Poly Manual

David C.J. Matthews

An original version of this document was published as University of Cambridge Computer Laboratory Technical Report 63 and appeared in SIGPLAN Notices Vol 20 No. 9 Sept. 1985.

Chapter 1. Introduction

Poly is a general purpose high-level programming language. It has a simple type system which is also very powerful. Higher order procedures, polymorphic operations, parameterised abstract types and modules are all supported by a single mechanism.

Poly is **strongly** typed. All objects have a **signature** which the compiler can use to check that operations applied to them are sensible. Type errors cannot cause run time faults.

The language is **safe**. Any faults that occur at run time will result in exceptions which can be caught and handled by the user. All variables must be initialised before they are used so faults due to undefined variables cannot occur.

Poly allows **higher** order procedures to be declared and used. A higher order procedure is one which takes a procedure as a parameter or returns a procedure as its result. Since Poly is **statically** scoped (the value bound to an identifier is determined by the static block structure) the procedure that is returned may refer to the arguments and local values belonging to the procedure that returned it, even though that procedure is no longer active.

Poly allows **polymorphic** operations. For example, in many languages such as Pascal or MODULA-2 programs to sort arrays of integers, arrays of strings or arrays of real numbers are textually almost identical. They differ only in the actual operation used to compare two elements of the array. In Poly one program can be written which will sort arrays of any type provided elements can be compared.

Poly allows **abstract** types to be created and manipulated. A"hash table" type can be defined that allows an arbitrary object to be stored along with a string which acts as a key. There would be a look-up function that will return the object stored with a given key. These can be written so that only the functions to create a table, enter a value against a key, and return the value with the key, are available to the user of the hash table. This has the advantages that the writer of the hash table function can change the **way** it is implemented provided it has the same **external** properties. The user can make use of the hash table knowing that he will not be able to upset its internal structure by accidentally using a function which was intended to be private.

Abstract types can be **parameterised** so that a set of types with similar properties can be defined in a single definition. A specific type can then be made from that. For example, a single hash table type could be declared from which hash tables to hold any particular type could be derived.

Types in Poly are similar to **modules** in other languages. For example, types can be separately compiled. An abstract type which makes use of other types can be written as though it were polymorphic - it will work if it is given any type (module) which has the required operations. Its operation may be simply to return a new type (module) as its result. This new type may be used directly or as a parameter to other polymorphic

abstract types. There is a mechanism for constructing large programs out of their modules and recompiling those which have been modified since they were last compiled.

Chapter 2. The Type System

The purpose of a type system is to avoid mistakes due to using a value in a way that was not intended, while making meaningful operations easy to express. For example, if we have two matrices with the same dimensions, it should be just as easy to write the instruction to add them together as if they were integers. However it should not be possible to add an integer to a matrix. A type system could be designed in which all these rules were built into the type checker. This has the problem that new types with new rules cannot be added in. A better way is to have a few simple rules which allow new types to be added and checked but which can be used to catch most of the faults which could be made. The Poly type system is based on this idea.

All objects have a **signature** which is checked by the type-checker. The signature corresponds to a **type** in other languages, but is more general to take account of the greater power of the type system. For example, in a language like Pascal, a parameter to a procedure may have type *integer*. This gives enough information for the compiler to check that the procedure is correctly used; it can only be applied to an integer value, but it does not specify precisely which value. It can be applied to 1, 2, 3 etc. but not to strings such as "hello" or "goodbye". The checking done by the compiler ensures that certain kinds of faults will not happen when the program is run, but it cannot prevent all faults.

In Poly this approach is generalised to include procedures, types and exceptions as well as values. The signature of an object contains the information which the compiler uses to check that it is correctly used without restricting it to precisely one object. Expressions and variables can be made to return any kind of object and the signature of the result can be worked out by the compiler, provided of course that all the signatures in the expression match. Specifications have a standard textual form which allows them to be written in a program or output by the compiler. The rules for matching each kind of signature and their textual forms are described below.

2.1. Values

The simplest kind of object is the **value** which can be operated on but does not do anything itself. Examples are the number 42 or the string "hello". A value is said to **belong** to a type or **have** a particular type, which in Poly is always a named type. The signature is the name of the type. For example, the signature of the number 42 is *integer* and that of "hello" is *string*. Two values only match if they belong to the same type.

Syntax
<value signature> ::= <identifier> |<value signature>$<identifier>

2.2. Procedures

Procedures are objects which perform a computation. They often take **parameters** which can be used by the procedure and always return a **result**. They may also have **side-effects** or raise **exceptions**. Examples of procedures are "+" which adds together two values and "*print*" which prints a value. "+" is an infix operator which takes two values as parameters, and returns a single result.

3 + 4

"*print*" is a procedure which has the side-effect of printing the value.

print(3+4);

prints out the value 7. It incidentally also produces a result, but it has type *void* which has only one value, and is ignored.

Procedures can take procedures or types as parameters as well as simple values and can also return them as results. Procedures match if their parameters and results all match. The various forms of procedures will be described in the section on the procedure constructor.

Syntax	
<procedure signature>	::= proc <mode list> <implied parameters> <actual parameters> <signature>
<mode list>	::= <empty> | <mode> <mode list>
<mode>	::= **infix** <digit> | **infixr** <digit> | **early** | **inline**
<implied parameters>	::= [<parameter list>] | <empty>
<actual parameters>	::= (<parameter list>)
<parameter list>	::= <empty> | <parameter> |<parameter>; <parameter list>
<parameter>	::= <identifier list>: <signature> | <signature>
<identifier list>	::= <identifier> | <identifier list>, <identifier>

2.3. Types

Poly has a novel view of **types** in that they are treated as being objects as well as having a role in checking the signature of values. Each type has a set of objects associated with it and a **type mark**. The type mark is used purely by the compiler in checking the signatures of objects and corresponds to the notion of a type in other languages, while the set of objects makes a type in Poly very similar to a module. All types have both a set of objects (which may however be empty) and a type mark, but one or the other may be more important in different circumstances.

2.3.1. Sets of Objects

As an example of the set of objects, the type *integer* has various operations such as addition, subtraction, multiplication etc. which can operate on values of the integer type. Any program which works on integers will ultimately be written in terms of these basic operations. Similarly the type *real* will have these operations along with others which convert between integer and real.

The signature of a type is the signature of the objects which make it up. Every object in a type has a name, and all the names in one type are different, although objects with the same name frequently exist in different types. So for instance, many types have a *print* procedure which takes as its parameter a value of the type, and prints it.

The name of an object in a type is intended to suggest the semantics of the operation, but there is no guarantee that the + operations in all types are commutative; in the type *string* it is used for concatenation. (This would require a completely new level of semantic checking which is outside the scope of a conventional compiler. The CLEAR system [Burstall and Goguen] attempts this kind of checking.).

Most of the objects in types are procedures, but a type can contain simple values as well as other types. For instance there may be a *first* and a *last* value which give the maximum and minimum values. There is a distinction between objects which are *part* of the type and those which been created by operations of it and are said to *belong* to the type or *have* that type. For example, there is no 3 in the type *integer* but the number 3 *has* type *integer*.

As types are regarded as sets it is reasonable to be able to take subsets or select a particular object from a type. For example,

type (*atype*)
x, *y*: *atype*;
add: **proc**(*atype*; *atype*)*atype*;
sub: **proc**(*atype*; *atype*)*atype*
end

this is the signature of a type with four objects, called *x*, *y*, *add* and *sub*. *x* and *y* are both values of this type, and *add* and *sub* are procedures which take a pair of values, and return a value. The name *atype* in brackets after the word **type** is the name used to represent the type itself inside the type signature. This type will match any of the following

type (*t*) { Only the name has changed }
x, *y*: *t*;
add: **proc**(*t*; *t*)*t*;
sub: **proc**(*t*; *t*)*t*
end

type (*atype*) { The objects are in a different order }
sub, *add*: **proc**(*t*; *t*)*t*;
y: *t*;
x: *t*;
end

type (*at*) { A subset }
x: *at*;
add: **proc**(*at*; *at*)*at*
end

type (*atype*) { Another subset }
add: **proc**(*atype*; *atype*)*atype*
end

type { Another subset - No need for an internal name }
end

This last example is the empty type which matches any type. The text in curly brackets is comment and

ignored by the compiler.

2.3.2 Type Marks and the Specifications of Values

The function of the **type mark** is in the checking of the signatures of values. Each type has a distinct type mark which is used to identify values which have that type. The signature of a value is simply the type mark of the type to which it belongs. Checking the signatures of two values to see if they match reduces to seeing if they are the same type mark, there is no question of comparing the signatures of the types themselves.

The reason is that there may be many types with the same signature (short and long precision integers may have the same set of operations, +, – etc. but they are different types). The compiler produces type marks in various circumstances so as to guarantee that two different types will always have different type marks. The converse of this is that there are many circumstances in which two types which are in fact identical may have different type marks, and values associated with them will be incompatible. An expression which returns a type always has a type mark which differs from any other, in particular if an existing type is bound to a new name then they will have different type marks. Values which have the old type have a different type mark from the new one and so are incompatible, despite the types being in fact identical.

2.3.3 Sets and Marks

There are circumstances when one or other of the two ways of viewing a type becomes more important. Some types are used only as collections of objects and there are no values associated with them. The type mark for those types is irrelevant. Equally there are occasions in which a type is used where the set of objects is irrelevant. Any type matches the empty type "**type end**" which has no objects in it. The type mark of the matching type is still there and is used by the compiler.

This is important because of the matching rules for procedure parameters if one or more is a type. A procedure which takes a type as a parameter may use the formal parameter name in the signature of other parameters. For example a procedure may have signature

aproc: **proc**(*atype*: **type end**; *x*: *atype*)*atype*

This procedure takes two parameters, a type which may be any type, and a value which has the same type as the **actual** parameter. Its result also has this type. This kind of procedure is known as polymorphic. It can therefore be applied to

aproc(*integer*, 99)

in which case the result will have type *integer* or

aproc(*string*, "hello")

returning a string. This procedure might be the identity function which simply returns its second parameter (the value) as its result. What is happening is that the actual type parameter is matched to the formal parameter using the matching rules for types (the formal parameter must be a subset of the actual parameter), and then the type mark of the formal parameter is replaced with the type mark of the actual parameter in the other parameters and the result. The other parameters therefore match if they have the type mark of the actual parameter. The signature of the result is obtained by replacing the formal parameter's type mark by the actual parameter. It is also possible to obtain the type from the type marks of values, and this is used to remove the

need to explicitly pass type parameters in many cases.

The reason for considering types both as sets and as marks is that it becomes possible to write polymorphic operations which make use of objects in types. For example a sorting procedure can be written which will work on any values provided they belong to a type whose values can be compared for ordering. How this is done will be described in detail in the section on procedures.

Syntax		
<type signature>	::=	**type** <internal name> <signature list> **end**
<internal name>	::=	<empty> | (<identifier>)
<signature list>	::=	<empty> | <object list>
<object list>	::=	<object> | <object>; <object list>
<object>	::=	<identifier list> : <signature>
<identifier list>	::=	<identifier> | <identifier>, <identifier list>

2.3.4. Exceptions

The fourth kind of object in Poly is the exception. The mechanism of exceptions is based on that of Standard ML.

Syntax		
<exception signature>	::=	**exception** <implied parameters> <actual parameters>

Chapter 3. Expressions and Values

The basic element of a Poly program is the **expression**. An expression has a value and a signature which ensures that the value is correctly used. Expressions consist of identifiers and constructors and operations on them, either procedure applications or selections from types.

3.1. Identifiers

An **identifier** is a name which represents an object. The binding of a name to a particular object is made by a **declaration**. An identifier may be any string of alphanumeric characters starting with a letter, or a string of one or more "special" characters. The underline character "_" is considered as an alphanumeric. Each of the following are identifiers, separated by spaces.

a Message integer j + := >>>>>> L999a
An_extremely_long_identifier

The "special" characters are often used for infix operators, but can be used for anything. They are

! # %& = - + * : < > / \ ? ~ ^ | . @

Certain words are **reserved** and cannot be used as identifiers because they are used for special purposes. These are

and	**begin**	**cand**	**catch**	**cor**	**do**	**early**
else	**end**	**exception**	**extends**	**if**	**infix**	**infixr**
inline	**let**	**letrec**	**proc**	**raise**	**record**	**struct**
then	**type**	**union**	**while**	**:**	**==**	**.**

Identifiers written in different cases are regarded as distinct, except that reserved words may be written in either or mixed case. In this manual reserved words are always shown in bold font while identifiers are given in italics.

Syntax		
< identifier>	::=	<letter> | <identifier><letter>|<identifier><digit>

Comments in Poly are written between curly brackets "{" and "}". Any text in the brackets is completely ignored and the whole comment is simply regarded as a separator between words in the same way as a space or a new line.

3.2. Selectors

A selector selects an object from a type.

integer$+

selects the "+" operation from the type *integer*, while

string$+

selects "+" from *string*.

Syntax		
<selector>	::=	<identifier>$<identifier>\|<selector>$<identifier>

3.3. Constructors

Constructors make values from other values. There are general constructors for procedures and types as well as three constructors which make special kinds of types: **records, unions,** and **structures**. There are also constructors for values which allow literal constants to be entered in a program.

1 999 "hello" 'A' 0xff

Literal constants are either numbers or strings of characters. Numbers are strings of digits or letters starting with a digit, and strings are any sequence of characters unclosed in quotation marks. By default numbers are converted to type **integer** and strings to either **char** if they are enclosed in single quotes ('), or **string** if they are in double quotes (").

3.4. Declarations

The result of any expression can be bound to an identifier by a declaration.

let *result* == 4+3* 2;

The identifier *result* can be used in place of the value that is bound to it.

result+6

will yield the value 16. As well as taking the value from the expression, the identifier also inherits its signature. The signature of *result* is therefore *integer*. This works for any expression including those which yield procedures or types.

let *p* == *print*;

declares *p* to be the same as *print* so

p 10;

will print the value 10.

An explicit signature may be given for an identifier.

let *i*: *integer* == 10;

The result of the expression must have this signature for the compiler to allow it. It is useful if a complex expression is being bound to an identifier to check the signature of the result when it is being bound rather than when it is used.

The identifier in an ordinary declaration is declared **after** the expression is evaluated.

let *i* == *i*+1

is valid provided *i* has been declared before. However when recursive procedures or types are being declared a different kind of declaration is needed.

letrec *p* == ....

letrec introduces a recursive declaration, and the *p* used in the expression will be the *p* that is being declared. Recursive declarations can only be used for procedures or types and will be described in more detail below.

Several declarations can be grouped together with **and**.

let *a* == 1 **and** *b* == 2

This declares both *a* and *b*. Grouping declarations together in this way is useful for mutually recursive declarations.

letrec *p* == .... **and** *q* == ....

Syntax

<declaration>	::= **let** <binding> **and** .... **and** <binding> | **letrec** <binding> **and** .... **and** <binding>
<binding>	::= <identifier> : <signature> == <expression> | <identifier> == <expression>

3.5. Blocks

Commands can be grouped by enclosing them in the bracketing symbols **begin** and **end** or **(** and **)**.

2* (3+4);

begin *print* "Hello"; *print* " again" **end**

A block can consist of several expressions separated by semicolons or just one. While **begin** and **end** or round brackets can be used in either case, it is usual to use **begin** and **end** to group several expressions together, and round brackets to group parts of an expression which are to be evaluated first. The value returned by the block is the value of the last expression. All the other expressions must return values with type *void*. Empty blocks are allowed and these return *void*.

Declarations may appear in the block as well as expressions. The identifier is then available in any of the other expressions after its declaration.

begin let *x* == 2; *x* + 3 **end**

This block returns the value 5. *x* is not available outside the block, but it is available in inner blocks.

begin
let *p* == *print*;
begin
let *ten* == 10;
p ten
end
end

An identifier declared in a block "hides" an identifier with the same name in a outer block.

Syntax	
<block>	::= **begin** <expressionlist> catchphrase **end** | **(** <expressionlist> <catchphrase> **)** | **( )** | **begin end**
<expressionlist>	::= <expordec>; ... ; <expordec>
<expordec>	::= <expression> | <declaration>

3.5.1. Conditionals

An expression can return different results according to the value of a test.

if 3 > 4 **then** 1 **else** 2;

The result of the conditional is the expression following **then** if the condition is *true* and the expression after

else if the expression is *false*. In this case the result will be 2, since the condition is clearly false. The expression to be tested must have type *boolean* which contains the two values *true* and *false*. The two expressions returned by the then- and else-parts must be the same. The else-part may be omitted if the then-part returns a *void* result.

if *x* > 3 **then** *print* "yes"

Conditionals may return procedures or types but the then- and else-parts must both return values with compatible signatures.

if ... **then** *integer*$*pred* **else** *integer*$*succ*

The expression returns a procedure which takes an integer parameter and returns an integer result.

Syntax		
<conditional>	::=	**if** <expression> **then** <expression>**else** <expression> | **if** <expression>**then** <expression>

Related to the if-expression are **cand** and **cor**. Syntactically they behave like infix operators of precedence -1 and -2 respectively but they are actually reserved words.

x = 1 **cand** *y* =2

is the same as

if *x* = 1 **then** *y* = 2 **else** *false*

and

x = 1 **cor** *y* =2

is the same as

if *x* = 1 **then** *true* **else** *y* = 2

3.5.2. Repetition

An expression can be repeated while some condition holds.

while @*x* > 0 **do** *x* := *pred*(@*x*)

decrements *x* until it is zero, by repeating the second expression until the first returns *false*. The expression after the **while** must have type *boolean* and the expression after **do** must have type *void*. The result of a "while-loop" has type *void*.

The "while-expression" is sometimes convenient, but it is usually both clearer and more efficient to use a recursive procedure.

Syntax

<while loop>	::=	**while** <expression> **do** <expression>

Chapter 4. Procedures

A procedure is an encapsulated piece of program which may take some parameters and returns a result.

4.1. The Procedure Constructor

A procedure is made by the **procedure constructor**, called a lambda expression in some languages, which is a expression preceded by a **procedure header**. The procedure header gives the names and signatures of the parameters as they will be used in the expression and the signature of the result.

proc(*i*: *integer*)*integer* . *i* + 1

This is a procedure which takes a parameter called *i* in the block, which is a value of type integer and it returns a result which is a value of type integer. The expression returns a result which is one more than the parameter. This expression is evaluated when the procedure is called and so it is equivalent to the successor function for integer.

The procedure constructor is an expression which returns a procedure as its result. It can be used directly in an expression, but usually it is bound to an identifier.

let *imax* == **proc**(*i*, *j*: *integer*)*integer* . **if** *i* > *j* **then** *i* **else** *j*

The identifier is then used in an expression

imax(*1*, imax(2, 3))

4.2. Recursive Procedures

Recursive procedures are declared using **letrec**.

letrec *fact* == **proc**(*i*: *integer*)*integer* . **if** *i* = 1 **then** 1 **else** *fact*(*i*-1) * *i*

This has made the usual recursive definition of the factorial function. Recursive procedures are the preferred way of making loops and repeating expressions in Poly.

4.3. Operators

Procedures can be declared so that they can be used as infix operators. Infix operators have a **precedence** which determines how tightly they bind. For example, the expression

1* 2+3* 4

would return 20 if it were evaluated strictly from left to right, but yields 14 if it is evaluated using the normal algebraic rules. In this case the multiplication operator * is said to have a higher precedence than the addition

operator +. In Poly the precedence of an infix operator is given as a number between 0 and 9, the higher the number the greater the precedence.

let *rem* ==
proc infix 7 (*i*, *j*: *integer*)*integer* . *i* - *i div j* * *j*

This declares *rem* to return the remainder after dividing *i* by *j*.

73 *rem* 4

In this case *rem* has been given a precedence of 7, which is the same as the multiplication and division operators. The precedence of the other operators is given in the description of the standard definitions. Operators with the same precedence declared with *infix* associate to the left. Operators can be made right associative by using *infixr*.

4.4. Polymorphic Procedures

The *imax* procedure declared above takes integer values and returns the larger of the two, which is of course also an integer. A similar procedure can be written to return the greater of two strings (in alphabetical order).

let *smax* == **proc**(*i*, *j*: *string*)*string* . **if** *i* > *j* **then** *i* **else** *j*

smax is exactly the same as *imax* except for the change in the names of the types. A similar procedure could be written for any type, provided of course that values of the type can be compared with a > operator. In Poly a single procedure can be written to handle all these cases, such a procedure is called **polymorphic**.

let *pmax* == **proc**(*a_type*: **type**(*t*) > : **proc**(*t*;*t*)*boolean* **end**; *i*, *j*: *a_type*)*a_type* .**if** *i* > *j* **then** *i* **else** *j*

It works by requiring an extra parameter, *a_type*, which is the type of the values (recall that types can be passed as parameters to procedures). The important thing about this type is that it must have a > operator which compares two values of the type and returns a boolean value. The type signature

type(*t*) > : **proc**(*t*; *t*)*boolean* **end**

expresses this constraint. The other two parameters and the result must be of this type. *pmax* can therefore be applied to any type which satisfies the constraint, and a pair of values of the type.

pmax(*integer*, 1, 2)

returns an integer result, while

pmax(*string*, "abc", "abd")

will return a string. However

pmax(*integer*, 1, "abc")
pmax(*string*, 3, 4)

will be rejected by the compiler because the signatures do not match.

max(boolean, true, false)

will also fail, because *boolean* does not possess a > operator.

4.5. Implied Parameters

It is not very convenient to have to write an extra parameter when calling polymorphic procedures, especially since it is just the type of the other parameters. Poly allows a polymorphic procedure to be written so that the type parameters need not be given explicitly but are passed implicitly.

let *max* ==
proc[*a_type*: **type** (*t*) > : **proc**(*t*;*t*)*boolean* **end]**(*i*, *j*: *a_type*)*a_type* . **if** *i* > *j* **then** *i* **else** *j*

The type parameter in this case is enclosed in square brackets to indicate that there will not be a corresponding actual parameter.

max(3,4)

looks at the actual parameters, finds that they are integers and so passes the type *integer* implicitly.

max("abc", "bcd")

passes the type *string*.

max(3, "abc")

is incorrect because the parameters must have the same type.

Implied parameters are a very powerful facility. All the operators such as + and > are polymorphic procedures which take the type of their explicit parameters as an implied parameter. Their only action is to select and apply the appropriate procedure from the type (This does not mean that adding two integers together requires two procedure calls. These operations are implemented as inline procedures and the compiler optimises it down to a single "add" instruction.) For example, the definition of + in the standard header is

let + { addition } == **proc early inline infix** 6 **[***inttype* : **type** (*t*) + : **proc** (*t*; *t*)*t* **end]** (*x*, *y* : *inttype*) *inttype*} . *x* *inttype*$+ *y*

The words **early** and **inline** are directives to the compiler. **early** tells the compiler that this procedure should be evaluated as soon as possible. This usually means that the compiler will attempt to execute it when it is compiled if its parameters are constants (Since procedures can have side-effects the compiler must not attempt to evaluate all procedures at compile-time. Consider, for example, a procedure which returns the current date and time). **inline** tells the compiler to insert the code for this procedure at the point of call rather than generate a procedure call. Both **early** and **inline** are hints to the compiler rather than instructions, and the compiler may choose to ignore either or both. \syntax{<procedure signature> . <expression>} { <procedure constructor> ::=<procedure signature> . <expression> }

Syntax		
<procedure constructor>	::=	<procedure signature> . <expression>

Chapter 5. Exceptions

Normally expressions in a block are executed one after another until the end of the block is reached. There are occasions, however, when an unusual case occurs and an escape is needed.

let *p* == *q div r*

For example, a program containing a divide operation could possibly fail if *r* were zero. Explicitly checking for zero before making the division would be tedious and unnecessary in most cases, so what happens is that an **exception** is generated. Exceptions are error conditions together with a string which identifies the cause of the failure. Dividing by zero, for example, results in an exception with the string *divideerror*. An exception can also be generated by using **raise**.

raise *an_error*

generates an exception with the name *an_error*.

Exceptions generated in a block may be caught by a **handler**. A handler is given control when any exception in the block happens and either returns a result or raises another exception. The handler is a procedure whose parameter is a string, the exception name, and result must be compatible with the result the block would return if the exception had not happened.

begin
i div j
catch proc (*name*: *string*)*integer*
begin
print("Exception-");
print(*name*);
9999
end
end

This block returns the result of dividing *i* by *j* unless an exception occurs. In that case it prints out *Exception-* followed by the name of the exception, and returns the (large) value 9999.

The handling procedure can be any which has the correct signature, but frequently it is written as a procedure constructor after the word **catch** . Any exceptions raised by the handler are, of course, not caught by it, but appear in the next block out. In addition, if the block contains declarations they are not available to the handler. This is because an exception could occur while the declarations were being made so the identifier would have no value.

begin
let *val* == *i div j*;
let *otherval* == *i*+1;
catch proc (*name*: *string*)...
begin { Cannot use val or otherval here }

end
end

If an exception is not caught in a block it automatically propagates to the containing block. This in turn can handle it, or allow it to propagate to the next block out. An exception raised in a procedure but not caught in it causes the procedure to return and the exception appears at the point where the procedure was called. The calling block will catch the exception if it has a handler or it will propagate back further.

Syntax		
<raise expression>	::=	**raise** <identifier>
<catch phrase>	::=	**catch** <expression>

Chapter 6. Specialised Type Constructors

There are three specialised constructors which make different kinds of types. They are normally used to provide the "concrete" type which implements an abstract type. The development of abstract types will be described in the next chapter.

6.1. Records

A **record type** allows objects composed of a group of values to be put together and taken apart.

let *int_pair* == **record** (*first*, *second*: *integer*) } makes a type with the operations for making pairs of integers. The names *first* and *second* are known as **field names** and the signature, in this case *integer* is the **field signature**. The result of this declaration is a type *int_pair* has three operations in it, *constr*, *first* and *second*.

Every record has a *constr* procedure which takes objects with the field signatures and makes them into a record value. The *constr* in *int_pair* takes two integer values and packages them up as a value of the *int_pair* type.

let *pair_val* == *int_pair*$*constr*(1, 2);

The field names *first* and *second* are procedures called **selectors** that take apart values of the *int_pair* type and return the first and second values respectively.

int_pair$*first*(*pair_val*)

will return 1 and

int_pair$*second*(*pair_val*)

will return 2. Records can be made with elements of any signature and any number of elements.

let *prec* == **record** (*val*: *integer*; *pr*: **proc** (*integer*)*integer*);

makes a record to hold a value and a procedure. A value of this type can be made by

let *prec_val* == *prec*$*constr*(1, *integer*$*succ*)

and the selecting functions *val* and *pr* now return an integer value and a procedure respectively.

prec$*pr*(*prec_val*)(99) + *prec*$*val*(*prec_val*)

Note, however that each invocation of the record constructor, as with the other constructors, yields a type with a new unique type mark. This means that two record types with identical field names and signatures are regarded as different types. A more convenient syntax for selection is available which allows

pair_val.*first pair_val*.*second*

to be used with exactly the same meaning as

int_pair$*first*(*pair_val*) *int_pair*$*second*(*pair_val*)

This syntax is not restricted to record selection but can be used with any procedure that is an object in a type and takes one argument of that type. The meaning of *X.Y* is

X_type$*Y*(*X*)

where *X_type* is the type to which *X* belongs. So for example,

99.*succ*.*print*

is equivalent to

integer$*print*(*integer*$*succ*(99))

Syntax		
<record constructor>	::=	**record (** <field list> **)**
<field list>	::=	<field> | <field>**;** <field list>
<field>	::=	<identifier list> **:** <signature>
<identifier list>	::=	<identifier> | <identifier> **,** <identifier list>

6.2. Unions

The record constructor makes types whose values are groups of objects. Another constructor, the **union** constructor, makes types whose values may have any of a set of signatures.

let *int_or_str* == **union** (*int*: *integer*; *str*: *string*)

This has created a type whose values can be either integers or strings. The names before the colons (*int* and *str*) are called **tags** and a tag and its signature (after the colon) is called a **variant**. An integer or string may be converted into this type by **injection** operations.

let *int_form* == *int_or_str*$*inj_int*(99)
let *str_form* == *int_or_str*$*inj_str*("hello")

The names of the injection operations are made by prepending the word *inj_* to the tags. The original string and integer values can be obtained by **projecting** the union value.

int_or_str$*proj_int*(*int_form*)
int_or_str$*proj_str*(*str_form*)

The names of these operations are made in a similar way to the injection operations and return a value with the appropriate signature. Of course it is possible to apply the wrong projection.

int_or_str$*proj_str*(*int_form*)

Since *int_form* contains an integer it cannot be made to return a string, and so this operation will cause an exception with the name *projecterror*. To avoid getting exceptions, the union value can be tested to see if it is a particular variant.

if *int_or_str*$*is_str*(*int_form*) **then** ...

will not execute the expression after **then** because the test will return false. However

int_or_str$*is_int*(*int_form*)

will return true. The alternative syntax for fields of records can be used when projecting or testing unions.

int_form.proj_int
str_form.proj_str
int_form.is_int

As with records the variants may be procedures or types as well as values and it is possible to have two variants with the same signature.

let *a_union* == **union** (*one*, *another*: *integer*; *p*: **proc** (*integer*)*integer*)

The two variants *one* and *another* are different, so

a_union$*proj_one*(*a_union*$*inj_another*(99))

will result in an exception.

Syntax		
<union constructor>	::=	**union (** <field list> **)**

6.3. Structures

The third built-in type constructor makes **structure** types. Structures are very similar to records in that their values are composed of groups of objects. The difference is that there is an additional value *nil* in the type and there are operations to compare structure values. Structures are mostly used in recursive declarations to create lists and trees. In fact most structures can be represented using record and union constructors but they are useful enough to be provided separately.

letrec *int_list* == **struct** (*hd*: *integer*; *tl*: *int_list*)

This has created a type which can construct lists of integers. It has five operations together with the the *nil* value. *constr* can be used to make *int_list* values.

let *a_list* == *int_list*$*constr*(1, *int_list*$*constr*(2, *int_list*$*nil*))

The *nil* value is used to end the list. Without it there would be no way to construct a structure since a value of the type is needed before a node can be made. The selector procedures, *hd* and *tl* are used to select the parts of the structure in the same way as for a record.

int_list$*hd*(*a_list*) *int_list*$*hd*(*int_list*$*tl*(*a_list*))

If a selector is applied to nil an exception *nilreference* is raised, since there is no value that can be returned. There are two operations = and <> which compare two structure values. = only returns *true* if they the structures are **identical**, that is they were made with the same call of *constr* or they are both *nil*.

let *b_list* == *int_list*$*constr*(2, *int_list*$*nil*)

has made a list with the same *hd* and *tl* values as the tail of *a_list* but

b_list = *a_list.tl*

will return *false*.

Syntax		
<structure constructor>	::=	**struct (** <field list> **)**

Chapter 7. Type Constructor

As well as the using the constructors described above it is possible to make a type by extending an existing one. This usually involves adding new procedures or replacing existing ones.

let *new_int* ==
type (*int*) **extends** *integer*;
let *sqr* == **proc** (*i*: *int*)*int* . *i***i*
end

This declares *new_int* to be a type which is an **extension** of the existing type *integer*. The name in brackets, *int*, is used inside the constructor to represent the type being made. For instance the parameter and result of *sqr* both have type *int*. The result of this constructor is a type which has all the operations which *integer* had, but in addition it has a *sqr* procedure which returns the square of its parameter. This new type is different from *integer* so it cannot be used directly on values with the integer type. The conversion operation *up* must be used to make an *integer* value into a *new_int* one.

new_int$*sqr*(*new_int*$*up*(99))

There is a similar operation *down* which will convert values in the opposite direction

10 + *new_int*$*down*(*new_int*$*sqr*(*new_int*$*up*(11)))

These two operations are added to the new type when an old type is being extended to allow conversion of values from the old to the new types. They can be redefined if necessary or, as we shall see, "hidden" so that conversion of values is not possible.

The above example shows how a new type can be made which is slightly different from an existing one.

7.1. A New Type

The same approach is used to construct a completely new type. We have already seen that a record can be used to make a pair of integers and this pair can be extended to make a double precision integer type. 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 x.lo = 9999
then d$constr(succ(x.hi), 0)
else if x.hi < 0& x.lo = 0
then d$constr(succ(x.hi), ~9999)
else d$constr(x.hi, succ(x.lo))
end;
let pred ==
proc (x:d)d
begin
if x.lo = ~9999
then d$ constr(pred(x.hi), 0)
else if x.hi > 0 & x.lo = 0
then d$constr(pred(x.hi), 9999)
else d$constr(x.hi, pred(x.lo))
end;
let zero == d$ constr(0,0);
let iszero == proc (x:d) boolean . x.hi = 0 & x.lo = 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*. Of course they can be included in the type if required.

7.2. Information Hiding

As it stands this type includes the operations *hi*, *lo* and *constr* which were inherited from the record type as well as the new operations. These old operations are a nuisance, they are not part of the double-precision type

as such and they provide a security risk since it would be possible for someone to produce a value which appeared to be a double-precision number but, for example, had a positive high order part and a negative low order part. Unwanted operations can be masked out by giving an explicit signature which contains only those operations which are actually required.

```
let dble:
type (d)
succ, pred: proc (d)d;
zero: d;
iszero: proc (d)boolean
end
== dp;
```

This has created a new type *dble* which takes objects from *dp* but only takes those which are explicitly given in the type signature. It is impossible to create a value of the *dble* type or take it apart except with the given operations. An alternative would have been to have given the explicit signature in the original declaration.

```
let dp:
type (d)
succ, pred: proc (d)d;
zero: d;
iszero: proc (d)boolean
end
==
type (d) extends ... { The body of dp as before. }
end
```

7.3. Conversions

This double-precision type suffers from the problem that the only constant value is *zero*. All other values have to be made by using *succ* or *pred*. It would be convenient if other constants could be made. One way would be to define a procedure inside the type constructor which would convert an *integer* value into a *dble* one.

```
let make_double == proc (int: integer)d. d$constr(0, int)
```

This assumes that no *integer* value is greater than 10000. If larger *integer* values are possible then the body of the procedure would be

```
d$constr(int div 10000, int mod 10000)
```

integer values can now be made into *dble* ones.

```
dble$make_double(42)
```

The maximum value is limited by the maximum possible *integer*, so very large double-precision numbers still cannot be made. It would be nice to be able to have large literal constants such as 12345678 in a program. A

solution to this is to convert literals directly from their string representations i.e. from the string "12345678". This is done by defining a conversion procedure with the name *convertn* inside the type.

```
let convertn ==
proc (rep: string)d
begin
letrec getch ==
{ Returns the result of converting the first i characters. }
proc (i: integer)d
begin
if i = 0
then zero
else
begin
let this_ch == rep sub i; { Obtains the ith. character }
if this_ch < '0' | this_ch > '9' { Must be a digit }
then raise conversionerror
else
{Convert the first i-1 characters}
{then multiply by 10 and add in this digit}
getch(i-1)* d$ make_value(10) + d$ make_value(ord(this_ch) - ord('0'))
end
end;
getch(string$ length(rep))
end
```

This procedure reads the string and converts it into the numeric value. If any of the characters is not a digit then it raises the exception *conversionerror*. We assume that + and * operations have been defined for the type.

With this operation it is possible to write expressions like

```
dble$12345678 + dble$99999
```

The compiler automatically generates a call to *dble$convertn* when it recognises these constants. It is usual to declare conversion routines as **early** so that the compiler will do the conversion, rather than leaving the conversion until the program is run. If a number is not preceeded by a type name then the conversion used is the value of *convertn* which is in scope. The standard header contains the binding

```
let convertn == integer$ convertn
```

so that numerical constants are converted to *integer* by default.

There are two other conversion operations, *convertc* for strings in single quotes, and *converts* for strings in double quotes. These default to *char$convertc* and *string$converts* respectively.

7.4. Generic Types

Types in Poly can be treated as ordinary values. We have already seen how the ability to pass types as

parameters to a procedure allows polymorphic operations, we shall now see how being able to return a type can be useful.

A type which could be used to hold lists of *integer* values was described above. It was defined as

letrec *int_list* == **struct** (*hd*: *integer*; *tl*: *int_list*)

A type for lists of strings would be almost identical.

letrec *str_list* == **struct** (*hd*: *string*; *tl*: *str_list*)

Indeed lists of any type could be defined in the same way. The signature of the type in each case is basically the same.

```
type (list)
hd: proc (list)integer;
tl: proc (list)list;
constr: proc (integer; list)list;
nil: list;
<>, = : proc (list; list)boolean
end
```

We can define a list type for an arbitrary element type using a procedure.

```
let list ==
proc (element_type: type end)
type (list)
hd: proc (list)element_type;
tl: proc (list)list;
constr: proc (element_type; list)list;
nil: list;
<>, = : proc (list; list)boolean
end .
begin
letrec list_type == struct (hd: element_type; tl: list_type);
list_type
end
```

This procedure can be applied to any type, since any type matches the empty type "**type end**".

```
let int_list == list(integer);
let str_list == list(string);
let dble_list == list(dble);
```

The result is always a list with the same operations, but different signatures.

```
let a_list == int_list$ constr(999, int_list$ nil);
let b_list == str_list$ constr("hello", str_list$ nil);
```

The list types are different types, so only operations of the correct type are possible.

int_list$ *hd*(*a_list*);
str_list$ *hd*(*b_list*)

are valid, but

int_list$ *hd*(*b_list*);
int_list$ *tl*(*b_list*);
let *c_list* == *int_list*$ *constr*(999, *b_list*)

are not.

Syntax		
<type constructor>	::=	**type** <internal name> <declarations> <extension> <declarations> **end**
<internal name>	::=	(<identifier>) | <empty>
<declarations>	::=	<dec list> | <empty>
<dec list>	::=	<declaration> | <dec list>; <declaration>
<extension>	::=	**extend** <expression> | <empty>

Chapter 8. Standard Definitions

Poly is extremely flexible because much of the system is built on top of the language rather than built into it. It can be changed as required by redefining or adding new definitions. The standard definitions contain types and procedures which are likely to be of use to many programmers. For efficiency reasons some are written in assembly code or are handled specially by the code generator, but they all have signatures like ordinary definitions and can be redefined if required.

8.1. Primitive Types

8.1.1. void

void is used as a form of place-holder when a type is expected. For example, procedures which have side effects but no result are considered as returning a value of type *void*. It has only one value *empty*, and its signature is simply

void :
type (*void*)
empty : *void*
end

8.1.2. boolean

Boolean is the type used in tests. It has two values *true* and *false*. The complete signature is

```
boolean :
type (boolean)
true, false : boolean;
& : proc infix 4(boolean; boolean)boolean;
| : proc infix 3(boolean; boolean)boolean;
~ : proc (boolean)boolean;
<>, = : proc infix 5(boolean; boolean)boolean;
repr : proc (boolean)string;
print : proc (boolean)
end
```

The **&**, **|** and **~** operations correspond to the normal *boolean* operations **AND** (the result is *true* only if both the arguments are *true*), **OR** (the result is *true* if either arguments are *true*) and **NOT** (the result is *true* if the argument is *false* and vice-versa). <> and = compare the two arguments and can be regarded as exclusive-OR and exclusive-NOR respectively. *repr* returns the string "true" if the argument is *true* and "false" if it is *false*. *print* prints the string representation on the terminal.

8.1.3. integer

The type *integer* is the basic type used for numbers. Its signature is

```
integer :
type (integer)
first, last, zero : integer;
+, - : proc infix 6(integer; integer)integer;
* , div, mod : proc infix 7(integer; integer)integer;
pred, succ, neg : proc (integer)integer;
~ : proc (integer)integer;
<, <=, <>, =, >, >= : proc infix 5(integer; integer)boolean;
convertn : proc (string)integer;
repr : proc (integer)string;
print : proc (integer);
end
```

first and *last* are the minimum and maximum values that an *integer* can have. *last* is frequently one less than the negative of *first*.

+ and - are the usual infix addition and subtraction operations. They raise the exception *range* if the result is outside the valid range.

* is the infix multiplication operator which also raises *range* is the result is out of range.

div is the division operator and *mod* returns the remainder. They both generate *divide* if they are asked to divide by zero, and *div* may generate *range* when *first* is divided by minus one.

pred and *succ* respectively subtract and add one to a number, raising *range* if the result is out of range.

neg returns the negative, raising *range* if its argument is *first*.

~ is the same as *neg*.

<, <=, <>, =, > and >= are the usual infix comparison operations.

convertn is the operation which converts literal constants into integers. It recognises strings of digits as decimal (base 10) values unless the first character is a '0' in which case it treats it as an octal value or hexadecimal if it starts with '0x'. *conversion* is raised if the string does not fit one of these forms or *rangeerror* if it is too large.

repr performs the reverse of *convertn* by generating a string from a number. It is always generated as a decimal number.

print prints the string representation on the terminal.

8.1.4. long_integer

long_integer is very similar to *integer* but it allows larger numbers to be handled. Its signature is

long_integer :
type (*long_integer*)
first, *last*, *zero* : *long_integer*;
+, **-** : **proc infix** 6(*long_integer*; *long_integer*)*long_integer*;
***** , *div*, *mod*: **proc infix** 7(*long_integer*; *long_integer*)*long_integer*;
pred, *succ*, *neg*: **proc** (*long_integer*)*long_integer*;
~ : **proc** (*long_integer*)*long_integer*;
<, <=, <>, =, >, >= : **proc infix** 5(*long_integer*; *long_integer*)*boolean*;
convertn : **proc** (*string*)*long_integer*;
repr : **proc** (*long_integer*)*string*;
print : **proc** (*long_integer*);
from_integer : **proc** (*integer*)*long_integer*;
to_integer : **proc** (*long_integer*)*integer*;
end

The signature is the same as that of integer with the addition of *from_integer* and *to_integer* which convert between *integer* and *long_integer*. *to_integer* generates a *rangeerror* exception if the value is too large to fit into an integer.

8.1.5. char

The type *char* is the type of character values. It has signature

char :
type (*char*)
first, *last* : *char*;
<, <=, <>, =, >, >= : **proc infix** 5(*char*; *char*)*boolean*;
convertc : **proc** (*string*)*char*;
pred, *succ* : **proc** (*char*)*char*;
repr : **proc** (*char*)*string*;
print : **proc** (*char*);

end

Characters are regarded as being ordered according to the underlying character code. The ordering will usually follow alphabetic order. *first* and *last* are the smallest and largest characters and *pred* and *succ* give the previous and succeeding characters according to the ordering. *pred* and *succ* raise the *range* exception if a value would be produced outside the range. The comparison operations compare values according to the ordering.

8.1.6. string

Character strings have type *string*.

string :
type (*string*)
mk : **proc** (*char*)*string*;
<, <=, <>, =, >, >= : **proc infix** 5(*string*; *string*)*boolean*;
converts : **proc** (*string*)*string*;
length : **proc** (*string*)*integer*;
print : **proc** (*string*);
repr : **proc** (*string*)*string*;
+ : **proc infix** 6(*string*; *string*)*string*;
sub : **proc infix** 8(*string*; *integer*)*char*;
substring : **proc** (*string*; *integer*; *integer*)*string*
end

mk converts a character into a single length string, while *sub* selects a character at a particular position. The character positions are numbered from 1 to the length of the string, returned by *length*. Selecting a character outside this range results in a *subscript* exception. Subcripting a zero length string will therefore always result in an exception. *substring* extracts a string from another. It takes as parameters a string, an integer which gives the starting position in the string, and a second integer which gives the number of characters to select.

string$*substring*("hello", 3,2);

results in the string "ll". If the first parameter is outside the string or there are not enough characters in the string then *subscript* is raised. Two strings can be concatenated by +.

8.2. Variables and Vectors

So far the language described has been purely applicative, that is procedures can be applied to values, but there is no way to change the value associated with an object. Variables and vectors can be created and used in Poly but they are not built into the type system.

8.2.1. new

Variables are created by the procedure *new* which has the following signature.

new : **proc [***base* : **type end]** (*base*)
type

assign : **proc** (*base*);
content : **proc** ()*base*
end

new is a procedure which takes a value of any type and returns a type with two operations *assign* and *content* as its result. For example,

let *v* == *new*(99);

declares *v* as a type with signature

v: **type**
assign : **proc** (*integer*);
content : **proc** ()*integer*
end

The type is here being used as a way of packaging together a pair of procedures, there is no such thing as a value of this type.

The parameter type of *assign* and the result of *content* were found from the type of the original argument (99 has type *integer*). The current value associated with the variable is found using the *content* procedure.

v$*content*()

will return 99, the initial value associated with it. The value can be changed using *assign*.

v$*assign*(*v*$*content*()+1);

sets the value to 100.

Variables can be passed as parameters and returned as results from procedures like any other value. However, note that

let *vv* == *v*;

makes *vv* the same value as *v* which means that it refers to the same variable, and hence changes to *vv* will affect the value returned from *v* and vice versa.

It is not necessary to write "$*content*()" after every variable name in order to extract its value, the compiler will attempt to call the *content* object of a type if it is given one when it expects an ordinary value. There is also an infix assignment operator defined in the standard header which allows use of the normal syntax for assignment.

v := *v*+1

is therefore equivalent to

v$*assign*(*v*$*content*()+1)

8.2.2. vector

vector is a procedure which creates an array of variables.

vector: **proc [***base* : **type end]** (*size*: *integer*; *initial_value*: *base*)
type
sub: **proc** (*integer*)
type
assign : **proc** (*base*);
content : **proc** ()*base*
end;
first, *last*: *integer*
end

It takes as parameters the size of the array (i.e. the number of variables) and a value which is the initial value for each.

let *v* == *vector*(10, "init")

A particular variable is selected by applying the *sub* procedure to a number between 1 and the size (the **index**). The result will be a variable which can be assigned a value, or its value can be read.

v$*sub*(4)
v$*sub*(5) := "new string"

Attempting to apply *sub* to a value outside the range causes a *subscript* exception.

first and *last* are two integer values that are set to the minimum and maximum index values (1 and the size respectively). If the size parameter given to *vector* is less than 1 it will raise a *range* exception.

8.3. Iterators

Many programs involve the processing of lists or sets of values processing each one or searching for one which satisfies some condition. The standard header contains definitions to make these easier. All of these work using a standard interface, a type, called an **iterator** which represents an abstract sequence of values. An iterator has the following signature.

type (*iterator*)
continue : **proc** (*iterator*)*boolean*;
init : **proc** ()*iterator*;
next : **proc** (*iterator*)*iterator*;
value : **proc** (*iterator*)*base_type*
end

Values of the iterator type are elements of a sequence such that each has a value of some base type associated with it and a way of getting to the next element. They can be regarded as elements of a list, but equally they can be a range of integer values. *init* generates the first element of the sequence, and *continue* tests it to see if is a valid entry (the sequence may be empty or exhausted). If it is valid then *value* may be used to extract the associated value and *next* used to return the next element in the sequence. To see how this works we will examine two procedures which use iterators.

8.3.1. for

The *for* procedure is designed to apply a given procedure to every member of a sequence. Its signature is

```
for : proc [base : type end ]
(iterator :
type (iterator)
continue : proc (iterator)boolean;
init : proc ()iterator;
next : proc (iterator)iterator;
value : proc (iterator)base
end;
body: proc (base))
```

It takes an iterator and applies the procedure *body* to each element in turn. The body of *for* in Poly is

```
begin
letrec successor ==
{ Loop until the condition is false }
proc inline (counter: iterator)
begin
if counter.continue
then
begin
body(counter.value);
successor(counter.next)
end
end { successor };
successor(iterator$init()) { The initial value of the iterator. }
end { for };
```

The *successor* loops generating elements of the sequence and applying *body* to each value until the sequence is exhausted.

8.3.2. first

The other procedure which operates on iterators is *first* which searches a sequence until a condition is found. It has signature

```
first : proc [base, result: type end ]
(iterator :
type (iterator)
continue : proc (iterator)boolean;
init : proc ()iterator;
next : proc (iterator)iterator;
value : proc (iterator)base
end;
test: proc (base)boolean;
```

success: **proc** (*base*)*result*;
failure: **proc** ()*result*
)*result*

As well as the iterator, *first* takes three other explicit parameters, all procedures. The first is the test which is applied to each value. If it succeeds (returns *true*) then the *success* procedure is called with the value as its parameter. If the sequence is exhausted before the test has succeeded the *failure* procedure is invoked. The result of *first* is the result of either *success* or *failure*.

Chapter 9. Compiler and Environment

This part of the system is still under development and is not guaranteed to remain stable. Parts of it are also heavily UNIX dependent.

The current environment support provides facilities for compiling text files and remaking a system from its composite modules, compiling those which have been modified. There is a simple history mechanism for re-executing commands.

The system is used interactively with Poly expressions and declarations being typed in by the user and the reponses printed by the computer. Poly is used as a command language as well as a programming language, so all commands are simply Poly procedure calls and have their signatures checked by the compiler. Commands must either return values of type *void*, in which case they are simply executed, or they must return values of a type which has a print operation so that the result can be printed. Variables and procedures with no parameters are allowed provided their results are void or can be printed.

9.1. environ

environ is the type which is the nearest equivalent to a file directory in Poly. It has signature

environ :
type (*environ*)
enter : **proc** (*environ*; *string*; *declaration*);
lookup : **proc** (*environ*; *string*)*declaration*;
delete : **proc** (*environ*; *string*);
print : **proc** (*environ*);
in : **proc** (
type
enter : **proc** (*string*; *declaration*);
lookup : **proc** (*string*)*declaration*;
delete : **proc** (*string*);
over : **type** (*iter*)
continue : **proc** (*iter*)*boolean*;
init : **proc** ()*iter*;
next : **proc** (*iter*)*iter*;
value : **proc** (*iter*)*declaration*
end
end
)*environ*;

out : **proc** (*environ*)
type
enter : **proc** (*string*; *declaration*);
lookup : **proc** (*string*)*declaration*;
delete : **proc** (*string*);
over : **type** (*iter*)
continue : **proc** (*iter*)*boolean*;
init : **proc** ()*iter*;
next : **proc** (*iter*)*iter*;
value : **proc** (*iter*)*declaration*
end
end
end

declaration is a type which the compiler uses to represent objects that it has created.

A value of the *environ* type is a set of procedures which map strings onto *declaration* values. The compiler uses the value of *current_env* as the environment in which to compile something. It uses the *lookup* procedure to find the value and signature of identifiers and calls *enter* to store the result of making declarations. A particular value of the environ type is made by using the *in* procedure to package up a type with the appropriate operations. The inverse operation *out* can be used to extract the type.

There is a procedure *mkenv* which can be used to create environ values. It has signature

mkenv : **proc** (*environ*)*environ*

It returns an environment which can be seen as an extension of the environment which was given as the parameter. New declarations result in entries in this new environment and they can be found by using the identifier. However, looking up an identifier which has not been declared in this environment results in a search in the environment originally passed as the parameter. This can be regarded in the same way as nested declarations in Poly where an identifier is first looked up in the current block and if it is not found there the enclosing blocks are searched. Typically *mkenv* is called with either the current environment or the global environment as parameter.

let *new_env* == *mkenv*(*current_env*);
current_env := *newenv*;
let *new_env* == *mkenv*(*global_env*);

The global environment contains declarations such as integer which it is expected that nearly all programs will require.

The computation involved when entering or looking up an identifier may be considerably more than just operating on a table. The *make* procedure, for example, uses an environment in which looking up an identifier may involve recursive calls to the compiler to compile the object.

9.2. ?

? prints the signature of an object which has previously been declared. It has signature

? : **proc** (*string*)

For example, the statement

? "?";

prints

? : **proc** (*string*)

It is useful to be able to check the signature of an object before using it.

9.3.

runs a text file through the compiler and executes the result. It has signature

\# : **proc** (*string*)

At present the string parameter it takes is an ordinary UNIX file name, without any processing of wild-cards.

9.4. sh

sh runs a line of text through the UNIX shell. It can be used to execute any command, including starting up interactive programs. It has signature

sh : **proc** (*string*)

For example,

sh "emacs fred";

will start up and run the "emacs" editor on a file called "fred". The Poly system will wait until the process is finished before continuing.

9.5. make

The *make* command in Poly is similar in function to the "make" command under UNIX. It constructs a Poly object by recompiling only those parts of it which have changed since it was last made.

It is generally good programming practice to break a large program down into several parts, usually called modules, and develop each independently. A module usually provides several related functions and so can be represented in Poly as a "type". Such types may or may not have values belonging to them. For example, a module to construct stacks could be the type "stack" and all stacks would be of that type, but a module for a set of trigonometrical functions would be simply a set of related procedures.

A module may be complete in itself or require other modules to make it work. The latter case is represented in Poly by a procedure which takes some types as parameters and returns a type as the result. So, for example, a module for a parser may use modules for the symbols and for the parse tree.

An important point about these modules is that each can be compiled independently and then the program can

be made by applying the modules which are procedures to their parameters. The process of applying a module to other modules is known as *binding*. Like any other procedure application in Poly it is subject to the normal rules for signature matching.

When a module is changed, for example to correct a bug, it must be recompiled and rebound. The purpose of the make procedure is to ensure that everything which must be recompiled has been and to rebind all the necessary modules. Note that a change to the signature of the module may require changes to other modules that use it, otherwise a signature fault may be generated by the compiler. A change of signature may not always require all the using modules to be recompiled. For example, a module which is a type may have several operations used by different using modules. Changing the signature of one of these operations will require changes only to those modules which actually use that operation.

The make procedure assumes that the source text of the modules is held in some UNIX text files in a set of related directories. As an example suppose we have a set of modules which are combined in the following fashion to make a program.

let *a* == *b*(*c*, *d*);
let *e* == *f*(*g*, *h*);
let *i* == *j*(*a*, *e*, *h*);
let *z* == *k*(*i*, *e*);

z is the result of binding the modules together and is the final program. The source text is arranged in a series of directories with the root directory called **z**.

z contains **k**, **i** and **e** and **h**.
z/k is the source text for *k*.
z/i is a directory containing **j** and **a**.
z/i/j is the source text for *j*.
z/i/a is a directory containing source files **b**, **c** and **d**. }
z/e is a directory containing source texts **f** and **g**.
z/h is the source text for *h*.

In addition each directory has a file called **poly_bind** which are the instructions for binding together the modules to make the result.

z/poly_bind contains "**let** *z* == *k*(*i*, *e*);"
i/poly_bind contains "**let** *i* == *j*(*a*, *e*, *h*);"
e/poly_bind contains "**let** *e* == *f*(*g*, *h*);"
a/poly_bind contains "**let** *a* == *b*(*c*, *d*);"

Supposing **h** has been modified and we wish to remake *z*. The command

make "z";

looks for a file "*z*" and examines its access permission. It discovers that it is a directory and so tries to compile the file "*z*/poly_bind". This contains the command

let *z* == *k*(*i*, *e*);

For each identifier in the command it looks up a file with that name in the current directory and only if that

fails does it treat it as an ordinary identifier. **k** is a text file so it compares the time that it was last modified (kept by the operating system) with the time on which an identifier called *k* was last declared (kept by the Poly system). It sees that the file has not been modified since *k* was declared so it uses that declaration. If it had found that the file was newer it would recompile **k** (by a recursive call to the compiler) before returning the newly compiled version. It does not perform any other consistency checks relying on the type checking to ensure that *k* really is a procedure which can correctly be applied to *i* and *e*.

It next encounters *i* which it discovers is a directory and so it executes the file **z/i/poly_bind**. **j** is treated in the same way as **k**, but **a** is again a directory. It recurses again to process **a** and checks **b**, **c** and **d**. Finding that all these are text files and are up to date and that *a* is newer than each of them, it concludes that *a* is up to date and uses its current value without rebinding.

e, being a directory, is processed in the same way as **a**. *f* and *g* are both found to be up to date, but **h** is not found in the directory. The directories are regarded as nested blocks so that if a file is not found in the current directory the make program looks in the immediately enclosing one (i.e. the parent directory). It fails to find **h** in **i** and so tries **z**. There it finds the source text for *h* and discovers that it has been modified and must be recompiled. It recompiles it, returning the newly compiled *h* as its result. *e* must now be rebound so the declaration is executed and the new value returned as the result.

The next identifier in the declaration of *i* is *h* itself. The program remembers that *h* has been checked and uses the new value, rather than repeating the check on when the files were modified. It does this whether it has recompiled the file or just checked that it does not need recompiling. It executes the declaration of *i* because *e* and *h* have been remade and returns this as its result.

In the declaration of *z* the next identifier is *e*. Again it uses the fact that *e* has been checked to save processing the declaration of *e* again. Finally it can rebind *z* and the construction is complete. If make is rerun immediately after this it will simply check everything and not rebind any of the files.

Note that each file must be "in scope" when it is required. Because **h** is used by both **i** and **e** it must be in the path to both of them i.e. in the **z** directory.

9.6. Persistent Storage

The Poly system runs under a persistent storage system, that is any declarations of identifiers or values in variables can be retained from one session to the next on permanent storage. The database is held on a file and objects are read in from it as required. Once read in they are retained in store until the end of the session when those which are to be retained are written out again. The criterion for writing something out to the database is whether it is reachable from the root procedure which is the one used when Poly is started up. In the normal Poly system this essentially means that any declarations made in the global environment will be retained. When the user exits normally from Poly all the reachable objects are written back and the database is updated. The database can also be written back by executing the procedure *commit*(); which writes back the database and exits from Poly. It is currently not possible to write the database and continue.

9.7. history

The normal Poly system reads commands from the input stream, usually the terminal, and compiles and executes them. It also remembers the last few commands typed so that they can be re-executed if necessary. The commands in the table can be printed by the *history* procedure.

history();

There are three procedures which execute commands from the history table. Each command prints the command before executing it, and also enters the command it will execute in the history table. The previous command can be executed by the !!procedure.

!!();

Another command can be executed using the **!-** procedure. It has signature

!- : **proc** (*integer*)

The integer parameter is the number of the command counting back from the current one, so

!- 1;

is equivalent to

!!();

The third command **!** has signature

! : **proc** (*string*)

The string is the first few characters of the command to be executed, so to re-execute the last declaration,

! "let";

can be used. The command found is the first one whose characters match, working from the last command back.