Technical Report

UCAM-CL-TR-28
ISSN 1476-2986

Number 28

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Poly report

D.C.J. Matthews

August 1982

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

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

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

POLY REPORT

D.C.J. Matthews, August 1982

Computer Laboratory,

University of Cambridge

Abstract

Poly was designed to provide a programming system with the same flexibility as a dynamically typed language but without the run-time oveheads. The type system, based on that of Russell allows polymorphic operations to be used to manipulate abstract objects, but with all the type checking being done at compile-time. Types may be passed explicitly or by inference as parameters to procedures, and may be returned from procedures. Overloading of names and generic types can be simulated by using the general procedure mechanism. Despite the generality of the language, or perhaps because of it, the type system is very simple, consisting of only three classes of object. There is an exception mechanism, similar to that of CLU, and the exceptions raised in a procedure are considered as part of its 'type'. The construction of abstract objects and hiding of internal details of the representation come naturally out of the type system.

1. INTRODUCTION

Poly was designed to provide a programming system with the same flexibility as a dynamically typed language but without the run-time oveheads. The type system, based on that of Russell [1, 2] allows polymorphic operations to be used to manipulate abstract objects, but with all the type checking being done at compile-time. Types may be passed explicitly or by inference as parameters to procedures, and may be returned from procedures. Overloading of names and generic types can be simulated by using the general procedure mechanism. Despite the generality of the language, or perhaps because of it, the type system is very simple, consisting of only three classes of object. There is an exception mechanism, similar to that of CLU [3], and the exceptions raised in a procedure are considered as part of its 'type'. The construction of abstract objects and hiding of internal details of the representation come naturally out of the type system.

1.1 Syntax

The description of the syntax in this report follows the Russell and Alphard reports in the use of superscripts $^{\#}$ $^{+}$ * to denote respectively optional items, repetition with at least one occurrence, and repetition with possibly no occurrence. Subscript symbols occur as separators between occurrences of the item. Braces { and } are used as meta-brackets.

For instance
const$^{\#}$ <name> is equivalent to
const <name> | <name>
and
<identifier>$_{,}^{+}$ is equivalent to
<identifier> | <identifier>,<identifier>|
<identifier>,<identifier>,<identifier> | ....

Reserved words, shown underlined in this report, may be written in upper, lower or mixed case. In identifiers the case of letters is significant and two identifiers are distinct if any of their letters are written in different cases.

<identifier> ::= <letter> | <letter><letter or digit>* | <symbol>*
<letter> ::= a|b|c|d|....|z|A|B|C|D|....|Z
<digit> ::= 0|1|2|3|4|5|6|7|8|9|.
<letter or digit> ::= <letter>|<digit>|
<symbol> ::= +|-|*|/|\|!|#|$|%|&|=|<|>|?|_|~|^

Comments are written by enclosing them in braces ({ and }). Words are separated by one or more spaces, newlines or comments. The following words have special meaning and cannot be used as ordinary identifiers.

==	:	any	begin	catch
const	do	early	else	end
extends	if	inline	infix	let
letrec	prefix	proc	raise	raises
record	then	type	union	while

2. SPECIFICATION CHECKING

Every object in Poly has both a value and a specification. The value is what is used when the object is used, the specification describes what can be done with it. There are three main classes of objects; constants, procedures and types. Exceptions (signals) could be regarded as a fourth class though they cannot be used in the same way as the other classes.

Constants are simple values which can be manipulated but have no visible structure of their own. A constant is the implementation of an abstract object.

Procedures are operations which can manipulate constants, other procedures or types. They may return objects which may be constants, procedures or types or they may raise exceptions. A procedure implements an abstract operation.

Types are simply sets of named objects. They may, like conventional types, have values belonging to them or they may be simply modules. A type implements an abstract set of co-operating operations which can together manipulate objects.

```
<specification> ::= <constant specification>
                  | <procedure specification>
                  | <type specification>
```

The specification of an object is checked when it is used in some context, either as a parameter to a procedure, or when an identifier is declared with an explicit specification. The object and the context must be of the same class (constant, procedure or type) and must satisfy the rules for that class. The rules themselves are given in the following sections.

2.1 Constants

<constant specification> ::= const# <name>

<name> ::= <name>$<identifier> | <identifier>

A constant is a simple unstructured value. It has no properties of its own and can only be manipulated by certain operations. All constants belong to a named type, which is the set of operations which can correctly interpret it. New procedures can be written which operate on a constant but they will always be written using existing operations from the type.

The specification of a constant is T (or const T) where T is some type name. It is then said to have type T. A value with type T can only be used in a context requiring a value of type T. This rule is similar to the name equivalence rule for type checking in other languages. Two different type names are incompatible even if they are derived from the same declaration.

e.g. let S,T == integer

creates S and T with all the operations of integer but values of types S and T cannot be combined with integer values or with each other.

2.2 Procedures

```
<procedure specification> ::= proc <operator mode>#
                             <implied argument list>#
                             <explicit argument list>
                             <result specification>#
                             {raises <exception list>}#

<implied argument list> ::= [ <argument list> ]
<explicit argument list> ::= ( <argument list> )

<argument list>          ::= <argument specification>*;

<argument specification> ::= <identifier list> : <specification>
                           | <specification>

<operator mode>          ::= infix | prefix

<exception list>         ::= any | <identifier>*,
```

Procedures constitute the most complex class of objects. In general a procedure takes objects as parameters, alters the global state and either returns a result or raises an exception. The specification of a procedure contains the specifications of its arguments and of its result. It also indicates whether the procedure is to be used as an operator, and what exceptions it may raise.

A procedure may have two argument lists, the explicit arguments and the implied arguments. The implied arguments must all be types and must be referred to by the explicit arguments. As far as checking the specification of a procedure in a context is concerned the two argument lists are considered as one, the only difference comes when the procedure is called. Only the arguments listed in the explicit argument list need be given, the ones in the implied argument list will be inferred from the explicit arguments.

Any arguments which are types may be used in the specifications of subsequent arguments or in the result. This allows polymorphic procedures. For instance

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

is the specification of a procedure which takes any type together with a value of that type and returns as result a value of the type. An alternative representation, using an implied parameter would be

```
proc [t : type end] (t)t
```

which would be compatible with the previous specification but when called only the constant would be supplied.

A procedure matches a given context if corresponding arguments in the value and the context have the same specifications and the result specifications are the same. For the specifications to match they must be the same except that where a specification refers to a preceeding type name in one argument list the corresponding specification in the other list must refer to the corresponding name. Apart from this the names of the arguments are ignored in the matching process. For instance

proc (tt : type end ; x : tt)tt

is the same as the examples above and would match them correctly. The exception lists have to match in that every exception listed with the procedure value must appear in the exception list of the context. An exception list with the word any is considered being the set of all possible exceptions.

2.3 Types

<type specification> ::= type {(<identifier>)}$^{\#}$
{<identifier list>:<specification>}*;
end

A type is a collection of attributes: procedures, constants or types. Its specification is the list of the names of the attributes, together with their specifications. The specification of a type is type (T) x : A; y : B; z : C;... end where t, the internal name, represents the type within the specifications A;B;C... The ordering of the attributes is irrelevant. A type value matches a context if every attribute in the specification of the context appears in the specification of the value. In other words, attributes may be lost from a type value to make it match a context, but if any required attribute is not present or has the wrong specification then the value will not match. For instance a type value with specification

type (i) zero: i; succ: proc(i)i end

would match a context with specification

type (i) succ: proc(i)i end

but not the other way round. The type specification of a context can be regarded as a filter which removes all attributes apart from those listed.

2.4 Coercions

There is one circumstance in which a coercion may be applied when an expression appears to break the above rules. It is included to allow the usual syntax of expressions using variables, when a variable is used to denote its current value. A variable in Poly is a type with two attributes, both procedures (see sections 5.4 and 5.5). 'assign' gives a new value to the variable, and 'content' returns its current value. If a type 't' is used in a context requiring a constant, 't' is replaced by 't$content()' if 't' has such an attribute. (i.e. the 'content' procedure of the type is called to return the current value).

3. STATEMENTS AND EXPRESSIONS

```
<expression>  ::= <if expression>
                | <while expression>
                | <infix expression>
                | <raise expression>
```

An expression describes a computation which returns a result and possibly has side-effects. All expressions in Poly return results. If a result is not returned explicitly then a value of 'void$empty' is returned. It is also returned from expressions like the 'while loop' which cannot return a general value.

3.1 Declarations

```
<declaration> ::= let <identifier>+, {: <specification>}#
                  == <expression>
                | letrec <identifier> {: <specification>}#
                  == <expression>
```

A declaration associates a name with a value. The name can then be used to represent the value in the block which contains the declarations and any inner blocks. Declarations can occur in compound expressions or type constructors. A declaration may contain a specification for the value bound to the name. This may be necessary to simplify checking when a complex expression is being bound or when the specification of the name is not the same as the expression.

Declaring a name in an inner scope will hide an outer declaration of the name; there is no overloading in Poly. Names may not be declared twice in the same scope. Names belonging to operations of a type are not automatically available in a scope where the type is available. The effect of overloading can often be achieved by overloading.

let and letrec differ in that letrec declares the identifier before the expression, making it available inside it, while let declares the identifiers afterwards. letrec must therefore be used for declaring recursive procedures. A declaration has scope from the point of declaration to the end of the block containing it. An identifier cannot be referred to before it is declared.

3.2 If Statement and If Expression

```
<if expression> ::= if <expression> then <expression>
                    else <expression>
                  | if <expression> then <expression>
```

The if expression causes an expression to be executed depending on the

value of the "guard" expression. The guard, which must have a result type of boolean, is evaluated and if it returns "true" the expression following the then is executed. If the guard is "false" the expression following the else is executed. The specifications of the values produced by the then-part and the else-part must be capable of being converted to a single specification, that of the result. The second form of the if expression, without the else-part, may only be used if the then-part returns a value of void$empty.

The ambiguity in the syntax of nested if expressions is resolved by requiring that an if-expression without an else-part may not be followed by else. An else-part is thus paired with the nearest unpaired then.

3.3 Compound Expression

```
<compound expression>  ::= begin
                             <expression block>
                             end
                         | ( <expression block> )

<expression block>     ::= {<declaration> | <expression>}*;
                           <catch expression>#
                           end

<catch expression>     ::= catch <expression>
```

The compound expression is used to introduce new identifiers and to group expressions together. All the expressions except the last must return a value of void$empty. The specification of the compound expression is the specification of the last expression. An empty compound expression or a compound expression containing only declarations returns void$empty.

The catch expression is used to trap any exceptions which may be raised in the expressions or declarations in the block. If an exception is raised within the block and not caught in an inner block, it may be caught at this level. The expression following the word catch must yield a procedure whose argument must have type 'string'. Its result must be similar to the result of the last expression in the compound expression (i.e. they must both be capable of being converted to a single specification, that of the result). When an exception is caught the name of the exception is passed as the parameter to this procedure and the result of the procedure is returned as the result of the compound expression. If there is no catch expression or a further exception is raised in the catch expression then it is propagated to the next level out where it may be caught or propagated further.

3.4 Operators

```
<infix expression>  ::= <infix expression> <operator>
                          <prefix expression>
                        | <prefix expression>

<prefix expression>  ::= <operator> <prefix expression>
                         |  <application>
```

The specification of a procedure can indicate that it is to be used as a prefix or infix operator. This allows a more convenient notation for some expressions than the procedure application, but does not affect the semantics. The precedence rules for operators are simple: All operators of the same mode (prefix or infix) have the same precedence, prefix operators being more binding than infix.

3.5 Procedure Application

```
<application> ::= <application> (<expression>*,)
                | <basic expression>

<basic expression> ::= <compound expression>
                     | <name>
                     | <manifest>
                     | <procedure constructor>
                     | <type constructor>
                     | <union type>
                     | <record type>
```

A procedure application causes the expression associated with the routine returned from the expression to be executed. The expressions in brackets, if any, provide values for the explicit formal parameters of the routine. The expressions must have specifications which match the specifications of the formal parameters There must be the same number of expressions as formal parameters in the explicit parameter list. The specifications of the parameters are matched from left to right, starting with the implied parameters. Each parameter is tested for a match using the rules in chapter 2. If it matches then any subsequent use of the formal parameter is renamed with the matched parameter value. The specification of the result of a procedure call is the result specification of the procedure, except that any formal parameters used are renamed with the actual parameter value. If 'plus' has specification

```
proc (inttype: type (i) + : proc (i; i)i end;
      x, y: inttype)inttype
```

and 'a' and 'b' have type integer then it can be correctly used as

```
plus(integer, a, b)
```

and the result will have type integer.

3.6 Names

<name> ::= <identifier> | <name>$<identifier>

A name yields the value given to it by its declaration. Names may be simple identifiers or a sequence of identifiers separated by the $ symbol. The first identifier must always have been declared in one of the currently open scopes. Subsequent names must be an attribute of the type referred to by the previous name, so all but the last must refer to a type. E.g. if 'atype' has specification

```
type (a)
x, y: proc(a)a;
z: type (z)
   p: proc (z)
   end
end
```

then

atype atype$x atype$y atype$z atype$z$p

are all valid names.

3.7 Manifests

<manifest> ::= <number>
| <single-quoted sequence>
| <double-quoted sequence>
<number> ::= <digit> <alphanumeric>*

<single-quoted sequence> ::= '<any char>*'

<double-quoted sequence> ::= "<any char>*"

Manifest constants are values which stand for themselves. There are three forms of manifest, the number and the single and double-quoted sequences.

```
0   9999   0x6f83   9xz   'X'   'hello'   '\n'   ''
""   "A message"
```

are examples of manifests. They can be converted to values of any type by defining a procedure "convertn" , "convertc" or "converts" to return a value of the appropriate type. For instance "convertn" for integer is defined as

```
convertn: proc(string)integer raises conversionerror
```

The compiler will act as though a call to the appropriate routine had been written and the conversion will be made. Prefixing a manifest with a type selector causes the compiler to use convertc, convertn or converts from that type.

3.8 Procedure Constructor

```
<procedure constructor> ::=*
      proc <operator mode>
      {[<argument list>]}# (<argument list>)
      <specification># {raises <exception list>}#
      <compound expression>
```

The procedure constructor creates procedure values. If a result specification is given the expression must return a value which satisfies it. If the result specification is omitted then the compound expression must return void$empty. The exceptions which may be raised in the compound expression must be equal to or a subset of those listed in the raises list. However omitting the raises list is taken to mean that a list should be made from the exceptions which may be raised in the compound expression.

3.9 Type Constructor

```
<type constructor> ::= type {(<identifier>)}#
                       {<declaration>;}*\
                       {extends <expression>;}#
                       <declaration>;* end
```

The type constructor makes a new type by collecting together a set of declarations. The declarations will usually be of procedures which provide additional operations to an existing type.

The "extends" clause defines an existing type as the basis for the new type. Any new operations can be written in terms of operations available on this type. For instance

```
let newint == type (int) extends integer;
                 let cube == proc(i: int)int
                    (i int$*  i int$* i)
              end;
```

declares "newint" to be like integer but with the new operation "cube" added. Its specification includes all the operations available for integer together with the new operation, however it is a completely separate type from integer. Values can be converted between the original type and the new type by means of two operations, 'up' and 'down' which are created when 'extends' is used. In this example they have specifications

```
up: proc (integer)int;      down: proc (int)integer
```

Within the declarations the identifier in parentheses, in this case "int", represents the type being created.

Existing operations on the base type may be overridden by declaring a new operation with the same name. If let is used to declare a new operation then it can be written using the existing one since the new operation will not replace the existing one until the end of the declaration. All the operations of the base type, together with any newly declared operations,

are returned by the type constructor. It is possible to hide operations by binding the result to a type name with a specification with fewer operations.

3.10 Union Type

<union type> ::= union ({<identifier>+:<specification>}*;)

The union type returns a type which is the union of the specifications listed. For each identifier x with specification T there are three operations on the union type. inj_x creates a union from a value of specification T and proj_x extracts a value of specification T from the union. is_x is a predicate which is true only if the union was created with inj_x. proj_x is only valid if is_x is true, otherwise "projectionerror" will be raised.

For instance the specification of the union created by union(x: S; y: T) is

```
type (U)
   is_x, is_y : proc (U)boolean;
   inj_x :      proc (S)U;
   inj_y :      proc (T)U;
   proj_x :     proc (U)S raises projectionerror;
   proj_y :     proc (U)T raises projectionerror
end
```

Two different identifiers listed with the same specification (e.g. union(x,y:T)) create different variants, so proj_y is not allowed on a union created with inj_x.

3.11 Record Type

<record type> ::= record ({<identifier>+:<specification>}*;)

The record type returns a type which is the Cartesian product of the specifications listed. The identifiers are declared as fields of the record and can be used as selecting procedures. The selecting procedures take a value of the record type as argument and return a value of the field specification as result. There is also a constructor procedure "constr" which makes a record out of values with the field specifications. For instance the record created by record(x: S; y: T); has specification

```
type (U)
   x      : proc(U)S;
   y      : proc(U)T;
   constr : proc(S; T)U
end
```

Because the fields may have any specification a record may be used to make types whose basic values are procedures or types. Hence procedure or

type variables can be used.

3.12 Raise Statement

<raise expression> ::= raise <identifier>

The raise expression causes the named exception to be raised. This causes further processing to be halted until the exception is caught. Working from the raise expression outwards each compound expression is examined until one is found which contains a catch phrase. If the exception is caught the corresponding procedure is executed and processing continues.

If the exception is not caught within the immediately enclosing procedure and it has not been included in its "raises" list then the program is in error. Otherwise the exception is raised at the point of call of the procedure, and the exception propagated further.

The raise expression may form part of an expression even though it does not return a value. For the purpose of specification checking it appears to have the required specification for the context.

3.13 While Statement

<while expression> ::= while <expression> do <expression>

The while expression causes the expression after the do to be executed repeatedly until the expression, which must have a result type boolean, returns "false". The expression is evaluated before the expression and if it is "false" on entry the expression is never executed. The body of the while statement must return void$empty. and the while statement itself returns void$empty.

4. STANDARD DECLARATIONS

The chapter describes the declarations that should be provided by the compiler or standard library for any implementation.

4.1 Void

Specification
```
type (v) empty: v end
```

The type "void" has only one value, "empty". "empty" is returned as the result of operations which do not otherwise return a result.

4.2 Boolean

Specification
```
type (bool)
   true, false: bool;
   &, ¦ : proc infix (bool; bool) bool;
   ~ : proc prefix (bool) bool;
   print: proc(bool)
end
```

The type "boolean" is one of the few types which are actually built into the language. It is required because various constructions in the language such as if and while expressions use values of specification "boolean".

4.3 Integer

Specification
```
type (i)
   convertn: proc(string)i raises conversionerror;
   +,-,* : proc  infix (i; i) i
                  raises rangeerror;
   div, mod: proc infix (i; i) i
                   raises divideerror;
   succ,pred,neg,abs: proc(i) i raises rangeerror;
   =,<>,>=,<=,>,< : proc(i; i) boolean;
   print: proc(i)
end
```

Integer represents the positive and negative integers. "convertn" is invoked automatically to convert a number (i.e. a sequence of letters or digits beginning with a digit) into an integer value. It raises "conversionerror" if the characters do not form a valid integer.

4.4 New

Specification

```
proc [base: type end] (initialval: base)
   type
   assign:  proc (base);
   content: proc ()base
   end
```

"new" is a procedure which creates and initialises variables. A variable in Poly is a type containing a pair of procedures, one of which (cont) extracts the value currently held, and the other (assign) stores a new value in it.

4.5 Vector

Specification

```
proc [base: type end] (size: integer; initial: base)
    proc (index: integer)
       type
       assign:  proc (base)
       content: proc ()base
       end
    raises subscripterror
raises rangeerror
```

Vector constructs one dimensional arrays of values which can be indexed by an integer value. The value of index must be in the range from 1 to size (inclusive) otherwise 'subscripterror' will be raised. The result of indexing the vector is a variable so that the element can either be read or updated. All the elements are initialised to the value 'initial' when the array is constructed. The size of the vector must be at least 1 otherwise rangeerror will be raised.

4.6 Char

Specification

```
type (c)
   print: proc (c);
   succ, pred: proc (c)c raises rangeerror;
   =, <> : proc (c;c)boolean;
   convertc: proc (string)c;
end
```

The type 'char' represents the characters used to form readable text.

4.7 String

Specification

```
type (str)
   sub: proc infix (str; integer)char raises subscripterror;
   + : proc infix (str; str)str;
   =, <> : proc (str; str)boolean;
   length: proc (str)integer;
   converts: proc (str)str;
   print: proc (str);
   mk: proc(char)str
end
```

String is the type used for arguments to converts, convertn and convertc, and in a catch phrase. A constant of this type is regarded as a sequence of characters of unspecified length. The 'length' procedure gives the number of characters in a string, and 'sub' can be used obtain a particular character. Strings can be concatenated using '+' and compared using '=' and '<>'.

5. REFERENCES

[1]
Demers A. and Donahue J. Report on the Programming Language Russell
TR79-371 Department of Computer Science, Cornell University, 1979

[2]
Demers A. and Donahue J. Revised Report on Russell
TR 79-389 Department of Computer Science, Cornell University, 1979

[3]
Liskov B. et al. CLU Reference Manual
Lecture Notes in Computer Science No. 114, Springer-Verlag, 1981