Technical Report

UCAM-CL-TR-29
ISSN 1476-2986

Number 29

Introduction to Poly

D.C.J. Matthews

May 1982

ISSN 1476-2986

INTRODUCTION TO POLY

D.C.J. Matthews, May 1982

Computer Laboratory,

University of Cambridge

Abstract

This report is a tutorial introduction to the programming language Poly. It describes how to write and run programs in Poly using the VAX/UNIX implementation. Examples given include polymorphic list functions, a double precision integer package and a subrange type constructor.

INTRODUCTION TO POLY

Poly is a programming language which supports polymorphic operations. This document explains how it is used on the VAX.

1. Commands and Declarations

The system is entered by running the appropriate program (e.g. **/mnt/dcjm/poly** at Cambridge). The compiler will then reply with a prompt (>). To exit from Poly at any time type ctrl-D (end-of-text) or ctrl-C (interrupt). There are three types of instructions which can be typed to Poly; declarations of identifiers, statements (commands), or expressions. An example of a command and the output it produces is

```
> print("Hello");
Hello
```

Note the closing semicolon which must be present to indicate the end of the command. If you forget it the compiler will print a # as a prompt to indicate that the command is not yet complete.

An example of an expression is

```
> "Hi";
Hi
```

Poly prints the value of an expression without the need to type the word 'print'.

Commands can be grouped by enclosing them with the bracketting symbols **begin** and **end** or **(** and **)**. For instance

```
> begin
#   print("Hello");
#   print(" again")
# end;
Hello again
```

Any object in Poly can be bound to an identifier by writing a declaration. For instance

```
> let message == "Hello ";
```

declares an identifier 'message' to have the value of the string 'Hello '. It can be printed in the same way as the string constant.

```
> message;
Hello
```

Names can be either a sequence of letters and digits starting with a letter, or a sequence of the special characters + - * = < > etc. Certain names are reserved to have special meanings and cannot be used in declarations. Those words can be written in upper, lower or mixed case, all other words are considered to be different if written in different cases. When declaring a name made up of the special characters remember to put a space between the name and the == or colon which follows it. Comments are enclosed in curly brackets { and }. They are ignored by the compiler and are equivalent to a single space or newline between words.

2. Procedures

Statements or groups of statements can be declared by making them into procedures.

```
> let printmessage ==
#    proc()
#       (print("A message "));
```

A procedure consists of a procedure header (in this case the word **proc** and parentheses **(** and **)**) and a body. The procedure body must be enclosed in bracketting symbols (in this case '(' and ')') even if there is only one statement.

This is simply another example of a declaration. Just as previously 'message' was declared to have the value "Hello ", 'printmessage' has been declared with the value of the procedure.

The procedure is called by typing the procedure name followed by **()**.

```
> printmessage();
A message
```

The effect of this is execute the body of the procedure and so print the string.

Procedures can take arguments so that values can be passed to them when they are called.

```
> let pmessage ==
#    proc(m : string)
#       begin
#       print("The message is :");
#       print(m)
#       end;
```

This can be called by typing

```
> pmessage("Hello");
The message is :Hello
```

or by typing

```
> pmessage("Goodbye");
The message is :Goodbye
```

3. Specifications

As well as having a value all objects in Poly have a specification, analogous to a type in other languages. It is used by the compiler to ensure that only meaningful statements will be accepted. You can find the specification of a declared name x by typing **? "x";**.

```
> ? "message";
message : string
```

This means that message is a constant belonging to the type 'string'.

```
> ? "pmessage";
pmessage : PROC(string)
```

This means that pmessage is a procedure taking a value of type string as its argument. Since message has that specification the call

```
> pmessage(message);
The message is :Hello
```

will work. Likewise the call

```
> pmessage("Hi");
The message is :Hi
```

will work because "Hi" also belongs to type string. However

```
> pmessage(pmessage);
Error - specifications have different forms
```

will fail because 'pmessage' has the wrong specification. Incidently, the specification of the procedure is the same as the header used when it was declared, ignoring the differences in the case of some of the words.

4. Integer and Boolean

So far the only constants used have been those belonging to the type string. Another type, **integer** provides operations on integral numbers.

```
> print(42);
42
```

The usual arithmetic operations +, -, *, div, mod, succ and pred are available.

```
> 42+10-2;
50
```

However, unlike other languages all infix operators have the same precedence so

```
> 4+3*2;
14
```

prints 14 rather than 10. Also - is an infix operator only, there is a procedure neg which complements its argument.

Another 'standard' type is **boolean** which has only two values **true** and **false**. Its main use is in tests for equality (the = operator), inequality

(<>) and magnitude (> < >= <=).

```
> let two == 2;
> 1 = two;
false
> 2 = two;
true
> 3 <> 4;
true
> 4 >= 5;
false
```

The expression '1 = two' has type boolean. Identifiers can be declared to have boolean values in the same way as integers and strings.

```
> let testtwo == two > 1;
```

declares testtwo to be 'true' since 'two' is greater than 1. There are three operators which work on boolean values, **&**, **|** and **~**. **~** is a prefix operator which complements its argument (i.e. if its argument was false the result is true, and vice-versa). **&** is an infix operator which returns true only if both its arguments are true. **|** is also an infix operator which returns true if either of its arguments is true.

5. If-Statement

Boolean values are particularly useful since they can be tested using **if**. The if-statement causes different statements to be obeyed depending on a condition.

```
> if two = 2
# then print("It is two")
# else print("It isn't two");
It is two
```

tests the value of the expression 'two = 2' and executes the statement after the word **then** if it is true, and the statement after the word **else** if it is false. This could be written as a procedure,

```
> let iszero ==
#  proc(i: integer)
#     (if i = 0 then print("It is zero")
#     else print("It isn't zero"));
```

which could then be called to test a value.

```
> iszero(4);
It isn't zero
```

since 4 is not zero. If-statements can return values as well as perform actions in the then and else parts. An alternative way of writing 'iszero' could have been

```
> let iszero ==
#  proc(i: integer)
#     (print(
#        if i = 0
#        then "It is zero"
#        else "It isn't zero"
#          ));
```

This version tests the condition, and returns one or other of the strings for printing. This can only be used if both the then and else parts return values with similar specifications (in this case both sides return string constants). The version of the if-statement which does not return a value can be written with only a then-part. If the then-part returns a value there must be an else-part (otherwise what value would be returned if the condition were false?).

6. More on Procedures

Procedures can be written which return results. For instance a further way of writing 'iszero' would be to allow it to return the value of the string.

```
> let iszero ==
#  proc(i: integer)string
#    (if i = 0 then "It is zero"
#     else "It isn't zero");
> ? "iszero";
iszero : PROC(integer)string
```

Calling it would then cause it to return the appropriate string which would then be printed.

```
> iszero(0);
It is zero
```

Another example is a procedure which returns the square of its argument.

```
> let sqr ==
#    proc(i: integer)integer (i*i);
```

declares sqr to be a procedure which takes an argument with type integer and returns a result with type integer. The body of the procedure evaluates the square of the argument i, and the result is the value of the expression. The call

```
> sqr(4);
16
```

will therefore print out the value 16.

Procedures in Poly can be written which call themselves, i.e. recursive procedures. These are declared using **letrec** rather than **let**.

```
> letrec fact ==
#    proc(i: integer)integer
#      (if i = 1 then 1
#       else i*fact(i-1));
```

This is the recursive definition of the factorial function. The procedure can be called by using

```
> fact(5);
120
```

which prints the result. **letrec** has the effect of making the name being declared available in the expression following the ==, whereas **let** does not declare it until after the closing semicolon.

7. Variables

Constants are objects whose value cannot be changed. There are also objects whose value can change, these are variables. Variables are created by declarations such as

```
> let v == new(0);
```

The procedure 'new' returns a variable whose initial value is the argument.

```
> v;
0
```

A new value can be given to v by using the assignment operator.

```
> v := 3;
> v;
3
```

Thus v now has the value 3. The new value can depend on the old value.

```
> v := (v+2);
```

Sets the value to be 5. The parentheses are necessary because otherwise the order of evaluation would be strictly left-to-right. Variables can be of any type.

```
> let sv == new("A string");
```

declares sv to be a string variable. The specification of a variable is not as simple as it may seem and will be dealt with later.

8. The While Loop

It is often necessary to repeat some statements more than once. This can be done using the **while** statement. For instance

```
> let x == new(10);
> while x <> 0
# do
#    begin
#    print(x*x);
#    print(" ");
#    x := pred(x)
#    end;
100 81 64 49 25 16 9 4 1
```

prints the square of all the numbers from 10 down to 1. The body of the loop (the statement after the word **do**) is executed repeatedly while the condition (the expression after the word **while**) is true. The condition is tested before the loop is entered, so

```
> while false
# do print("Looping");
```

will not print anything.

9. Operators

We have already seen examples of operators such as + and &. In Poly operators are just procedures whose specifications include the words **infix** or **prefix**. They are declared in a similar way to procedures, for instance

```
> let sq == proc prefix (i : integer)integer (i*i);
```

has declared sq as a prefix operator. It can be used like any other prefix operator:

```
> sq 3;
9
```

The difference between a prefix operator and other procedures is that the argument to a prefix operator does not need to be in parentheses. Infix operators can be defined similarly.

10. The Specifications of Types

All objects in Poly have specifications. This includes types such as string, integer and boolean.

```
> ? "boolean";
boolean : TYPE (boolean)
   & : PROC INFIX (boolean; boolean)boolean;
   false : boolean;
   print : PROC (boolean);
   true : boolean;
   | : PROC INFIX (boolean; boolean)boolean;
   ~ : PROC PREFIX (boolean)boolean
END
```

Types in Poly are regarded as sets of "attributes". These attributes are usually procedures or constants but could be other types. The attributes of a type can be used exactly like ordinary objects with the same specification. However, since different types may have attributes with the same name, it is necessary to prefix the name of the attribute with the name of the type separated by **$**.

```
> integer$print(5);
5
```

This invokes the attribute 'print' belonging to integer and prints the number. Most types have a print attribute which prints a value of that type in an appropriate format. $ acts a selector which finds the attribute belonging to a particular type. It is not an operator so operators always work on the selected name rather than the type name.

```
> ~ boolean$true;
false
```

11. Records

Poly allows new types to be created in the same way as new procedures, constants or variables. One way of creating a new type is by making a record. A record is a group of similar or dissimilar objects.

```
> let rec == record(a, b: integer);
```

This declares 'rec' to be a record with two components, a and b, both of type integer.

```
> ? "rec";
rec : TYPE (rec)
  a : PROC(rec)integer;
  b : PROC(rec)integer;
  constr : PROC(integer;integer)rec
END
```

'constr' is a procedure which makes a record by taking two integers, and 'a' and 'b' are procedures which return the 'a' and 'b' values of the record.

```
> let recv == rec$constr(3, 4);
```

creates a new record with 3 in the first field (a) and 4 in the second field (b). The result is given the name 'recv'.

```
> rec$a(recv);
3
> rec$b(recv);
4
```

show that the values of the individual fields can be found by using 'a' and 'b' as procedures. They must of course be prefixed by 'rec$' to show the type they belong to.

Records can be made with fields of any specification, not just constants.

```
> let arec ==
# record(x:integer; p: proc(integer)integer);
```

declares a record with fields x and p, x being an integer constant and p a procedure.

```
> let apply ==
#   proc(z : arec)integer
#      begin
#      let pp == arec$p(z);
#      pp(arec$x(z))
#      end;
```

is a procedure which takes a constant of this record type and applies the procedure p to the value x and returns the result. In fact, it is not necessary to declare pp in the body of the procedure. An alternative way of writing apply is

```
> let apply ==
#   proc(z : arec)integer
#      (arec$p(z)(arec$x(z)));
```

12. Unions

Another way of constructing a type is using a 'union'. A union is a type whose values can be constructed from the values of several other types. For instance a value of a union of integer and string could be either an integer or a string.

```
> let un == union(int: integer; str: string);
```

This has created a type which is the union of integer and string. A value of the union type can be constructed by using an injection function. This union type has two such functions, their names made by appending 'int' and 'str' onto the letters 'inj_', making 'inj_int' and 'inj_str'. ('int' and 'str' were the 'tags' given in the declaration, in a similar way to fields in a record).

```
> let intunion == un$inj_int(3);
```

This has created a value with type 'un' containing the integer value 3.

```
> let stringunion == un$inj_str("The string");
```

creates a value, also with type 'un', but this time containing a string. Given a value of a union type it is often useful to be able to decide which of its constituent types it was made from. For each of the 'tags' there is a procedure whose name is made by prefixing with the letters 'is_', which returns 'true' or 'false' depending on whether its argument was made from the corresponding injection function.

```
> un$is_int(intunion);
true
```

prints 'true' because intunion was made from 'inj_int'. However

```
> un$is_str(intunion);
false
```

Values of the original types can be obtained by using 'projection' functions, which are the reverse of the 'injection' functions. Their names are made by prefixing the tags with 'proj_' to make names like 'proj_str' and 'proj_int'.

```
> un$proj_int(intunion);
3
> un$proj_str(stringunion);
The string
```

print the original values. It is possible to write

```
> un$proj_str(intunion);
Exception projecte raised
```

because 'intunion' has type 'un', just like 'stringunion'. However, 'proj_str' is expected to return a value with type string so when this is run it will cause an error. The effect will be to raise an 'exception' called 'projecterror' which means that a projection procedure was given an argument constructed using a different injection procedure.

```
> let unprojstr == un$proj_str;
> ? "unprojstr";
unprojstr : PROC(un)string RAISES projecterror
```

shows that 'proj_str' may raise 'projecterror'. Exceptions will be dealt with in more detail later on.

13. The Type-Constructor

It is often useful to be able to construct a type which is similar to an existing one but with additional attributes. This can be done by using the type-constructor.

```
> let nrec ==
#   type (r) extends rec;
#   let print ==
#      proc(v : r)
#        begin
#        print(r$a(v));
#        print(",");
#        print(r$b(v))
#        end
#    end;
> ? "nrec";
nrec : TYPE (nrec)
   a : PROC (nrec)integer;
   b : PROC (nrec)integer;
   constr : PROC (integer; integer)nrec;
   print : PROC (nrec)
   END
```

This declares 'nrec' to be a new type which is an 'extension' of an existing type 'rec'. It then lists the new attributes, in this case just the procedure 'print', which are declared just as though they were ordinary declarations. The name 'r' in parentheses which follows the word 'type' is the name for the new type within the body of the type constructor, so the argument of the procedure 'print' is given the type 'r'. It is important to remember that the new type is a completely separate type from 'rec'. Values can be changed from the old to the new type and vice versa, but they cannot be used interchangeably. The specification of nrec is similar to that of rec except that there is now an extra procedure 'print'.

```
> let nrecv == nrec$constr(5,6);
> nrec$print(nrecv);
5,6
```

makes a value with type nrec, and prints it using the new 'print' attribute. It is possible to write simply

```
> print(nrecv);
5,6
```

because there is a procedure 'print' which looks for the 'print' attribute of the type of the value given, and then calls it. This is the way integers and strings are printed (they both have 'print' attributes). Many of the other operations such as ':=' and '+' work in a similar way. A further alternative is to write an expression.

```
> nrecv;
5,6
```

In this case the compiler looks for the 'print' attribute and applies it.

14. A Further Example

This record could be extended in a different way, to make a double-precision integer. Suppose that the maximum range of numbers which could be held in a single integer was from -9999 to 9999. Then a double-precision number could be defined by representing it as a record with two fields, a high and low order part, and the actual number would have value (high)*10000 + (low). This can be implemented as follows.

```
> let dp ==
#   type (d) extends record(hi, lo: integer);
#   let succ ==
#      proc(x:d)d
#        begin
#        if d$lo(x) = 9999
#        then d$constr(succ(d$hi(x)), 0)
#        else if (d$hi(x) < 0) & (d$lo(x) = 0)
#        then d$constr(succ(d$hi(x)), neg(9999))
#        else d$constr(d$hi(x), succ(d$lo(x)))
#        end;
#   let pred ==
#      proc(x:d)d
#        begin
#        if d$lo(x) = neg(9999)
#        then d$constr(pred(d$hi(x)), 0)
#        else if (d$hi(x) > 0) & (d$lo(x) = 0)
#        then d$constr(pred(d$hi(x)), 9999)
#        else d$constr(d$hi(x), pred(d$lo(x)))
#        end;
#   let print ==
#      proc(x:d)
#        begin
#        if d$hi(x) <> 0
#        then
#           begin
#           print(d$hi(x));
#           if abs(d$lo(x)) < 10
#           then print("000")
#           else if abs(d$lo(x)) < 100
#           then print("00")
#           else if abs(d$lo(x)) < 1000
#           then print("0");
#           print(abs(d$lo(x)))
#           end
#        else print(d$lo(x))
#        end;
#   let zero == d$constr(0,0);
#   let iszero ==
#      proc(x:d) boolean
#        ((d$hi(x) = 0) & (d$lo(x) = 0))
#   end;
```

This is sufficient to provide the basis of all the arithmetic operations, since +,-,* etc. can all be defined in terms of succ, pred, zero and iszero.

15. Exceptions

In the section on union types above mention was made of exceptions. In the case of the projection operations of a union type an exception is raised when attempting to project a union value onto a type which was not the one used in the injection. An exception is simply a name and any exception can be raised by writing 'raise' followed by the name of the exception.

```
> raise somefault;
Exception somefault raised
```

raises an exception called 'somefault'.

```
> let procraises
#    == proc(b: boolean)
#         (if b then raise afault);
```

has specification

```
PROC(b: boolean) RAISES afault
```

Various operations, as well as projection, may raise exceptions. For instance many of the attributes of integer, such as 'succ' raise the exception 'rangeerror' if the result of the operation is outside the range which can be held in an integer constant. 'div' will raise 'divideerror' if it is asked to divide something by 0.

As well as being raised exceptions can also be caught, which allows a program to recover from an error. A group of statements enclosed in brackets or 'begin' and 'end' can have a 'catch phrase' as the last item. A catch phrase is the word **catch** followed by a procedure. e.g. 'catch p' will catch any exception raised in the group of statements and apply p to its name.

```
>let proccatches ==
#   proc(excp: string) (print(excp));
> begin
# procraises(true);
# catch proccatches
# end;
afault
```

'proccatches' has been declared as a procedure which takes a argument of type string. The exception is raised by 'procraises' and, since it is not caught in that procedure it propagates back to the point at which 'procraises' was called. The catch phrase catches the exception and calls the procedure with the name of the exception as the argument. The catching procedure can then look at the argument and decide what to do.

```
> begin
# procraises(false);
# catch proccatches
# end;
```

does not print anything because an exception has not been raised and so the procedure is not called.

If the block containing the catch phrase returns a value, then the catching procedure must return a similar value.

```
> let infinity == 99999;
> let divi ==
# proc infix(a, b: integer)integer
#    begin
#    a div b
#    catch proc(string)integer (infinity)
#    end;
```

This declares 'divi' to be similar to 'div' except that instead of raising an exception it returns a large number. Since 'a div b' returns an integer value the catch phrase must also return an integer.

16. The Specification of Variables

The specification of a variable in Poly is not, as one might expect, a constant of some reference type or a separate kind of specification, but each variable is in fact a separate type. Since a type in Poly is simply a set of constants, procedures or other types, a type can be used simply as a way of conveniently grouping together objects.

```
> let intpair ==
# type
# let first == 1;
# let second == 2
# end;
```

This has declared 'intpair' to be a pair of integers containing the values

1 and 2. 'intpair$first' and 'intpair$second' can be used as integer values directly.

The specification of an integer variable is

```
TYPE
assign: PROC(integer);
content: PROC()integer
END
```

A variable is a pair of procedures, 'assign' which stores a new value in the variable, and 'content' which extracts the current value from it. The standard assignment operator ':=' simply calls 'assign' on the variable. The compiler inserts a call to 'content' automatically when a variable is used when a constant is expected. 'assign' and 'content' can both be called explicitly.

```
> let vx == new(5);
> vx$assign(vx$content() + 1);
> vx$content();
6
```

As an example of a more complicated variable, suppose we wanted to write a subrange variable, similar to a subrange in Pascal, which could hold values between 0 and 10.

```
> let sr ==
#   begin
#   let varbl == new(0);
#      type
#      let content == varbl$content;
#      let assign ==
#         proc(i: integer)
#           (if (i < 0) | (i > 10)
#            then raise rangeerror
#            else varbl$assign(i))
#       end
#   end;
```

'varbl' is an integer variable which is initially set to 0. 'assign' checks the value before assigning it to 'varbl', and raises an exception if it is out of range. 'content' is just the 'content' procedure of the variable. It can be used in a similar way to a simple variable.

```
> sr := 2;
> sr;
2
> sr := 20;
Exception rangeerror raised
> sr;
2
```

17. Specifications in Declarations

The double-precision type declared above has one drawback. The specification contains the 'hi', 'lo' and 'constr' attributes in the specification of the type which would allow someone to construct a value which had the type 'dp', but had, for instance, fields outside the range -9999 to 9999 or with different signs. This could make some of the operations fail to work. We need a way of hiding details of the internals of a type declaration so that they do not appear in the specification, and so cannot be used outside. In Poly a specification can be given to something explicitly as well as having it inferred from the declaration.

```
> let aconst: integer == 2;
```

declares 'aconst' and forces it to have type 'integer'. The specification is written in the same way as the specification of the argument of a procedure.

```
> let quote : proc(string)
#    == proc(x: string)
#         begin
#         print("`");
#         print(x);
#         print("'")
#         end;
```

is another example of explicitly giving a specification to a value. An explicitly written specification is the specification of the name which is being declared. It need not be identical to the specification of the value following the '=='. However it must be possible to convert the specification of the value to the explicit specification (the 'context').

```
> let avar == new(3);
> let bconst: integer == avar;
```

declares 'avar' to be an integer variable and 'bconst' to be an integer constant. In the latter case the specification is necessary, otherwise 'bconst' would have been a variable and would have been another name for 'avar'. The conversion of a variable to a constant in order to match a given specification is one example of a 'coercion' of a value to match a 'context'. There are several others which can be applied depending on the particular specification. For instance the specification of a procedure may be changed from an operator to a simple procedure or vice versa.

```
> let plus:
#   proc(integer;integer)integer raises rangeerror
#   == integer$+ ;
```

declares 'plus' as a procedure which is the same as the '+' attribute of integer except that it is not an infix operator.

```
> plus(3,4);
7
```

The list of exceptions raised by the procedure must be included in the specification. The exception list in the specification given must include all the exceptions which may be raised, but may include others as well. A

special exception name **any** can be used to indicate that a procedure can raise any exception. Any exception list will match a context with exception list 'raises any'.
The specifications of the arguments and result must all match.

```
> let dble:
#    type (d)
#    succ, pred: proc(d)d raises rangeerror;
#    print: proc(d) raises rangeerror;
#    zero: d;
#    iszero: proc(d)boolean;
#    end
#   == dp;
```

creates a new type 'dble' with the specification given. The specification is the same as that of 'dp' but with some of the attributes of dp missing.

In the case of types the specification of the value must possess all the attributes of the explicit specification, but the explicit specification need not include all the attributes of the value. If a type is regarded as a set of named attributes then it is possible to take a subset of them and make them into a new type, simply by giving the new type the required specification. The specification of each attribute must itself match the specification that is given for it.

This mechanism provides a way of 'hiding' internal operations from the specification of a type. The specification of 'dble' above has only those attributes which are necessary to use it, and none of the operations which are used internally.

18. Types as Results of Procedures

So far we have considered procedures which take constants as arguments or return constants as results. In Poly values of any specification can be passed to or returned from a procedure. For instance

```
> let subrange
#   == proc(min, max, initial: integer)
#         type (s)
#         content: proc()integer;
#         assign: proc(integer) raises outofrange
#         end
#      begin
#         type
#         let varbl == new(initial);
#         let content == varbl$content;
#         let assign ==
#            proc(i: integer)
#              (if (i < min) | (i > max)
#               then raise outofrange
#               else varbl$assign(i))
#         end
#      end;
```

This procedure is similar to the definition of the subrange type 'sr' previously. However the bounds of the type are now arguments of a procedure so their values can be supplied when the program is run. Also new subrange variables can be created by calling the procedure.

```
> let sv == subrange(0,10,0);
```

This creates 'sv' as a variable of this subrange type. As with any procedure the arguments can be arbitrary expressions provided they return results with the correct specification.

19. Types as Arguments to Procedures

Types can be passed as arguments as well as being returned from procedures.

```
> let copy ==
#   proc(atype: type end)
#      type (t)
#      into: proc(atype)t;
#      outof: proc(t)atype
#      end
#   begin
#      type (t) extends atype;
#      let into == t$up
#      let outof == t$down
#      end
#   end;
```

This procedure takes a type and returns a type with two operations 'into' and 'outof'. 'up' and 'down' are procedures which are created when 'extends' is used, and provide a way of converting between the original and the resulting types. The specification of 'atype' merely says that it must be passed a type as an argument, but since it does not list any attributes then any type can be used as an actual argument (this is effectively saying that the empty set is a subset of every set). The procedure can be called, giving it an actual type as argument.

```
> let copyint == copy(integer);
```

The specification of the result is

```
   TYPE (copyint)
   into: PROC(integer)copyint;
   outof: PROC(copyint)integer
   END;
```

The specification of copyint allows mapping between integer and copyint since the type integer has been included in the specification.

```
> let copy5 == copyint$into(5);
> copyint$outof(copy5);
5
```

has mapped the integer constant 5 into and out of 'copyint'.

```
> let copychar == copy(char);
```

creates a similar type which maps between char and copychar.

20. Polymorphic Procedures

There are often cases where, in addition to passing a type as a argument, one or more values of that type are passed as well. For instance a procedure to find the second successor of a value might be written as

```
> let add2 ==
#   proc(atype:
#          type (t)
#          succ: proc(t)t raises rangeerror
#          end;
#        val: atype)
#     (atype$succ(atype$succ(val)));
```

The specification of 'val' is that it must be a constant, and its type is 'atype'. However 'atype' is also an argument to the procedure so the specification really means that this procedure could be called by giving it any type with the required attributes, and a constant which must be of the same type as the first argument.

```
> add2(integer, 2);
4
```

Similarly

```
> add2(char, 'A');
C
```

However

```
> add2(integer, 'A');
```

and

```
> add2(string, "A string");
```

both fail, in the first case because 'A' is not integer, and in the second because string does not have a successor function.

21. Implied Arguments

Many types have a 'print' attribute which prints a constant of the type.

```
> let pri ==
#   proc(printable: type (t) print(t) end; val: printable)
#      (printable$print(val));
```

declares 'pri' as a procedure which takes as arguments a type and a constant of that type and prints the constant using the 'print' attribute. This can be called by writing

```
> pri(integer, 3);
or
> pri(char, 'a');
```

since both 'integer' and 'char' have a 'print' attribute. Having to pass the type explicitly is really unneccessary, since it is possible for the system to find the type from the specification of the constant. It would be possible for the system to convert 'pri(3)' into 'pri(integer,3)' since '3' has type integer. In Poly types which can be deduced from the specifications of other arguments can be declared as 'implied' arguments. A argument list written in square brackets, [and], can preceed the normal argument list and those parameters, which must be all be types, are inferred from the other actual arguments when the procedure is called.

```
> let prin ==
#   proc [printable: type (t) print: proc(t) end]
#          (val: printable)
#      (printable$print(val));
```

This can now be called by writing

```
> prin(3);
or
> prin("hello");
```

and is in fact the definition of 'print' in the standard library. Alternatively 'prin' could have been declared by giving it an explicit specification and using 'pri'.

```
> let prin: proc[printable: type (t) print: proc(t) end]
#                 (printable)
#       == pri;
```

This is another form of conversion which can be made using an explicit specification. Using implied parameters can simplify considerably the use of procedures with types as arguments, and allow infix or prefix operators to be used in cases where they could not otherwise be used. For instance, consider an addition operation defined as

```
> let add ==
#   proc(summ: type (s) + : proc infix (s;s) raises rangeerror
#              end;
#        i, j: summ)summ
#          (i + j);
```

would be used by writing

```
> add(integer, 1, 2);
3
```

However, by writing

```
> let +
# : proc infix [summ: type(s)
#                    + : proc infix (s;s)raises rangeerror
#                     end]
#              (i, j: summ)summ raises rangeerror
#    == add;
```

'+' can become an infix operator, since it has only two actual arguments. Similar definitions are used for many of the other declarations in the library.

22. Literals

We have already seen how constants can be written as "Hello" or 42. These are known as literal constants, because their values are given by the characters which form them, rather than by some previous declaration. They are however, only sequences of characters, it is only by convention that "Hello" is a string constant and 42 an integer constant. This is only important when we wish to use some other definition than the 'standard' one. For instance, if the type integer were restricted to the range -9999 to 9999 then the constant 100000 would be an error if it were treated as an integer. The definition of double-precision integer above, would, however, be able to represent it.

In Poly, therefore, literals have no intrinsic type, they must be converted into a value by the use of a conversion routine. The compiler recognises certain sequences of characters as literals rather than names or special symbols. The three forms of literal constants recognised by the compiler are 'numbers', 'double-quoted sequences' and 'single-quoted sequences'. 'Numbers' begin with a digit and may consist of numbers or letters.

```
42 0H3F6A 3p14159
```

are examples of 'numbers'. 'Double-quoted sequences' are sequences of characters contained in double-quotes. A double-quote character inside the sequence must be written twice.

```
"Hello"     ""     "He said ""Hello"""
```

'Single-quoted sequences' are similar to double-quoted sequences but single rather than double-quotes are used.

```
'Hello'     ''     'He said ''Hello'''
```

When the compiler recognises one of these literals it tries to construct a call to a conversion routine which can interpret it as a value of some type. For instance, the standard library contains a definition of 'convertn' which the compiler calls if it finds a 'number'. That definition has specification

```
PROC(string)integer
```

All conversion routines must have similar specifications, but the result type will differ and some exceptions may be raised. The literal is supplied as a constant of type 'string'. The conversion routine can examine the characters which form the literal and return the appropriate value. It may of course raise an exception if the characters do not form a valid value, if either the value would be out of range or if the literal contains illegal characters.

There are also two other conversion routines in the standard library, 'converts' which converts double-quoted sequences into string values, and 'convertc' which converts single-quoted sequences into values of the type 'char'. These definitions can be overridden by preceeding the literal by the name of a type and a $ sign. For instance

```
> let int == integer;
> let one == int$1;
```

applies the 'convertn' routine belonging to 'int', so that 'one' has type int rather than integer.

23. Lists

Lists are a convenient example for polymorphic operations. List types can be constructed by the following procedure.

```
> let list ==
# proc(base: type end)
#    type (list)
#    car : proc(list)base raises nil_list;
#    cdr : proc(list)list raises nil_list;
#    cons: proc(base; list)list;
#    nil : list;
#    null: proc(list)boolean
#    end
#  begin
#     type (list)
#     let node == record(cr: base; cd: list);
#     extends union(nl : void; nnl : node);
#
#     let cons ==
#       proc(bb: base; ll: list)list
#         (list$inj_nnl(node$constr(bb, ll)));
#
#     let car ==
#       proc(ll: list)base
#         begin
#         node$cr(list$proj_nnl(ll))
#         catch proc(string)base (raise nil_list)
#         end;
#
#     let cdr ==
#       proc(ll: list)list
#         begin
#         node$cd(list$proj_nnl(ll))
#         catch proc(string)list (raise nil_list)
#         end;
#
#     let nil == list$inj_nl(void$empty);
#
#     let null == list$is_nl
#     end
#   end;
```

'void' is a standard type which has only one value (empty), and is used to represent the 'nil' value of the list. The list structure is made using a recursive union with each node containing a value of the 'base' type and the next item of the list, or containing a nil value. 'cons' makes a new node of the list, 'car' and 'cdr' find the 'base' and 'list' parts of a node respectively, and 'null' tests for the value 'nil'. 'car' and 'cdr' both trap the exeception which would be raised if a projection error occured and raise 'nil_value' in its place.

A particular list type can now be created, for instance a list of integers.

```
> let ilist == list(integer);
> let il == ilist$cons(1, ilist$cons(2, ilist$cons(3, ilist$nil)));
```

A polymorphic 'cons' function could be declared to work on lists of any base type.

```
> let cons ==
#    proc[base: type end;
#         list: type (l) cons: proc(base; l)l end]
#        (bb: base; ll: list)list
#           (list$cons(bb, ll));
```

It is now possible to write simply

```
> let il == cons(1, cons(2, cons(3, ilist$nil)));
```

Polymorphic 'car', 'cdr' and 'null' functions can be written similarly. As further examples some other polymorphic list functions are given.

```
> letrec append ==
# proc[base: type end;
#      list: type (l)
#            car: proc(l)base raises nil_list;
#            cdr: proc(l)l raises nil_list;
#            cons: proc(base; l)l;
#            null: proc(l)boolean end]
#     (first, second: list)list
#     ( if null(first) then second
#       else cons(car(first), append(cdr(first), second)) );
> letrec reverse ==
# proc[base: type end;
#      list: type (l)
#            car: proc(l)base raises nil_list;
#            cdr: proc(l)l raises nil_list;
#            cons: proc(base; l)l;
#            nil: l;
#            null: proc(l)boolean end]
#     (ll: list)list
#     ( if null(ll) then list$nil
#       else append(reverse(cdr(ll)), cons(car(ll), list$nil)) );
```

A useful function would be one which would print the data part of a list if the base type could be printed.

```
> letrec pr ==
# proc [base: type(b) print: proc(b) end;
#        list: type(l) car: proc(l)base raises nil_list;
#                      cdr: proc(l)l raises nil_list;
#                      null: proc(l)boolean
#              end ]
#       (ll: list)
#       begin
#       if null(ll)
#       then print("nil")
#       else
#          begin
#          print("( ");
#          print(list$car(ll));
#          print(". ");
#          pr(list$cdr(ll));
#          print(") ")
#          end
#       catch proc(string) ()
#       end;
```

The list created above can now be printed.

```
> pr(il);
( 1. ( 2. ( 3. nil) ) )
```

Other polymorphic functions on lists can be declared in a similar way.

24. Conclusion

This document is intended as an introduction to Poly and to give some idea of the ways in which it can be used. It is not a rigorous description and various details, such as the precise checking rules for specifications, have been deliberately skated over in order to explain the language simply. A companion document, the Poly Report, is the reference for the precise details of the language.