and parameterised types by returning types as results.

Although Poly is not intended specifically as a database programming language it was convenient to implement it in a persistent storage system. This which allows the user to retain data structures from one session to the next and has allowed large and can support large programming systems such as the Poly compiler and a Standard ML system.

1 Poly and its Type System

Poly[Mat85] is based on the idea of types as first class values first used in the language Russell.[Dem79] In the terms used by Cardelli and MacQueen[Car85] it uses the *abstract witness* model of a type. Treating a type this way means that polymorphism, parameterised types and modules are all handled by the general concept of function application.

1.1 Types as Values

A type in Poly is a set of values, normally functions. For example the type *integer* has operations +, − etc. Other types may have these operations, the type *real* also has + and − but will not have a *mod* (remainder) operation. The operations need not be functions, *integer* also has *zero*, *first* and *last* which are *simple values*, and other types may contain types.

All values in Poly have a *signature*, called a *specification* in earlier reports, which is only used at compile-time. It is the analogue of a type in languages like Pascal and corresponds in many ways to the idea of a type in Ponder[Fai85]. There are three classes of value in Poly, the *simple value* which corresponds to what are normally thought of as values in, say Pascal, numbers, strings, vectors etc.; the *procedure* or function which operates on values and the *type* which is a set of values. Each kind of value has a signature.

To show why this view of types is useful we will consider some properties of other languages, and how they are handled in Poly.

1.2 Polymorphism

A polymorphic function is one that can be applied to values of many different types. The phrase is sometimes used where *overloading* would be more appropriate, for example the + operator in Pascal. In Pascal, or languages like it, there are operators which can be applied to values of more than one data type and their meanings are different according to the type of their arguments. They can be thought of as a set of overloaded operators in the same way as operators in Ada can be overloaded. Truly polymorphic functions are somewhat different. They are functions which are applicable to values of a wide variety of data types, including types which may not exist at the time the function is written. The fundamental difference is that a new polymorphic function can be written in terms of other polymorphic functions, while a function written in terms of overloaded functions must be defined for each data type even if the program is the same for each. For example

```
function min(i,j: integer):integer
    begin
    if i < j then min := i else min := j
    end;
function min(i,j: real):real
    begin
    if i < j then min := i else min := j
    end;
```

The ML [Mil84] programming language provides polymorphic operations on an all-or-nothing basis. This allows one to write an identity function which simply returns its argument, and this function is applicable to values of any type. One can also write functions which operate on lists of any type or on functions of any type. This generally works very well but has problems when one wants to write an operation which operates differently on different data types. For example it is still necessary to overload = since comparing two integers is different to comparing two lists of real numbers. The *min* function cannot be written as a single function in ML. What is required is a way of writing operations which are *type-dependent.*

A type in Poly is characterised by the operations it has. Both *real* and *integer* have < operations though they will be implemented in different ways. Many other types may have < operations since Poly allows the user to make new types. Poly allows a function to be written which selects certain operations from a type and values of any type with those operations can be used as a parameter. For example there is a *single* < function which works on types which have a < operation and simply applies the operations to the arguments. The effect is as though < were being overloaded. However, we can write a function in terms of this, such as the *min* function. This will also work on values of any type which has a < operation. For example, *min* is a function which will work on values of any type with the < operation. Such a type has signature

```
type (t) < : proc(t:t)boolean end
```

This type has an operation, <, which takes two values and returns a *boolean*. We will first write a version of *min* which takes three parameters; a type and two values of this type and returns a value of the type. It has signature:

```
proc(t: type (t) < : proc(t:t)boolean end: t; t)t
```

We can write the whole function.

```
let min ==
proc(t: type (t) < : proc(t: t)boolean end; x, y: t)t
    begin
    if x < y then x else y
    end;
```

It can be applied to integer values

```
min(integer, 1, 2)
```

or string values

```
min(string, "abc", "abd")
```

or values of any type with a < operation.

The first parameter is a type which must have a < operation which compares two values of the type, and the second and third parameters must be values of the type. When we call

```
min(integer, 1, 2)
```

the actual parameters are matched to the formal parameters from left to right. First the types are matched by checking that the type given has the appropriate operation, and then the values are matched. They are not of course the same type as t, since they have type *integer*, but we invoke a matching rule which says that if we have matched an actual type parameter to a formal type then we can match values of corresponding types. In addition the type of the result becomes matched so that the result has type *integer*. This can be thought of as a systematic renaming of t with *integer*.

1.3 Implied Parameters

Having to pass the types explicitly is often a nuisance so there is a sugared form which gives a way of omitting the types and having the compiler insert them automatically using the types of the parameters. The only difference to the definition of the function is that the types are written in square brackets before the other parameters. The definition of *min* would then be:

```
let min ==
proc[t: type (t) < : proc(t; t)boolean end] (x, y: t)t
    begin
    if x < y then x else y
    end;
```

It can be used by just giving the values.

```
min(1, 2);
min("abc", "abd");
```

This sugaring also allows us to define operators such as + and < which simply apply the operation with the same name from the types of their arguments giving the effect of overloading.

```
let + ==
proc infix 6 [t: type (t) + : proc(t; t)t end] (x, y: t)t
    begin
    t$+ (x, y)
    end;
```

2 Parameterised Types

So far we have seen how having types as parameters to a procedure allows us to write polymorphic operations. Types can also be returned from procedures and this provides a way of defining types which are parameterised by either types or values. As an example, suppose we wanted to construct an associative memory in which to store values of arbitrary type together with a number which would identify each. This could be defined as follows

```
let associative ==
proc(element: type end)
    type (assoc)
    enter: proc(assoc; integer; element)assoc;
    lookup: proc(assoc; integer)element;
    empty: assoc
    end
    begin
        type (assoc)
        extends struct(next: assoc; index: integer; value: element);
        let empty == assoc$nil;
        let enter ==
        proc(table: assoc; num: integer; val: element)assoc
            begin
            assoc$constr(table, num, val)
            end;
        letrec lookup ==
        proc(table: assoc; num: integer)element
            begin
            if table = assoc$nil
            then raise not_found
            else if table.index = num
            then table.value
            else lookup(table.next, num)
            end
        end
    end;
```

This is a very simple minded definition but it illustrates the point. We start by giving the header of the procedure which includes the signature of the argument, in this case that *element* is a type but that any type will do, and the signature of the result. The result is a type with three objects, a value which denotes the empty table and procedures to enter and look up items from the table. It is implemented in terms of a **struct** (a record with a *nil* value and equality) which makes up a list of index/value pairs. *enter* just returns a new list with the new pair "cons-ed" onto the front[1]. A better implementation would check to see if there was already an entry with that index and return a list with the old entry replaced by the new one. *lookup* searches the list for an entry with the required index and either returns the value or raises an exception.

There is no particular reason why we should use integers as the indexing value, it would be perfectly possible to use any type which had an equality operation. The procedure header would then be

```
proc(element: type end;
index_type: type (i) = : proc(i;i)boolean end)...
```

with *integer* replaced everywhere in the body by *index_type*.

[1]We could have written simply

```
let enter == assoc$constr;
```

since the arguments are in the same order

A more efficient implementation for index types with an ordering would be to use binary trees rather than lists. We would then have to add a $>$ or $<$ to *index_type*, or at least replace the $=$ by one of these. Now, since types are values we could incorporate an if-statement into the procedure and use one or other of the implementations depending on the value of a further parameter. We might want to do this because one implementation may be more efficient for, say, small tables and the other for larger ones. For the example we will assume a parameter *use_binary_tree*. The procedure will now look something like this.

```
proc(element: type end;
index_type: type (i) = , < : proc(i;i)boolean end;
use_binary_tree: boolean)...
begin
if use_binary_tree
then
    type .... { Binary tree implementation }
    end
else
    type .... { List implementation }
    end
end
```

This could now be called as

```
let a_table == associative(string, integer, true);
let another_table == associative(string, integer, size > 30);
```

In the second case the expression may not be able to be evaluated when the call to the procedure is compiled, *but this does not matter.* We do not know at compile-time which of the two implementations of the type will be used, but we know that either of them have all the operations required so they will do equally well.

There is however a problem with this idea of types which this example shows quite nicely. Since the expression may not be evaluated at compile-time how do we know when two values have the same type? The type system must ensure that we apply the *lookup* procedure which understands the representation of the particular associative memory. It would be catastrophic to try to look up a value assuming that the value represented a tree when it was in fact a list. We need the type system to assure us at compile-time that the expressions

```
let y == X$enter(X$empty, 1, "hello");
X$lookup(y);
```

where X stands for a type or type-returning expression, will not give faults at run-time because of a mistake in interpreting the representations.

There are several possible approaches to the problem of which Poly and Russell illustrate two. In Russell values can have types such as

associative(*string*, *integer*, *size* > 30)

provided nothing in the expression involves a global variable[2]. This essentially means that all functions have to be "variable-free", not just those which directly return types. Given this restriction it is possible to say that if two expressions are syntatically the same in a given context then they return the same value. If however, *size* were a variable, or *associative* looked at the value of a global variable, then we could not say with certainty that two values with type

associative(*string*, *integer*, *size* > 30)

had the same type. Taking a purely synatactic view means that expressions like

associative(*string*, *integer*, 2 > 1)

and

associative(*string*, *integer*, *true*)

are not the same type.

In Poly types are only regarded as the same if they are the same *named* type. So while values with types which are expressions can sometimes be produced there is very little that can be done with them. To be useful a type-returning expression has to be bound to an identifier.

```
let a_table == associative(string, integer, true);
let a_val == a_table$enter(a_table$empty, 1, "hello");
let another_table == associative(string, integer, true);
let another_val == another_table$enter(another_table$empty, 1, "hello");
```

a_val and *another_val* have distinct types *a_table* and *another_table*.

A side-effect of this is that "types" such as

[2]Variable in this context means something whose value can be changed by assignment.

list(*integer*)

cannot be used directly. We have to write

let *int_list* == *list*(*integer*);

and then use *int_list* as the type. However this is not such a problem as might at first appear. Since we can write functions which take implied parameters we can write an *append* function which will work on values of any type with the appropriate *hd*, *tl* etc., irrespective of their actual implementations.

3 Modules

A module is conventionally thought of as a collection of types and functions which can be separately compiled. It has an interface which is the types of these functions so that other modules can make use of it without having to know the precise implementation.

Types in Poly can be thought of in the same way. A type is a collection of operations and its signature gives their "types"[3]. A module which makes use of other modules, *imports* them in conventional terms, can be represented as a procedure which is applied to types and returns a type. One of the big advantages of this view of modules is that binding modules together is done using statements written in Poly and type-checked using the normal Poly type-checker. There is no need, as with MESA and C-MESA[Mit79] for a separate module binding language.

The module system for ML[Har85] is essentially a system built on top of the kernel language. *Structures* and *functors* correspond to values and functions in the kernel but the ML type system makes it impossible to unify these concepts.

4 Persistence in Poly

Poly is an interactive system in which the user types expressions and declarations and these are compiled and executed immediately. When objects are declared they are added to the objects the system knows about and they can be used in subsequent expressions. Such systems are quite common and usually work on a

[3]We usually think of a type as being something like *integer* which has values, but a type in Poly can be any collection of objects. So a collection of floating point functions *sin*, *cos* etc. could be combined as a type even though there is no such thing as a value of this type.

core image which can be saved from one session to the next. This is fine provided that the core image does not grow too big. However as the core image gets larger the costs of reading it in and writing it out get more serious. Also the cost of garbage-collection rises. There is a further question about the security of the data if the machine crashes while writing out a large image.

For these reasons Poly is implemented in a persistent store[Atk81a][Atk81b] which can be thought of as a core image where objects are only read in when they are actually required. The cost of loading objects from the image, or database, depends on the amount of the store which is used by a program rather than the total size of the image. A simple transaction mechanism ensures that the database remains in a consistent state in the event of a machine crash or if the program is killed halfway through writing out. Some experiments have been done on using multiple databases so that large programs such as the compiler can be shared between several users.

Using this persistent store the Poly compiler has been boot-strapped so that it is just another procedure. A Standard ML compiler has also been written which uses the same back-end as the Poly compiler.

In a typical interactive programming system there is a single name space for all identifiers, but as the number of declarations have grown it has become necessary to divide up the name space into separate *environments*. An environment is very similar to a directory in a filing system or to a block in a programming language. When an environment is selected all new identifiers are entered into it and looked up in it. There is the equivalent of the scope rules in a programming language so that an identifier is looked up in a series of nested environments until it is found. It could be thought of as a Poly type since it is a collection of objects, but it cannot be quite the same because declarations can be added or removed dynamically to an environment while a Poly type must be "frozen".

5 Conclusions

Poly was designed as a general purpose language and has been used successfully for some medium scale projects (there is about 20000 lines of code in the Poly and ML compilers). After some years of programming in it the type system has proved to work very well. Treating types as first-class values seems to result in a generally simpler language than languages where types are treated as purely compile-time objects. Experience with Standard ML suggests that pattern-matching and exceptions with parameters (exceptions in Poly cannot carry parameters) are something that should be added. Some kind of type inference based on unification could be used to reduce the amount of type information which must be given explicitly, though it cannot remove it completely.

The presence of a persistent store tends to break down the distinction between compile-time and run-time, since the compiler is just another function to be applied. Compile-time does have some meaning in this system however. Compiling an expression means checking the interfaces between functions and their arguments so that the result can be guaranteed not to produce a type-checking error later on. If we compile a procedure then we want to produce a type for the procedure as a whole and remove the type information within it. Not only does this improve the efficiency of the procedure but it also gives us a degree of certainty that the procedure will not fail. It is a little way along the road to proving the correctness of the procedure. There is a cost in this static type checking in Poly in that some procedures which are in fact type-correct will fail to pass a static type-checker, but the advantages of static type-checking more than outweigh the disadvantages.

References

[Atk81a] Atkinson M.P., Chisholm K.J. and Cockshott W.P. "PS-Algol: An Algol with a Persistent Heap." Technical Report CSR-94-81, Computer Science Dept., University of Edinburgh.

[Atk81b] Atkinson, M.P., Bailey P., Cockshott W.P., Chisholm K.J. and Morrison R. "Progress with Persistent Programming." Technical Report PPR-8-81, Computer Science Dept., University of Edinburgh.

[Car85] Cardelli L. and MacQueen D. "Persistence and Type Abstraction." Proc. of the Persistence and Data Types Workshop, August 1985.

[Dem79] Demers A. and Donahue J. "Revised Report on Russell." TR 79-389 Dept. of Computer Science, Cornell University.

[Fai85] Fairbairn J. "A New Type-Checker for a Functional Language." Proc. of the Persistence and Data Types Workshop, August 1985.

[Har85] Harper R. "Modules and Persistence in Standard ML." Proc. of the Persistence and Data Types Workshop, August 1985.

[Mat85] Matthews D.C.J. "Poly Manual" SIGPLAN Notices. Vol.20 No.9 Sept. 1985.

[Mil84] Milner R. "A Proposal for Standard ML" in "Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming". Austin, Texas 1984.

[Mit79] Mitchell James G. et al. "MESA Language Manual." XEROX PARC, 1979