Technical Report

UCAM-CL-TR-49
ISSN 1476-2986

Number 49

Programming language design with polymorphism

David Charles James Matthews

ISSN 1476-2986

CONTENTS

PREFACE

I am indebted to Dr. M. J. Gordon for his help and encouragement in my research and to my supervisor Prof. R. M. Needham.

I would also like to thank Dr. J. G. Mitchell whose suggestions prompted this work.

I am grateful to the Science and Engineering Research Council for their financial support.

I declare that this dissertation is not substantially the same as any I have submitted for a degree diploma or any other qualification at any other institution.

No part of this dissertation has been submitted for any such degree diploma or qualification.

Except where explicitly stated in the text, the work described in this dissertation is my own, and was done without collaboration.

1. INTRODUCTION

In this dissertation some topics in programming language design are explored. Several recent languages are examined and the design and implementation of a new language, Poly, is explained.

This chapter provides some background information on programming languages and contains an overview of the dissertation.

1.1 Motivation and Results

The basic motivation for the research was the desire for a simple yet general way of expressing many of the concepts used in modern programming languages. It was felt that many of the features of a language like Ada are presented by the language as a haphazard collection of facilities rather than a consistent whole. Recent work on type systems could be expected to provide the basis for a practical general purpose language with similar expressive power but in a language which was conceptually much simpler.

While it is possible to discuss programming language theory in the abstract, it was felt that a concrete design for a language was the best method for expressing my ideas. It would provide a means to test the ideas against other parts of the language and, if the language were implemented, allow practical testing of the whole system.

The result of the research was the design and implementation of a language known as Poly. Poly was designed to be a general purpose programming language which could be reasonably easily and efficiently implemented. It was designed around a type system which, by allowing far more flexibility than in conventional languages, should permit simpler and, hopefully, more reliable, programming.

1.2 Background

In order to put the research in perspective the remainder of this chapter is a discussion of various topics which are common to a number of programming languages.

The primary use of a programming language is as a tool for instructing a computer, but this is not its only purpose. It is important that programs should be readable, both for documentation and to aid modification. A language should make it easy to write correct programs and difficult to write wrong ones. This is, of course, extremely difficult to achieve, but developments in programming methodologies and languages are attempting to improve the situation.

1.2.1 **Types**

Much research on programming languages has concentrated on the concept of a type. A name in a program is often used to represent a value which is computed dynamically. It is not possible to predict what that value will be, it may depend on some values to be read in when the program is run, but certain information will be known about it beforehand. In many cases this information concerns the range of values which it may take or the operations which will be performed on it. Such information can be described as the type of the object associated with the name, or simply the type of the name.

For instance an object which can have only two possible values, "true" or "false" is described as being of type "boolean", whereas a value which can have any of the integral values in a certain (large) range may have type "integer". Alternatively we can say that "boolean" values are characterised by having "and" and "or" operations available on them, and that these operations have the property that only the values "true" or "false" can be produced. These two views of a type, as a set of all the possible values which an object of the type can take, or as a set of operations which manipulate values, are complementary views which will be explored in more detail later.

Types perform two functions in a programming language. They are used to ensure that operations are applied only to values of the appropriate type, and they can be used to select a particular operation from those with the same name (overloaded operations). Both type checking and operation selection can be performed when the program is run (dynamic typing) by associating a "type field" or "tag" with each value, but in all the languages to be considered in this dissertation the checking and selection are done at compile-time (strong typing). Performing the checking at compile-time has the advantages that less space is used at run-time since the type fields are not required and also that the type errors can be detected before the program is put into service. The disadvantage is that some programs which would be type correct if they were run will be rejected by a type system which works at compile-time. This is because the general case of whether a program is type correct is undecidable, so a compile-time type checking scheme which will reject all type incorrect programs must also reject some type correct ones as well.

1.2.2 Overloading and Polymorphism

Types have another use apart from for checking. An identifier may often represent more than one value, the correct one being found by using type information. A common example in many languages is that the symbol "+" can represent both the addition of integer values and real values. The selection of the correct operation to apply is done by the compiler on the basis of the types of the operands. This is an example of "overloading" (the name "+" is **overloaded** with more than one definition).

A related concept is **polymorphism.** A polymorphic operation can be applied to values of any type with a given structure and does not require a separate procedure for each type. For instance the identity function, which just returns the value of its argument, should be the same for all types. Most programming languages do not permit full polymorphism, though some languages have a restricted form. It is often possible to write a procedure with a parameter which is an array of arbitrary size. Among the languages to be discussed in chapter 2, two of them, Russell and ML, provide full polymorphic operations.

1.2.3 Abstraction

Work on programming methodologies has emphasised the development of programs by successively developing an abstract concept through various layers of abstractions to a concrete realisation as a program. For many years languages have included subroutines or procedures which implement an abstraction of control. Calling the procedure performs an operation which, while conceptually very simple, such as writing a value to a file, may in practice be extremely complex.

Data abstraction is a much more recent development but it can be viewed in the same way as procedural abstraction. Just as a procedure is a name for a potentially complex computation, so an abstract object is the name for a potentially complex data structure. As a procedure can be called in a very simple fashion to perform the computation, so an abstract object can be operated on by operations which manipulate the data structure which implements the abstract object.

Abstract objects can be written in many languages. A list structure can simulated in Fortran using arrays, and an array simulated in LISP using lists. Provided the abstract objects are manipulated using only the operations written specifically for the purpose they will faithfully represent their abstraction. However, if through mischief or just through a mistake, the structures are modified by an operation which goes "behind the back" of the specially written operations, the representation may no longer work properly.

1.2.4 Abstract Types

The use of abstract types combines the concept of abstract objects with a type system which ensures that an object is only modified by operations which preserve its integrity. An abstract type consists of information which gives the representation of an object of the type in terms of other types, and additional operations which will implement the abstract operations. Only the abstract operations are exported from the abstract type so none of the internal details are visible outside it. This means that an object declared to be of this type has an internal structure which is hidden from the user. There is no way that its internal data structures can be altered, or even examined, except by use of the abstract operations.

The use of abstract types has another advantage. Because the internal details are hidden it is much easier for the implementor of an abstract type to change its implementation. Provided the external interface remains the same any programs which make use of it should continue to work unchanged.

1.2.5 Parameterised Types

Just as procedures can be parameterised to allow them to perform a range of operations so an abstract type should be able to be parameterised to generate a range of types. Several of the languages to be discussed in chapter 2 allow parameterised (or "generic" types) but of those only Russell allows parameterisation at run-time. The others regard parameterised types as macros which must be expanded at compile-time. Parameterised types can present problems for type checking and the checking rules are often complex. Further complexity is introduced by the need to parameterise with types as well as with more conventional values. For instance, the type "stack" might be written as a type parameterised with the type of the elements which it could hold. As we shall see, it is possible to treat a parameterised type as a procedure, which allows a great deal more generality than macro expansion.

1.2.6 Exception Handling

A frequent problem in programs concerns the handling of errors. The most common example is attempting to divide a number by zero. This is usually detected by the hardware and results in some form of hardware trap, followed, in many languages, by catastrophic failure of the program. A program which forms part of an operating system, for instance, cannot be allowed to fail in this way so either the program must be proved never to fail or there must be a mechanism for recovering from such errors. A scheme for handling these kinds of errors is usually known as an "exception" system. An operation is said to "raise" an exception when it fails. Languages with exception systems usually allow a user program to raise an exception by a special statement.

There are many variations of exception system but they can generally be divided into two groups. One group, exemplified by PL/1, treats exceptions essentially as asynchronous events. Whenever an exception (error) occurs a handling procedure for that exception is entered which can return a suitable default value or set an error flag. Control then returns to a point immediately after the exception was raised.

The other method of dealing with exceptions, which is used in more modern languages, is to treat an exception almost as an "error" value which can be returned from a failing operation. Whenever an exception is raised the normal control flow is abandoned and a search is made for the nearest "exception handler". The handler can perform some tidying-up and raise the exception again or it can return a value suitable for the context in which it appears. In this scheme control continues after the handler rather than after the point at which the exception is raised.

1.2.7 Separate Compilation amd Modularity

The use of abstract types has been discussed from the point of view of convenience for the programmer and programming methodology. Another reason for their use is concerned with separate compilation.

Being able to compile parts of a program separately has two main advantages. First, it allows libraries of subprograms to be written and used by many other programs. Obvious examples are numerical subroutine libraries and input-output packages. Secondly, it allows a program to be developed, and often tested, in stages, so that it does not require the complete program to be written before testing can begin. Testing a single subroutine is generally easier than testing a complete system.

A module or package, as a unit of separate compilation is often called, is very similar to an abstract type. It has an interface which gives its properties while its internal implementation is usually hidden.

1.3 Other Topics in Programming Language Research

Two other topics which will be touched on in this dissertation but not explored in depth are program verification and parallel processing. They are both areas which have influenced programming language design, but where there is much scope for research.

1.3.1 Verification

An area of research which has had an influence on programming languages in recent years has been the attempts to prove the correctness of programs. The use of abstract types was prompted, particularly in the design of Alphard, by the help that they give to structuring a proof. Since the abstraction is separated from its implementation a proof of a program using the abstract type can be written in terms of the abstract properties, and it is only necessary to prove once that the implementation satisfies the abstraction. A change of implementation which does not change the abstraction requires only the implementation to be re-verified and not the uses of the abstraction.

1.3.2 Parallel Processing

Another area of research in programming languages concerns the handling of multiple processes. Languages intended for operating systems or embedded systems (the software part of a machine to perform a particular task) usually include some method of running and synchronising multiple processes. The choice of mechanism for synchronisation and multiple exclusion (monitors, messages, rendezvous, CSP's) has varied from one language to another and there seems to be no consensus on the "best" method to use.

1.4 Overview

The remainder of the dissertation contains a survey of several programming languages (chapter 2) and a description of the programming language Poly (chapter 3). Chapter 4 describes the design decisions which resulted in Poly and the type-system. In chapter 5 the implications of the type-system and the rest of the language are described. Chapter 6 covers the implementation and finally chapter 7 summarises the work and discusses areas for future research on Poly.

2. SURVEY OF PROGRAMMING LANGUAGES

This chapter consists of a survey of modern programming languages, Ada, CLU, Russell, ML and the Cedar Mesa Kernel. Similar concepts, particularly abstract types, appear in most of the languages so only the differences between the languages will be described.

2.1 Ada

Ada [DOD80, ICH79] was designed to a set of requirements issued by the American Department of Defense to provide a language for "embedded computer systems" [DOD77]. These requirements specified large parts of the language fairly closely so that, in effect, the language was designed by those who drew up the requirements. The one area where the designers of Ada where able to use their discretion was the mechanism for multitasking, since the requirements were least specific about them.

2.1.1 Type System

The Dod requirements which resulted in Ada were heavily influenced by Pascal. The conventional parts of the language therefore resemble Pascal. The language is "strongly typed" which is taken to mean that "the type of each expression shall be determinable at translation time". However a precise definition of type is not given except to say that differences of type excludes differences of range, precision, scale or representation, at least as far as the need for explicit type conversion is concerned. In practice the type checking rules are quite complex to deal with the various possibilities.

2.1.2 **Overloading**

Ada allows some names to be overloaded so that the same name can denote different operations. Only procedures, functions, operators and constants can be overloaded but unlike other languages (such as Algol68) which allow operators or functions to be overloaded this requires algorithms involving repeated scans over the parse tree to resolve the overloading.

2.1.3 **Generics**

A generic procedure or type in Ada is a template for a class of related objects. Types and procedures are not normally allowed as parameters but they can be used as parameters of a generic.

2.1.4 **Packages**

A package is the term used to describe a module in Ada. It can be regarded as a unit for separate compilation and also as a form of abstract type. A package has an external interface (the "public part" of the package) and a private internal representation. It is not a type in itself but it may export one or more types whose representations are hidden in the package.

2.1.5 **Tasking**

Parallel processing or "tasking" in Ada is one of the few areas in which the designers were able to use their initiative. The "Ironman" requirements were fairly vague on the subject and did not specify the mechanism to be used.

Processes communicate by means of "rendezvous". A rendezvous occurs when two processes, one executing an "entry" to a task and the other "accepting" it, run synchronously. The "accept" statement is similar to a procedure body and the "entry" to a procedure call. When a task executes an "entry" or "accept" it is suspended until the corresponding "accept" or "entry". The tasks then run in synchrony while executing the body of the "accept" and then can continue asynchronously. Parameters can be passed with the entry to transfer data between tasks.

2.1.6 Exceptions

Exceptions in Ada follow the pattern of most modern languages. An exception may be raised explicitly by the user or by an operation. An exception is associated with a name which describes the form of failure. When an exception is raised normal sequential control flow is abandoned and a search is made for the nearest handler. If this handler includes a trap for this named exception, or includes a default handler then the statement associated with the handler is executed. If the handler re-raises the exception, or does not include a trap for the exception, then the search resumes for the next handler, otherwise normal control resumes from the next statement.

A further complexity is introduced by the tasking mechanism. An exception raised in an "accept" statement causes an exception in both of the tasks involved in the "rendezvous".

2.1.7 Implementation

Various groups are working on implementations of Ada. So far no full implementation has been produced although several have been reported as being close to completion. As Ada is a large and complicated language compilers for it will be large and probably slow.

2.2 CLU

CLU [LIS81] is an object orientated language which resulted from work on abstract data types. It was one of the first languages with a true abstract type.

2.2.1 Objects in CLU

There are two basic elements in CLU, variables and objects. Objects are handled by references which can be stored in variables. A variable is simply a name for an object. Several variables can refer to the same object: this is the mechanism for parameter passing. An object can refer to another object and even to itself.

Instead of distinguishing between variables and constants CLU distinguishes mutable and immutable objects. An immutable object is one which, once it is created cannot be changed, whereas a mutable object can. An immutable object can contain a reference to a mutable object. It will always refer to the same object, though that object itself may change its value.

2.2.2 Clusters

A "cluster" is the CLU term for an abstract data type. It is a set of related procedures which have access to the implementation of an abstract type. A user of the cluster has access to the object only through those procedures. A cluster may be parameterised, but only by values which can be computed at compile-time. The values must belong to one of a few primitive types (integer, boolean etc.) or must be types themselves. The reason for this restriction is that these values are all immutable so that a parameterised cluster with certain parameter values will always denote the same value (see Russell below). When a type is used as a parameter it is possible to specify that it must possess certain operations, by means of a "where" clause. For instance

```
set = cluster[t:type] is create,insert,elements
   where t has equal:proctype(t,t)returns(bool)
```

is the header for a parameterised cluster implementing a set. The type parameter is required to have an "equal" operation which will compare two values of the type. Since the integer type, int, has such an operation set[int] is legal.

"Where" clauses can be used for particular operations inside a cluster. For instance the definition of a vector might contain operations such as "equal" which would compare two vectors elements by element and return "true" only if all the corresponding elements were the same. This requires the element type to have a suitable "equal" operation. "equal" will only be required by the vector comparison operation so it is not included in the "where" clause for "vector", but it must appear in the where clause associated with the comparison operation. This allows vectors to be made from types which do not have an "equal" operation, though it will not be possible to compare them.

2.2.3 Exceptions

Exceptions in CLU have similar semantics to Ada exceptions. Raising an exception causes normal sequential execution to be abandoned and control to be transferred to a handler for the exception. A handler can catch particular named exceptions or all the exceptions raised in a block. Exceptions can have parameters associated with them so as to return results to the handler. The corresponding handler can then either discard the values or use them.

The exceptions raised by a routine must be listed as part of its type specification. It is therefore possible to compute the exceptions which may be raised by a piece of program and check that only the exceptions listed in the header of a routine can be raised in it. There is a default exception "failure" which can be raised by any routine and any exceptions not caught are converted into "failure".

2.2.4 Implementation

CLU has been successfully implemented on a number of machines.

2.3 Russell

Russell [DEM79a, DEM79b, DEM80a, DEM80b] was based on the idea that a type is a set of operations and that it is a value in the same way as a number or a procedure. "The world" is viewed as a single value space which can be interpreted in many different ways. This is very similar to what happens in practice inside most conventional computers. A word can be regarded as a fixed-point value, an address or an instruction depending on how it is handled. The Russell view is that a type is a set of operations which gives an interpretation to this value space. For instance the type "boolean" is the set of operations "and", "or", "not" and two nullary functions (constants) "true" and "false". The actual representation of the two values "true" and "false" is irrelevant, it is sufficient that the operations of "boolean" have a consistent interpretation of the value space (i.e. that abstract properties such as "true and false = false" hold for all time).

2.3.1 Type Checking and Signatures

In order to ensure that the interpretation of a value is consistent, Russell imposes a type system on these values. To distinguish this type system, which works entirely at compile-time, from "types" which, being values, can exist at run-time, the word "signature" is used to denote the compile-time "type" of an object. For instance a boolean value has signature "val boolean". The signature checking rules allow only operations belonging to the type "boolean" to interpret these values: such operations have signatures with parameters whose signatures are also "val boolean". For instance the "and" operation of boolean will have signature

func [val boolean; val boolean] val boolean

in other words a function from two boolean values yielding a boolean value. The checking rules for values are simply that only operations with signatures "val T" can interpret values with signatures "val T", where "T", as well as being a name like "boolean", can be an arbitrary expression.

However the comparison is made on the basis of the static form of the expression "T" (i.e. the symbols which make it up) and not on the value which it represents, since the value may not be known at compile-time.

2.3.2 Type Completeness

One of the principles on which Russell was based was that of "type completeness". This essentially says that every expression should have a type which is expressible in the type system, and that it should be possible to write an expression in the language with any type which can be expressed. i.e. the type system should be a complete description of the language.

One consequence of this is that, if types are to be considered as values, the type system must be capable of specifying them, and the values which can be produced from those types. We have seen that type parameters in CLU can be specified by a "where" clause which requires a type to have one or more operations of particular forms. Although in CLU clusters with type parameters are always instantiated at compile-time, the same principle can be used if types are values at run-time. The signature of a type is expressed in Russell as the set of signatures of the operations which make up the type. Just as a "type" of a value in other languages gives information about a value which allows the compiler to check that it is correctly used, so this signature ensures that a type which may be the result of an expression can be correctly specified at compile-time. A type can be used as a parameter only if its signature contains all the operations, with the correct signatures, listed in the signature of the formal parameter. i.e. a subset of a type can be taken. In this way types as parameters and types as results of expressions are given signatures. Parameterised types are treated as procedures which return types as their results.

2.3.3 **Signatures of Values from Type Expressions**

There is a serious problem with allowing types as run-time values and that concerns the values which may be created by them. We have seen that the signature of a boolean value is "val boolean" which means that it can be correctly interpreted by the type "boolean". The signature checking system, working entirely at compile-time, must be certain that the word "boolean" always represents the same type at compile-time. More precisely, it must always represent a type which has the same interpretation of the value space. For example, if "boolean" were a variable which was initially set to a type which interpreted a word containing 0 as true and -1 as false and then changed to a type which made the reverse interpretation, then a value with signature "val boolean" created before the change would be misinterpreted. Worse situations can be envisaged. This problem does not occur in the other languages with parameterised types discussed in this chapter because the types are always parameterised at compile-time by values which must be constants.

2.3.4 **Variable-Free Expressions**

The solution to this problem in Russell is to prevent types from being variables. This is more complicated than it might at first appear, since types may be expressions. This means that any expression which returns a type must not contain a variable, but even this is not strong enough. In Russell no procedure may import a variable except as a parameter, since any procedure could be used in an expression which would return a type, even if it does not itself return one. An expression which does not involve any global variables has the property that its value is always the same in a particular scope. This is known as the "substitution property" because such an expression can be replaced with an identifier having that value in any other expression. This means that if types are always variable-free expressions then two type expressions give the same value if they have the same syntactic form.

The "import restriction" has a very serious impact on the rest of the language, making variables extremely difficult to use. For example input of data requires a file, treated as a variable, to be passed as a parameter to the "read" procedure. Since the file cannot be imported, any call to "read" inside a procedure requires that procedure to take the file as a parameter. A similar problem arises with the pointer type which requires a "heap" variable in order to indirect on a pointer.

2.3.5 Variables

The most obvious effect of the import rule is that variables must have a different signature from other values. The type completeness principle means that variables can be passed as parameters and returned from procedures just like other values. Variables are created by an operation called "New" which is a component of most types and constructors like "record". The "New" operation of a type returns a variable of the type. That is "T$New[]" returns a value with signature "var T". (T$ means, as in CLU, the operation belonging to the type T). The primitive operations on variables have to be carefully framed to preserve the substitution property. Operations like subscripting an array to give an element which may be assigned to, or selecting a field of a record, can only be applied to variables of the array or record types. This requires, for example, the subscript operation of an array to be overloaded so that an array value when subscripted gives a value of the element type, and a variable of the element type can only be obtained by subscripting an array variable. Pointer (or reference) values cannot be indirected on (dereferenced) directly to give a variable because that could break the substitution property. Instead the indirection operation takes a pointer and a "heap" variable to give a variable of the base type, in much the same way as subscripting an array.

2.3.6 Parameterisation

The ability of procedures to take types as parameters and return types as results in Russell allows parameterised types to be treated simply as procedures. It is possible to write a procedure "list" which takes a type as a parameter and returns a type with operations for manipulating lists of values of that type. It would have a signature similar to

```
func [base: type ()]
    type l ( nil : val l .....
```

2.3.7 Image

In Russell types are regarded as values which can be manipulated at run-time. They do not, however, have a "type" in the same way as a value such as a number. This means that variables which can directly contain types cannot be written because the operations which manipulate variables belong to the type of the variable. (e.g. integer variables, those with signature var integer, are manipulated by "integer$New" and "integer$:="). Since a type does not belong to a type there can be no type-valued variables.

In addition there are certain other operations which could be applied to types if they had a "type". For instance a list creating function can be written which takes a type as parameter and returns a type with operations for making lists of values of that type. This cannot be used to make lists of type values because type values do not have a type.

Instead of giving types a "type", the "image" construction is provided to "package up" types (or functions) as values of a type. "image" takes a signature of a type (or function) and returns a type with operations which allow the type (or function) to be packaged up into a value of the image type. For example if we wanted a variable which could contain different types with the "boolean" operations "and, or, not etc." then we would first have to create a type by using

```
image ( type bool ( and: func [val bool; val bool]val bool
        ....
      )
```

The type created has operations "In" and "Out" which map values from the "boolean" type to the image, and back again. The image type also has "New" and assignment operations so variables can be created and manipulated. To assign a particular "boolean" type to an image variable the type must first be "packaged up" by using the "In" operator, and then assigned to the variable.

"image" essentially provides a way of "circularising" the type-system so that instead of an infinite hierarchy of types, types of types, types of types of types etc. the "type" of a type can be expressed as a type. The type created by

```
image ( type () )
```

is the type of all types, including itself, since all types can be reduced to "type ()" by taking the empty subset.

2.3.8 Implementation

There is no current implementation of Russell although some work on it has been done.

2.4 ML

ML [GOR78] was based on Milner's [MIL78] theories that a program contains enough information for the type of any expression to be inferred without giving explicit type information. The type of any expression can be expressed as a set of equations involving "type variables" which can be solved. For instance the declaration

```
let f n = n+1;;
```

declares "f" as a function of n. The type of "f" can be deduced by solving the following set of equations. (T(n) denotes the type of n, # denotes

Cartesian product and x and z are type variables).

```
T(f) = x -> y;
T(1) = int;
T(+) = int # int -> int;
T(n) = T(first argument of +);
x = T(n);
y = T(result of +);
Hence f has type int -> int
```

This type inference system can be applied to more complicated expressions including recursive declarations.

2.4.1 Polymorphism

The set of equations which give the type of an expression may, like any other set of simultaneous equations, be inconsistent or insufficient. An inconsistent set of equations means that the expression is not type correct. An insufficient set means that the expression is polymorphic. For instance the declaration

```
let f = \n.n;;
```

declares "f" to be a function which returns its argument ("\" represent lambda). The set of equations for the type of "f" are

```
T(f) = x -> y;
T(n) = z;
x = z;
x = y;
Hence T(f) = z -> z, or using ML notation * -> *
```

This means that the function can be applied to an argument of any type and returns a value of the same type as its argument.

2.4.2 Type Operators

A type operator is similar to a type returning procedure in Russell. For instance "list" is a type operator which can be used to represent the type of all lists. "int list" is a list of integers, "* list" is a list of objects of any type (but all objects in the list have the same type). This is not quite the same as a list constructing function in Russell. In Russell an object can have type "list(integer)" and "list" can be passed as a function, so achieving the effect of "* list". Unlike a type returning function in Russell a type operator does not actually take a type as a parameter, indeed there is no concept of a particular instantiation of the type operator for a particular type.

Type operators can be written only in terms of records, unions, functions or other type operators. This ensures that a type operator always denotes the same value for particular parameters without requiring an import rule as in Russell. In fact, since a type operator does not depend on its "argument", it need never be instantiated with a particular type. It is does not need the import rule since, unlike a procedure in Russell, it is instantiated only once. A type operator is essentially a polymorphic type.

2.4.3 Exceptions

ML has an exception mechanism in which an expression can raise an exception instead of returning an ordinary result. An exception is a name which usually indicates the form of failure or the failing operation. An exception can be caught otherwise it propagates back to the caller.

2.4.4 Implementation

There are at least two distinct implementations of ML in existence. One implementation, developed at Edinburgh, compiles down to LISP, while the other, written in Pascal, compiles to machine code.

2.5 Cedar Mesa and the Cedar Kernel

Cedar Mesa is a development of the Mesa language [MIT79] designed and implemented at Xerox PARC. It is basically the same as Mesa but has among other things a garbage collector. Both Cedar and Mesa are large languages designed to support systems programming. They have a module system, parallel processing and coroutines. The basic language is fairly conventional and so will not be discussed in detail.

2.5.1 Cedar Mesa

Cedar Mesa is a development of the original Mesa language. There is very little difference at the language level but it supports a garbage collector and a form of dynamic typing. The facilities for dynamic type checking allow a pointer to be converted from pointing to a value of a particular type (ref T) to a pointer to any type (ref any) and back again. The conversion from ref any to ref T (narrowing) requires a dynamic check which is performed using information provided for the garbage collector.

2.5.2 The System Modeller

The System Modeller (SM) [SCH82] in the Cedar system has replaced the C/Mesa language used to describe the interfaces between Mesa modules. A Mesa (or Cedar) module may make use of other modules (import them) and it defines some routines, types or values for use outside (exports them). An interface file gives the names and types of objects, but no implementation. A module which claims to implement a particular interface file must export an implementation of every object in the file. There may be several implementations of an interface, for example a symbol-table may be implemented in different ways.

SM is a polymorphic applicative language which treats modules as functions which are parameterised by the modules and interfaces they import and return a value (the fully bound module) as the result. Treating the interface files as types and their implementations as values of those types means that the functions must be polymorphic.

2.5.3 The Cedar Kernel

The Cedar Kernel [LAM82] is an attempt to design a very small language by removing non-essentially facilities from Cedar. A Cedar program can be translated into a Kernel program by applying rewriting ("desugaring") rules.

It is intended that Cedar should eventually be a superset of the Kernel but at the moment the Kernel has several facilities not available in Cedar. The most important of these are that the Kernel allows types and bindings to be passed to and returned from procedures, while Cedar restricts them to the literal forms. This means that the Kernel, unlike Cedar, allows polymorphic operations so SM can also be considered as a (specialised) superset of the Kernel.

2.5.4 Types in the Kernel

Types in the Cedar Kernel are regarded as a cluster of operations (as in Russell) together with a predicate. The predicate is a test which can be applied to a value to determine whether it belongs to the type. This is a very general view and includes such things as range-checking as part of type-checking, but it will in general require run-time action.

2.5.5 Implementation

Mesa has been implemented on several machines at Xerox and large software projects have been developed in it. Cedar is currently under development. The Kernel is still in the design stage and there are no immediate plans to implement it.

2.6 Other Languages

The remainder of this chapter contains a brief description and critique of some other relevant languages.

2.6.1 Simula

The first language to provide any form of abstract type was Simula [DAH66, BIR73]. Simula was designed for writing discrete-event simulations and provided a process mechanism to represent the events. The "class" mechanism provided a form of abstract type since the operations on an object could be associated with the object. However the absence of any information hiding meant that a fully abstract type, with implementation details hidden from the user, was not possible. Simula allowed classes to be extended to make a new class with all the properties of the old one together with the additional properties.

2.6.2 Algol68

Algol68 [WIJ76] was the result of attempts to bring Algol60 more up to date with developments in language design. Like Ada it is a large language incorporating many features such as parallel-processing which are of limited use for most applications. It has no facilities for abstract types, and types (or "modes") are compared using a form of structural equality (i.e. two types are the same if they represent the same structure). The language relies heavily on automatic type conversion ("coercions") which have the advantage of conciseness but the risk of misunderstandings. It has infix and prefix operators, which may be overloaded, but unlike Ada all overloading must be resolvable on the basis of the argument types, and must not depend only on the result type. The concept of "orthogonality", that there should be one construction for each concept, and each construction should be applicable to any value, has had a considerable influence on language designers.

2.6.3 Pascal

Pascal [JEN75] was designed to a large extent as a reaction against Algol68. It stresses simplicity rather than power, possibly too strongly, but since it was designed as a teaching language, this is reasonable. It has no facilities for abstract types or separate compilation. It has some interesting features, such as sets and subrange types, which allow bit-strings and different lengths of integers to be treated in the language as high level concepts. Perhaps its most significant contribution has been as a way of introducing many users to high-level languages by providing them with an easily implemented and easily learnt language. It has been the basis, with Simula, of many of the modern high-level languages.

2.6.4 Euclid

Euclid [LAM77] was based on Pascal but designed for systems programs which are to be verified. It contains a module construction which allows abstract types to be made and contains facilities for machine code sections, necessary in an operating system. In order to make verification possible it has restrictions to prevent aliasing of variables. Those cases that the compiler cannot check by simple analysis result in legality assertions being generated which should be proved by a verifier.

2.6.5 Alphard

Alphard [SHA81] was designed to test some theories in program verification. While it was never implemented the work on it considerably influenced many languages, particularly CLU. It was based round the "form", an implementation of an abstract type or abstract procedure. Since verification was the primary aim, a form had pre- and post-conditions and invariants in both the abstract and implementation parts. The abstract properties could be used outside the form in verifying uses of it without needing to know the implementation. They were considered as much a part of the interface to the form as the types of the arguments. Alphard, like CLU, is variable-based in that all values are held in objects, which are potentially updatable. There are no "naked values". Alphard had many novel features which were incorporated in other languages. For example iteration was done by "for" and "first" constructions which were based around specialised forms called "iterators". The "first" loop, which selected the first element from a set satisfying some condition, is one of the few new control constructions to appear in recent languages.

3. INTRODUCTION TO POLY

In this chapter the programming language, Poly, is explained by means of a number of examples.

3.1 Commands and Declarations

There are three types of instructions which can be given to Poly; declarations of identifiers, statements (commands), or expressions. An example of a command and the output it produces is

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

An example of an expression is

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

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

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

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

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

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

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

```
> message;
Hello
```

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

3.2 Procedures

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

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

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

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

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

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

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

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

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

This can be called by typing

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

or by typing

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

3.3 Specifications

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

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

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

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

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

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

will work. Likewise the call

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

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

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

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

3.4 Integer and Boolean

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5 If-Statement

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

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

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

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

which could then be called to test a value.

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

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

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

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

3.6 More on Procedures

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

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

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

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

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

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

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

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

will therefore print out the value 16.

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

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

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

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

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

3.7 Variables

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

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

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

```
> v;
0
```

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

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

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

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

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

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

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

3.8 The While Loop

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

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

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

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

will not print anything.

3.9 Operators

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

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

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

```
> sq 3;
9
```

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

3.10 The Specifications of Types

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

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

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

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

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

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

3.11 Records

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.12 Unions

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

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

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

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

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

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

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

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

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

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

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

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

print the original values. It is possible to write

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

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

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

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

3.13 The Type-Constructor

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

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

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

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

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

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

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

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

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

3.14 A Further Example

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

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

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

3.15 Exceptions

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

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

raises an exception called 'somefault'.

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

has specification

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

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

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

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

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

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

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

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

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

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

3.16 The Specification of Variables

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

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

This has declared 'intpair' to be a pair of integers containing the values 1 and 2. 'intpair$first' and 'intpair$second' can be used as integer values directly.

The specification of an integer variable is

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

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

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

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

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

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

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

3.17 Specifications in Declarations

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

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

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

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

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

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

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

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

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

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

The list of exceptions raised by the procedure must be included in the specification. The exception list in the specification given must include all the exceptions which may be raised, but may include others as well. A special exception name **any** can be used to indicate that a procedure can raise any exception. Any exception list will match a context with exception list 'raises any'.
The specifications of the arguments and result must all match.

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

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

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

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

3.18 Types as Results of Procedures

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

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

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

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

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

3.19 Types as Arguments to Procedures

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

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

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

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

The specification of the result is

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

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

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

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

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

creates a similar type which maps between char and copychar.

3.20 Polymorphic Procedures

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

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

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

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

Similarly

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

However

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

and

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

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

3.21 Implied Arguments

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

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

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

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

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

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

This can now be called by writing

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

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

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

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

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

would be used by writing

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

However, by writing

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

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

3.22 Literals

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

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

```
42 0H3F6A 3p14159
```

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

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

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

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

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

```
PROC(string)integer
```

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

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

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

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

3.23 Early

As far as the meaning of a program is concerned it does not matter whether the conversion procedures are evaluated when the program is compiled or when it is eventually run. However the program is likely to run much more quickly if the literals are converted at compile-time. This is particularly important if a literal is used in a loop or in a procedure which will be evaluated many times. The conversion procedures, or indeed any procedure, can be evaluated at compile-time by adding the word EARLY to its specification.

```
let convertn: proc early (string)integer == integer$convertn;
```

Applying "convertn" to an argument which is a constant (either a literal or a previously evaluated expression) will cause "convertn" to be evaluated at compile-time.

Other procedures can be declared as "early" if they do not use any values which are not known at compile-time. Procedures, such as "print", which write out messages, can be evaluated early, in which case the message will appear when the program is compiled. This can be useful for reporting errors which can be detected at compile-time.

3.24 Lists

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

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

```
#     let car ==
#       proc(ll: list)base
#         begin
#         node$cr(list$proj_nnl(ll))
#         catch proc(string)base (raise nil_list)
#         end;
#
#     let cdr ==
#       proc(ll: list)list
#         begin
#         node$cd(list$proj_nnl(ll))
#         catch proc(string)list (raise nil_list)
#         end;
#
#     let nil == list$inj_nl(void$empty);
#
#     let null == list$is_nl
#     end
#   end;
```

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

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

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

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

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

It is now possible to write simply

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

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

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

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

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

The list created above can now be printed.

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

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

3.25 A More Formal Description

So far in this chapter Poly has been described informally by a series of examples. To complete the description a more formal definition of certain parts of the language, and in particular the checking rules for specifications, will be given.

The language as presented to the user can be divided up into three areas. The "core language" is the minimal system capable of supporting the full language. In particular the core language contains all the specification-checking rules for any constructions that can be written.

It is augmented by a set of "library objects" which are either written in the core language or provided by means of assembly-coded routines. These objects all have specifications in the Poly type system, and any use of them is subject to the standard checking rules. Finally there are some "sugarings" which allow the user to write programs in a more convenient fashion than the core language allows. These are simply rewriting rules and any sugared construction can be rewritten in the core language.

3.25.1 The Poly Core

The core language is an applicative subset of the user-level language. It supports values, procedures and types as first-class objects and ensures that operations on them satisfy the checking rules for their specifications. The core language contains only those parts of the user-level language which cannot be provided either in the library or by sugaring.

The only types in the core language are "void" and "boolean". Void is required as the type of a statement which returns no result, and boolean is required for a conditional. The procedure and type constructors, the record and union constructors are part of the core language. Exceptions also form part of it.

The core language allows polymorphic operations, but type parameters must be passed explicitly. The core language has only one coercion: Where a type is used as a parameter to a procedure, attributes may be "forgotten" from the actual type parameter to make it match the formal. (i.e. a type with more attributes than required may be used as a parameter).

3.25.2 The Library

The library consists of a set of objects, usually types and procedures, which are used in many programs. They may be written in Poly or in assembly language but they all have specifications in the Poly specification system and so any use of them is subject to the normal checking rules. The library contains standard types such as integer and string and procedures to create variables and vectors. Since variables cannot be provided by constructions in the core language the procedure to create them must be written in assembly language.

3.25.3 Syntactic Sugaring

The core language, even with the standard library routines added, is not a particularly convenient language to program in because of the lack of coercions. The user-level language allows certain expressions to be written in a more convenient notation than the core language by essentially providing a set of rewriting rules which allow a user-level language program to be translated into a core language one.

The rewriting rules allow infix and prefix operators, implied type parameters and use of a variable where a value is required. Infix and prefix operators are rewritten in the functional form. Procedures using implied parameters are rewritten with the types as explicit parameters. Variables used where values are required are rewritten as applications of the "content" procedure of the variable (remember that variables are types).

3.26 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 cooperating operations which can together manipulate objects.

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.

3.26.1 Constants

The specification of a constant is 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.

3.26.2 Procedures

Procedures constitute the most complex class of objects. In the core language a procedure takes objects as parameters and returns a result or raises an exception. In the user-level language, which has variables, a procedure may also alter the global state. The specification of a procedure contains the specifications of its arguments and of its result. It also indicates what exceptions, if any, may be raised in the procedure.

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

3.26.3 Types

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.

4. THE DESIGN OF POLY

The previous chapter described Poly as it currently exists. In this chapter we consider the requirements which guided the language design and which underlie all the subsequent design decisions. The way these decisions influenced the choice of type-system and the rest of the language, is then described.

4.1 Defects of Large Languages

The underlying motivation for the design was to find a simpler way of expressing, and in particular type-checking, the kinds of expressions which occur in modern high-level languages. Languages like Ada have a large number of special constructions to perform conceptually similar operations, the aim was to try to generalise them without too much loss of efficiency.

In his Turing Award lecture [HOA81] C. A. R. Hoare singled out three languages PL1, Algol68 and Ada as being too obscure, complex and overambitious and praised their respective predecessors Fortran, Algol60 and Pascal for their simplicity. In all these three cases, but most particularly Ada, the large language is just the smaller language overloaded with special facilities which are useful in special circumstances.

Building facilities into a language makes it large and complicated to learn and to implement. This means that the chances of bugs in the implementation and misunderstandings by users are much higher than in a small language. Just as important is that building these facilities into the language means that the user cannot override the standard definitions with his own if the standard definitions are not appropriate.

4.1.1 Specific Examples

As examples of the way languages tend to build in special facilities and impose restrictions consider the following.

In Algol68 procedures can be treated almost as first-class values. The exception is that the most useful case of returning a procedure "out of scope" is forbidden because it cannot easily be implemented using a stack. Similarly in Pascal procedures may not return structured values because many implementations use a single register to return a result. In Ada procedures cannot even be used as parameters to normal procedures. They can only be used as parameters to generics.

Most languages provide standard definitions of types which cannot be overridden. For instance most provide rectangular arrays but no way of defining triangular or sparse arrays. Shaw, London and Wulf in [SHA81] give several examples of this. It means that the programmer, far from designing a program from the top down, has design decisions made for him by the language designer.

In almost every language particular forms of literal constant are tied to particular types. Any numerical literal is automatically of type integer,and the conversion is done at compile-time. There is no way a user can write an arbitrary-precision integer package and provide a routine to read and convert an arbitrary length numerical literal, and no way to ensure that it is evaluated at compile-time.

In Pascal and Ada the user cannot define new symbols for infix and prefix operators and in Pascal operators cannot be used for any user-defined procedures. The designers of these languages restricted a very convenient notation so that it cannot be used except at the lowest level of programming.

Some languages which allow abstract types will allow constructions such as sparse arrays, which can behave almost like variables. Different operations are performed depending on whether they appear on the left or right-hand side of an assignment. They cannot, however, be treated exactly as variables, and passed as reference parameters to procedures, because most languages assume that variables are the addresses of storage locations.

4.2 Simplifying the Design of Languages

The above examples show how facilities which are built into a language restrict the programmer. Some of the problems illustrated will be discussed in greater detail in this and the following chapter.

4.2.1 The Kernel Language

It should be possible to design a kernel language making use of extension facilities to build up to a user-level language. The kernel must have powerful extension facilities and type system, but be as small as possible. It would then have simple semantics and be easy to transport.

4.2.2 The Use of Libraries

Languages would be simpler if many facilities normally built into the language could be provided by standard library routines. They can then easily be redefined by the user because they are not treated specially by the compiler. To do this requires the language to have suitable extension facilities and in particular a powerful enough type system to be able to describe all the objects in the library. For example if "array" is not built in but provided in the library it must have a type which can be expressed in the type system. A simple kernel language may not have all the primitives required for a particular application area. It will usually be necessary to add extra primitives by means of procedures or objects written in, say, assembly language, (for example input/output or parallel processing primitives) but they must have an external interface and type consistent with the language. They can then be treated exactly as though they had been written in the kernel language, and the type system, by preventing the user any knowledge of the implementation, will ensure that they cannot be distinguished from kernel language routines.

4.2.3 Generalisation

Another, complementary, way of simplifying a language is to generalise constructions so that one general mechanism covers several special cases. Generalisation of a construction may provide additional facilities not covered by any of the special cases. For instance allowing a procedure to be used anywhere that a conventional value can be used is conceptually simpler than having special rules for procedures. It also allows higher-order procedures which are sometimes extremely useful.

The main argument against generalisation is that special cases can be implemented more efficiently than the general case, and that in any case the vast majority of cases can be handled by the special case. For instance most procedures are not polymorphic and do not return procedures as their results.

This is not a satisfactory argument against providing the general case. Polymorphic and higher-order procedures are sufficiently useful that they should not be forbidden on efficiency grounds. The aim should be to ensure that a suitable optimising compiler can recognise when a procedure does not require the completely general implementation and use a more efficient implementation in those cases. Recognising special cases is a well-known operation in optimising compilers for conventional languages and there is no reason to believe that it would be any more difficult in a language like Poly. The efficiency can then be close to that in languages in which only the special cases are allowed (see below for a description of the implementation of procedures in Poly). In any case the increase in availability and use of microprogrammable personal computers means that operations which may be expensive on a conventional computer can be made cheap. [1]

4.2.4 **Requirements for the Type System**

The most important part of the language is therefore the type-system which must allow abstractions to be handled and type-checked. The type-system should not know about the internals of the objects but should be able to decide whether an operation is type-correct only on the basis of information about its external specification. The language must provide a means of constructing abstract types and operations and manipulating them in a way which does not require knowledge of their implementations.

4.3 The Basic Design Rules

It is possible to summarise the conclusions reached in the previous section as rules for a language design. These were the guiding principles behind Poly. The primary rule was that <u>nothing should be built into the language</u> which cannot be built on top of it. In practice this cannot be achieved for various reasons (e.g. "boolean" must be built into the "while" and "if" constructions), but an aim should be that types such as "integer", which are often built into a language, should not be. Of course for efficiency reasons certain types have to be treated specially as far as code-generation is concerned, but this is a separate issue.

Other requirements of the language were that it should be capable of complete <u>compile-time type checking</u>. This is important because a great many faults can be picked up by type-checking which would not be detected until a program was being tested if it were untyped. Run-time checking is not satisfactory because by the time a fault is detected it is too late to do anything about it. This is connected with another requirement, which is that there should be no such thing as a program 'failing'. <u>Any failures</u>

[1] The implementations of Mesa at Xerox PARC use a microprogrammed storage allocator to allocate frames for procedure calls rather than using a stack.

at run-time should be able to be handled within the language as exceptions. This allows the whole system, compiler and operating system, to be written in Poly or at least regarded as Poly procedures or types.

Efficiency of the compiled code is regarded as a secondary goal to be achieved after all the above requirements have been satisfied. This does not mean that it is not important, but that it is a mistake to design a language from the point of view of the implementation.

4.4 The Type System

In the previous sections we arrived at some rules which guided the design of Poly. We now see how they influenced the choice of type-system and how the Poly type-system developed. In particular we see why Poly treats types as first-class values and how this affects the checking of simple values.

4.4.1 Parameterised Types

One of the aims of Poly was to avoid building things into the language. For instance, it should be possible to define "array" in the standard library. Among other things this means that the type-system must be capable of expressing the type of "array".

"Array" is what is sometimes called a "type-constructor", that is something which makes a particular type. It is parameterised by the element type and the bounds of the array. In CLU it would be represented as a parameterised cluster, in other words the set of operations, in particular subscripting, which comprises an array, parameterised by the element type and the range.

A parameterised cluster, and its analogue in Ada the generic type, are both instantiated at compile-time. Hence they behave like macros. They cannot be instantiated at run-time using values computed at run-time, for instance this version of array could not have bounds which were run-time values, because the type-checking scheme would not work. The reasons for this are described below.

4.4.2 Parameterised Types as Procedures

A disadvantage of parameterised types and generics is that they provide an alternative method of parameterisation from procedures. The principle of simplicity would seem to suggest that the same mechanism could be used for both. "Array" could then be regarded as a procedure which took a type (the element type) and two integers (the bounds) as parameters and returned a type as its result.

The problems with this scheme are more to do with type-checking than with the efficiency of parameterising at run-time compared with compile-time. There are two real problems with the type-checking scheme. The first, and simpler, is how to specify the "type" of a type. Since one of the parameters of "array" is a type, and the result is also a type, there must be some way of giving the "type" of these parameters. To avoid further confusion the term **specification**, which corresponds to signature in Russell, will be used to mean the properties checked by the type-checking system, and reserve "type" for the value which is passed as a parameter and returned as a result, at run-time, by a procedure like "array". The question then is how to give the specification of a type?

A second, and more serious problem, concerns the checking of values belonging to types returned by procedures. If any procedure can in principle return a type as its value how is it possible to guarantee, at compile-time, that two values belong to the same type? It is undecidable whether two procedures return the same type so without run-time type-checking there must be some restrictions on the forms of type-returning procedures.

Procedures are the only method of parameterisation in Russell and Poly. The Russell solution to the type-checking problems have been described above. The Poly solution, and the differences from Russell will be discussed below.

4.4.3 Specifying a Type

In order to give its specification we must consider exactly what is meant by a type. A type in Poly is very similar to a package in Ada or a module in Mesa, or indeed to types in other languages which support abstract types to a greater or lesser extent. A package in Ada has a public part which lists the operations provided by the package which are available to the user, and a private part which contains details of the implementation hidden from the user. Two packages with identical public parts but differing in their implementations are to all intents and purposes indistinguishable. The specification of a type is then the "public part" of the type definition i.e. the attributes which are available to the user.

The specification of a type in its most general form is the abstraction which the type implements. This includes assertions about the values or operations e.g. succ(pred(x))=x, succ(x)>x etc. Since such checking cannot in practice be done by a compiler, the specification is reduced to naming and specifying the attributes as procedures with given argument and result types or constants or variables of a given type. The argument here, assumed in Ada, is that suitable choice of names will reflect the expected semantics of the attributes. So that while it may not be possible to check that the procedure "+" corresponds semantically with the addition function the programmer is expected to use the name in that way in any abstract type he might develop.

4.4.4 Specifying Values When Types are First-Class Values

The general principle of type checking in any strongly typed language is that a value of one type is only compatible with a value of the same type. This means that passing a value as a parameter or assigning a value to a variable involves checks that the value has a suitable type for the parameter or the variable. For simplicity the word **context** will be used to describe the requirements of the formal parameter or the variable which a particular value must match if it is to be type-correct. There are often coercions provided which allow a value to be used in a context which requires a different type, but they can be regarded as procedures which take a parameter of the type of the value and return a value which matches the context exactly.

There are various definitions of "matching" used in different languages. Some of these are more formalised than others, most consist of a fairly complex set of rules, nearly all have rules which ensure that the compiler can make the checks. This is simple if types cannot be created at run-time because the compiler can examine the structure of the types and decide whether the values match.

If types can be created at run-time then the matching is in general undecidable. Two possible approaches can be taken. One, used in the Cedar Kernel design, requires in general run-time checking but can make use of static checking in many, possibly most cases. It has the disadvantage that some type-faults may not be detected until run-time when it is often too late to fix them. Whether run-time type-faults result in exceptions or some other form of failure is unclear. The second alternative, used in Russell and Poly, is to make restrictions on the types which will match so that the types can be checked at compile-time. This will result in some types which are in fact the same being regarded as different, but all types which the type-checker considers to be the same will be the same. Rather than being a disadvantage this restricted matching actually can be useful as a way of hiding implementation details. The matching rules can be arranged so that when an abstract type is constructed from a representation the two types, though in practice the same, will not be type-compatible outside the representation. Russell and Poly differ in the matching rules between types and this is the basis for most of the differences between the languages.

4.4.5 Type Matching in Poly and Russell

As an example of the differences between the type matching rules, consider the following function, written in a modified form of Poly.

```
let f == proc(a: boolean) type +,-:...; =: ... end
   begin
   if a then integer
        else real
   end;
```

It returns a type with +, - and other operations, in fact, integer if a is true otherwise real. Suppose it is used in the following pieces of program, all of which would be type correct if dynamic typing were used since in each case the result of the function is the type integer.

```
a)  let t == f(true);
    var i : t;
       begin
       var j : t;
       i := j;

b) var i : f(true);
      begin
      var j : f(true);
      i := j;

c) var i : f(true);
   j : integer;
   i := j;

d) var x : boolean;
   x := <complex expression returning true>;
    begin
    var i : f(x);
    var j : integer;
    i := j;
```

If the compiler is to check that all expressions are type correct then in the most general case it must attempt to evaluate every type-returning function. In case (d) for instance it would have to verify that x was true for every path through the program leading to the assignment, but not necessarily if the assignment were not executed.

Case (c) is a simpler example but it still requires the compiler to be able to verify that passing "true" to the function returns the result "integer".

Case (b) is simpler still. It no longer relies on the function returning type "integer" since it would work equally well if "false" were used in both cases. It could be checked by merely comparing the syntactic form of the expressions, provided it is known that the function always returns the same result if given the same arguments. This requires that no function which could be used in a type-returning expression may use a non-local variable, otherwise its value might change from one call to the next. This kind of type-checking is possible at compile-time but it is at the expense of reporting type errors in cases where more extensive checking would show that the types were in fact correct.

Case (a) is the simplest of all to check. Since the value of the function is bound to an identifier and i and j declared with type t rather than f(true), checking the types reduces to checking that type "t" is used in both cases and has not been redefined in between.

Case (b) illustrates the form of type expression permitted in Russell. In Russell no function may refer to non-local variables except as parameters, even if the function is never used in type-returning expressions. Note that it is not sufficient for functions that directly return types to be variable-free, since in the example above, if f(true) were replaced by f(g(1)), say, then 'g' must also be variable-free. The restriction that all functions must be variable-free is so severe as to make many conventional imperative programs virtually impossible to write. For instance, a program which does input or output must pass the file as a parameter to the "read" or "write" procedure and it must be passed explicitly through every procedure in the calling sequence.

An improvement on this scheme, which would still permit values to have types which were expressions, is to distinguish two kinds of procedure, those which were variable-free and could be used in type-expressions, and those which imported variables and could not be. In practice this could work quite satisfactorily since the functions which appear in type expressions are only a small class of the functions in a program. It seems likely that a future version of Poly would use this system.

The current scheme used by Poly is illustrated by case (a). In Poly a value can only be declared with a type which has been bound to a name. This was the simplest and so, initially at any rate, the best choice. It is restrictive, but not as seriously as might be thought. To explain why requires an explanation of how polymorphism works in Russell and Poly.

4.5 Polymorphism

We have already seen that procedures can be written which take types as parameters and/or return types as results. Treating types as parameters can have another use, allowing polymorphic procedures.

4.5.1 Polymorphism in Poly and Russell

In Russell the actual parameters of a procedure are tested for matching their formal parameters, not in isolation but in the context of all the other procedures. To explain this consider the following example (written in Poly).

```
f: proc(t: type  +,-,... end; val: t)
```

The parameter "val" has a type which is not a global, but another parameter. Using the Russell (and indeed Poly) matching rules this procedure can be applied as

```
f(integer, i)
```

provided that "i" has type "integer". This is because the signature (specification in Poly) of each actual parameter is matched with the signature obtained by renaming any formal parameters with all the other actual parameters. Therefore the type of "i" is compared with "integer" rather than "t". In Russell this renaming applies to all parameters e.g. if there is a procedure with specification (in modified Poly)

```
p: proc(vec: array(1, n, integer); n: integer)
```

then it will match

```
p(v, 10)   if v has type "array(1,10,integer)"
```

This example, however, can also be used to illustrate some of the problems with the Russell type matching scheme. The above procedure will not match

```
a) p(v, ten)  if v has type "array(1, 10, integer)" even if
              "ten" always has the value 10.

b) p(v, 10)  if v has type "vec10" even if "vec10" had been
             set to "array(1, 10, integer)"

c) p(v, 2*5)  unless v has type "array(1, 2*5, integer)"
              and not if it has type "array(1, 5*2, integer)"
```

The reason that none of the above examples will type check is that the checking is done purely by a comparison of the syntactic forms (after any renaming) so that if the expressions differ at all in the written forms then they are not regarded as the same. These examples illustrate the limitations of the Russell scheme and suggest that allowing values to have types which are expressions is probably not a significant advantage over the Poly scheme.

In Poly a similar form of polymorphic procedure can be written. Since all values belong to named types there are no type expressions and so no need to rename every formal parameter with the corresponding actual. Only types can appear as part of the specification of other parameters so it is only necessary to rename parameters which are types. In keeping with the rest of the language, where an identifier cannot be used before it has been declared, type parameters must always precede any use of them in an argument list. This means that a simple left to right scan of the argument list can be used by the compiler.

The solution in Poly to the problem of the procedure taking an array of unspecified size as a parameter would be to look at the problem slightly differently. If the procedure were written to take a type and a value it could be expressed in Poly. Its specification would then be something like

```
p: proc(vectype: type length:...; subscript:... end; vec: vectype)
```

In both Poly and Russell, the result specification of a procedure may refer to the parameters and is obtained by the same process of renaming as is used to check the parameters. e.g. the procedure "f" declared above could be made to return a result

```
f: proc(t: type  +,-,... end; val: t)t
```

If it is called as

```
f(integer, i)
```

the result will be a value of type "integer", since "t" has been renamed as "integer".

4.5.2 Implied Parameters

One of the disadvantages of polymorphism in Russell was the need to write the types out explicitly. Since they could be inferred from the signatures (specifications) of the other arguments they were really superfluous. In Poly there is a mechanism for indicating that parameters which are types can be inferred from the specifications of other arguments. They are known as "implied arguments".

To see how this works consider the following example. Suppose there is a value belonging to a type "t" which has an operation called "print", which prints a representation of it. That procedure will have specification (in Poly)

```
print: proc(val:t)
```

If there are several types with such a procedure we might want to write a polymorphic operation to print a value of any of them. That procedure can be written in Poly as

```
let print == proc (ptype: type print: proc(val: ptype) end;
                   value: ptype)
   begin ptype$print(value) end;
```

"ptype" is declared as being a type with a "print" procedure with the given form, and "value" is a value of that type. The procedure simply applies the "print" operation of "ptype" to "value". (the $ symbol means selection of an operation from a type). This can then be applied as

```
print(integer, 3)  or print(char, 'A') or print(string, "Hello")
```

or applied to any user-defined type with a suitable "print" operation. It would, however, be more convenient if the type did not need to be written in each case, especially because it can perfectly well be inferred from the type of the value. For instance we would like to be able to write

```
print(3) or print('A') or print("Hello")
```

In Poly this shorter notation can be used by writing the type as an implied parameter, which causes the compiler to insert its value automatically. The procedure is written as

```
let print == proc[ptype: type print: proc(val: ptype) end]
                 (value: ptype)
   begin ptype$print(value) end;
```

The only difference from the previous definition is that "ptype" has been written in square brackets. The implied parameters, written in square brackets, are not passed explicitly but are inferred from the types of the explicit parameters. It is now possible to write

```
print(3)  or  print("Hello")
```

This is a purely notational convenience and makes no difference to the semantics of Poly. It provides exactly the same facilities as overloading in languages like Algol68, where the overloading can be resolved from the arguments alone and ignoring the result. It does make the restriction that the type must be uniquely determinable from the actual argument which a) prevents overloading of names, particularly constants and b) makes restrictions on the kinds of coercions that can be employed in the language.

In Poly all the implied parameters must be determinable from the explicit parameters. If they were allowed to be determined from the result type then the kind of overloading found in Ada could be simulated. This would of course require the same kinds of algorithms required for resolving overloading in Ada to find the implied parameters. It would also lead to the possibility of ambiguous expressions.

4.5.3 Polymorphism in ML

The language which became Poly was not designed as an exercise in polymorphism but polymorphism came naturally out of the requirements for the language. It is however useful to consider Poly in the light of another language which provides polymorphic operations based on a different system, ML. Consider the definitions of the identity function in Poly and ML.

```
ML:   let f = \n.n;;
Poly: let f == proc [t: type end] (n: t)t (n)
```

The difference between the Poly and ML versions is that in Poly the polymorphism must be expressed explicitly, by declaring the type to be a parameter, whereas in ML it comes from insufficient specification of the types of the arguments. Any polymorphic ML procedure can be written in Poly simply by declaring suitable type parameters as "type end". In addition Poly allows attributes of a type to be passed, which is not possible in ML because there is no way of associating a procedure with a type.

The notion of "type operators" does however allow ML to express some polymorphic procedures in a more convenient way than Poly. A type operator is similar to a type returning procedure in Poly. For instance "list" is a type operator which can be used to represent the type of all lists. "int list" is a list of integers, "* list" is a list of objects of any type (but all objects in the list have the same type). This is not quite the same as a list constructing function in Poly or even in Russell. In Russell an object can have type "list(integer)" and "list" can be passed as a function, so achieving the effect of "* list". In Poly an object can only have a type which is "listint", since types must be simple names and "* list" can only be represented by specifying a type with properties like hd, tl, nil etc. Unlike a type returning function in Poly or Russell a type operator does not actually take a type as a parameter, indeed there is no concept of a particular instantiation of the type operator for a particular type. Type operators can be written only in terms of records, unions, functions or other type operators. This ensures that a type operator always denotes the same value for particular parameters without requiring an import rule as in Russell. In fact, since a type operator does not depend on its "argument", it need never be instantiated with a particular type. It is does not need the import rule since, unlike a procedure in Russell, it is instantiated only once. A type operator is essentially a polymorphic type.

As an example of the advantages and disadvantages of the different kinds of polymorphism consider the following definitions of "append".

<u>ML</u>

```
letrec append a b =
   if null a then b else cons(hd a, append(tl a, b));;
```

Poly

```
letrec append == proc [base: type end;
   list: type hd: proc(list) base;
              tl: proc(list) list;
              null: proc(list) boolean;
              cons: proc(base; list) list
         end] (a, b: list) list
   begin if null(a) then b
         else cons(hd(a), append(tl(a), b))
   end;
```

Russell

```
let append ==
 func[base: type(); a, b: val list(base)] val list(base)
  { if list(base)$null(a) ==> b
   [] else ==> list(base)$cons(list(base)$hd(a),
                    append(base, list(base)$tl(a), b))
   end };
```

This example shows the merits of the various systems of polymorphism. The ML example is obviously the most concise, but it relies, like the Russell version, on a global definition of a list. This is fine provided that we do not want to use this function on any other list type. If, for instance, we wanted to define a list type using any other implementation (say an array) then we would have to define a new append function. While this is not a problem in many applications there are plenty of cases where a structure can be regarded as a list for some purposes while not being constructed using the normal list type. The Poly example while being more verbose than the other two, gives a more abstract definition since it is independent of the implementation. The Russell example requires the base type of the list to be passed explicitly since Russell does not have implied parameters. It also relies on the signature of the argument being precisely "list(x)" where "x" is any expression. It will not work on any type which is not of this form. It is of course possible to pass the list type as a parameter as well as the base type which will achieve the same effect as the Poly example.

4.6 Further Development of the Type System

The type-system of Poly as it is at the moment is general enough to allow any expression in either Russell or ML to be written in Poly. However it is sometimes necessary to add extra declarations and/or parameters when type operators (in ML) or type-returning functions (in Russell) are used. Also the requirement that the number of attributes in a type be known at compile-time can sometimes be restrictive. These problems are discussed in the this section, and finally there is a discussion of the problem of whether procedures and types themselves belong to types.

4.6.1 Type Inference in Poly

It seems likely that a future version of Poly would incorporate some of the features of ML. It would be possible for the compiler to infer the attributes required of a type and so build up the specifications of type arguments. For instance suppose the above definition of "append" were written as follows.

```
letrec append == proc [list] (a,b:list)list
   begin if null(a) then b
         else cons(hd(a), append(tl(a), b))
   end
```

Each of the polymorphic operations "cons", "hd", "tl" and "null" require that their argument type possess corresponding attributes. This will cause the specification of "list" to be computed with those attributes. It would even be possible to remove types entirely and allow type inference as in ML, but with the difference that the types would be specified with attributes, rather than as now in ML, simply as "*".

While Poly does allow types to be inferred in many cases, there are cases where it will not work, and the type has to be passed explicitly. This only occurs where the type of the result of a function cannot be found directly from the arguments. For example, consider the function "map" which returns a list formed by applying a function to every element of a list. Since the function may have different argument and result types the list types may be different. In ML 'map' has type

```
* list # (* -> **) -> ** list
```

i.e. a function taking a list of one type and a function from that type to another, and returning a list of that other type. In Poly the result type cannot be expressed as, say, "list(integer)", and must be passed explicitly. This is essentially saying that the result type could be any type which possesses "cons" and "nil" and not necessarily the same form of list type as the argument.

4.6.2 Attribute Selection

A formal type parameter is specified by listing the names of the attributes the actual type must have together with their specifications. Any type which has all those attributes can be used as an actual parameter. This is the best way of specifying a type which is being used as a, probably implied, parameter to a polymorphic procedure. The body of the procedure will make use of, presumably, all of the attributes so they must all be present.

An alternative use of a type parameter is as a parameter to a type returning function such as "vector" or "list". Here the attributes have a somewhat different purpose. In both these cases the simple definitions of "vector" and "list" will not require any attributes since vectors or lists can be constructed from any type. If, however, we want to define a list which can be printed by printing the individual elements of a list, there is no way it can be incorporated in the basic list constructor without requiring all types from which lists can be constructed to have a "print" attribute. What we would like would be able to specify that the result of the list type has a "print" attribute with the given specification if and only if the base type had a "print" attribute with suitable specification. This argument can be extended to operations to compare lists or any other such operation. In CLU a parameterised cluster can specify the parameter type using a "where" clause, which is similar to a type specification in Poly. Individual operations in the cluster can have their own "where" clauses and they will only be included in the resulting cluster if the actual type satisfies them. Parameterised clusters in CLU are not procedures so there is no problem of dynamic instantiation. It would be much harder to implement in Poly, and there is also the problem of writing the specification of the "list" function.

There is another, related, form of attribute selection which might be employed. Suppose we wanted to define a vector creating function which took a type as a parameter and returned a vector with the unary and binary operations of the base type written as corresponding unary and binary operations of the vector type. This would require the specifications of the base type and the vector type to be written as expressions which would pose the same kind of problems as with the other form of conditional specification.

4.6.3 The Type of Procedures and Types

Procedures and types in Poly have specifications which describe their parameters and attributes. They do not have "types" in the same way as a value. A values has a specification which contains the name of a type which is a physical object at run-time. The specifications of procedures and types do not have the same kind of physical object associated with them, but it would sometimes be convenient to pretend that they do. This can best be explained by means of an example.

Suppose we write a function which constructs a variable and initialises it with a given value.
e.g. new(10) returns a variable initialised to 10. Its "type" is integer because it contains the integer value 10. We will denote the specification of the variable by "var(integer)" but this must be read as a shorthand (a macro) for a more complicated specification and not as a function call. "new" has specification

```
proc [base: type end] (init: base) var(base)
```

This will work with a value of any type since any type can be coerced to "type end".

A type defined as

```
type a: proc...; b: proc... end
```

has a value which is the procedures a and b. A type specified as type end has no such value because it contains no attributes. It is simply a marker which is used, in the case of the "new" procedure above, to give the specification of the result in terms of the argument.

Suppose now we wanted to create procedure valued variables (i.e. a variable to which a procedure could be assigned). We cannot write simply "new(integer$succ)" because "integer$succ" is a procedure not a value of a type. It is possible to "package up" a procedure so that it is a value of a type but there is an alternative. Suppose we considered "integer$succ" as a value whose type was "proc(integer)integer raises rangeerror" in the same way that "10" has type "integer". While "integer" has attributes "print", "+" etc. "proc(integer)..." has no attributes at all, unless we count the ability to apply it. This is no problem if we use it in a context where it need have no attributes, for instance as a parameter to "new". The result of "new(integer$succ)" would then be a

```
var(proc(integer)integer raises rangeerror)
```

Similarly types could be considered as having types. e.g. integer would have type

```
type succ: proc ... ; print: proc ... end
```

and "new(integer)" would return

```
var(type succ..... end)
```

If types are values belonging to other types, then what is the type of

```
type succ: proc ... ; ... end
```

i.e. the type of the type integer? One could say that, since there is nothing that can be done with it, it has no attributes and so has type "type end". "type end" is of course also a type and belongs to itself. An alternative view of the type of a type is as a set of the operations which map a type into a type with fewer attributes.

Treating procedures and types in this way does have advantages in that they then become truly "first class" values, but it can also lead to complications with coercions.

4.7 Exceptions

The exception system of Poly is fairly conventional, providing similar facilities to exceptions in Ada and ML. The principal difference is that the exceptions raised in a procedure are considered as part of its specification in a similar way to exceptions in CLU.

4.7.1 Exceptions in Poly

Exceptions in Poly are similar to exceptions in ML. That is an exception is a token returned as the result of an operation which indicates that a result could not be returned and/or the post-condition of the operation satisfied either because its pre-condition was not satisfied or the representation had failed. It is thus similar to an exception in Ada or CLU and to an "error" as opposed to a "signal" in Mesa. Unlike a Mesa signal there is no possibility of resuming the procedure which raised the exception. Unlike CLU an exception does not have parameters so a single handler can catch all exceptions.

4.7.2 Exceptions as Part of the Type of a Procedure

Like CLU the exceptions raised by a procedure are considered as part of its type. This means that the specification of a procedure contains a list of all the exceptions which may be raised if it is called. In CLU any procedure may raise "failure" and any exceptions not listed in the result of a procedure are converted to "failure". In Poly if an explicit list of exceptions is given then it is an error (detectable at compile-time) if any others can be raised. This means that a procedure whose exception list is empty, is guaranteed never to raise an exception. The main reason behind treating exceptions as part of the type of a procedure is that it makes it easier to detect when details of the implementation of an abstract object would appear outside it. Failures of the representation of an abstract object should not appear to the user of the object in terms of the implementation but in terms of the abstraction.

To simplify the process the exceptions raised by a procedure do not need to be explicitly listed by the user when a procedure is declared. Instead the compiler will create a list from the exceptions found in the body, removing, of course those trapped by a handler. This is easy in Poly for several reasons. Since the exception set of every procedure is known, any call to a procedure results in its exception set being added to the set. The only problem is with a recursive call. Since there is no mutual recursion in Poly only a simple recursive call need be considered and this can be dealt with by assuming that the procedure does not raise any exceptions when it is called. [2] There is also an exception "<u>any</u>" which is the set of all possible exceptions. It is useful for procedures which take procedures as parameters. If the exceptions which may be raised by the actual procedure parameters are unknown then the formal parameter may be given an exception list of "<u>any</u>". This will allow a procedure to be

----

[2] This can be proved to give the correct answer as follows. Let S be the set of exceptions raised by the procedure call, and A be the exceptions raised by any other procedure calls or explicit "raise" statements (i.e. the set obtained by assuming that S is empty). Then either

$S = S + A$

or

$S = A$

depending on whether there is a recursive call outside the range of a handler (since in Poly a handler catches all the exceptions raised in its block). The least solution in both cases is

$S = A$

and hence the correct exception set will be found by assuming that a recursive call does not raise any exceptions. More complicated cases can arise with recursive procedures nested within recursive procedures but the correct result will be obtained for the outer procedure which is the only one which can be used outside.

used as a parameter which raises any exceptions. Of course a call to the formal procedure will cause the compiler to compute the exception set as "_any_", so unless a handler is used the calling procedure will have an exception list of "_any_". In practice many of the polymorphic operators like ":=" appear to raise "_any_" because they take procedure parameters, and so most procedures without handlers have exception lists of "_any_". A more complicated scheme, in which the exceptions raised by a procedure parameter could be passed as an additional parameter, would allow a better estimate of the exception sets to be made. The exceptions raised by a procedure would then be an expression involving some or all of the exception parameters and possibly other exceptions.

5. APPLYING THE TYPE SYSTEM

The previous chapter described how types in Poly are specified and how values of the types can be type-checked. This chapter is concerned with the implications of the type system for the rest of the language. The parts of the system which are actually built into the language are described first, followed by the standard library definitions. The effects of a polymorphic type system on both of these aspects is covered. There is a description of some convenient syntactic constructions and finally the mechanism of binding Poly modules.

5.1 Type Constructor

In the discussion above we have said that types can be constructed without describing how this can be done. Since a type is a set of named objects the mechanism for constructing a type, the "type constructor" ought to be simply a block consisting of a number of declarations which returns the objects declared as its result. This will suffice for types which are simple collections of objects where a value of the type is not required, but if the type is to have values belonging to it, the situation is complicated.

5.1.1 Type Extension

If a type is to have values then the type constructor must define the structure of a value as well as the set of operations. The structure will always be in terms of an existing type so the type constructor can be considered as "extending" that type by adding new operations. The resultant type has all the operations of the previous type together with the new operations. When a type is extended all references to the original type name in the specifications of the operations are replaced with the new type name, so there is no way that values of the resultant type and the old type can be interchanged. When the previous type is a named type, say, integer, it is sometimes useful to be able to interchange values. The Poly type constructor creates two procedures "up" and "down" which allow values to be converted from the previous to the new type, and vice-versa. As an example, suppose we wanted to write a type "absolute" which could be used in an assembler to represent an absolute, rather than relocatable address.

```
let absolute :
    type
    zero: absolute;
    succ: proc(absolute)absolute;
    pred: proc(absolute)absolute;
    +   : proc(absolute; integer)absolute;
    -   : proc(absolute; absolute)integer
    end
 ==
   type extends integer;
   { zero, succ and pred are inherited from integer, with renaming }
   let + : proc infix (a: absolute; b: integer)absolute
      begin
      a absolute$+ absolute$up(b)
      end;
   let - : proc infix (a, b: absolute)integer
      begin
      absolute$down(a absolute$- b)
      end
   end;
```

"absolute" has no other operations than the ones we have listed in the specification of the result. In particular the "up" and "down" operations cannot be used outside it because they do not appear in the specification.

5.1.2 **Overriding and Recursive Definitions**

The above example also illustrates another point. Here new versions of "+" and "-" have been declared which override the existing declarations. They have been written using the previous declarations of "+" and "-".

Declaring an operation using let rather than letrec allows the old binding of the name to be used in the body.

5.2 Records

A record (or structure) is a mechanism for grouping together a number of (usually) unlike objects. A record can be regarded as a Cartesian product of values, though it is often implemented by treating all the fields as variables.

5.2.1 Records in Poly

Variables are implemented in Poly by means of a constructing function, so the implementation of a record in Poly, should, if possible, separate the treatment of records and variables. Regarding a record as a variable as in Pascal has the advantage that construction of the record, and initialisation, are separated. There is therefore no need for a construction, such as the record aggregate in Ada, to make a record from its constituents. If, on the other hand, records are treated as values, where, once a record value has been created it cannot be changed, there must be a function or construction to make it. Using a function for this purpose has the advantage, in Poly, of using an existing mechanism. Similarly the field selectors can be written as functions from the record to the field type.

A particular record type can then have a specification as a type having a constructor function and some selector functions. Can "record" itself be expressed as, say, a function in the Poly type system taking a description of the fields required and returning the record type? The answer is, unfortunately, no, for two reasons. First it is not possible to give the specification of a type possessing an unspecified number of operations and with unspecified names. Secondly the specification of an arbitrary construction function requires it to have an unspecified number of parameters. The nearest that can be achieved is to use a polymorphic pairing function, as in ML, which takes two types and returns a type with functions "makepair", "first" and "second". Repeated use of this function can produce the equivalent of a record with any number of fields. The inconvenience of having to remember the route to a particular field can be reduced by defining functions with mnemonic names to return a particular field. Similarly a constructor function can be written. However this is still a fairly tedious process for what is after all a fairly common operation.

5.2.2 Records and Images

The record constructor has another use in Poly since it allows fields which are procedures or types. A record with a procedure or type as a field can be used to "package it up" so as to make it a value of the record type. This type can then be used to create variables or vectors of procedures or types. This use of the record constructor is similar to the "image" construction in Russell.

5.2.3 Records and Types

To a certain extent records are unnecessary in Poly because a type can behave very much like a record. The type constructor can construct a value composed of a set of other values and the values can be extracted from the type using its name. With this scheme every record value is a separate type, and there is no record type as such. To create a record type however requires a mechanism similar to the record constructor to "package it up" as a value. For this reason, and also because the type constructor is not a particularly convenient notation for creating record values, a record constructor was included in the language.

There is however one circumstance in which a record value must be represented by a type. Suppose we wished to construct a pair consisting of a type and a value of that type. We might try to write it as a record

```
let pair == record(t: type end; x:t)
```

but this would not work. "t" is a procedure which extracts a value from the record type and so "x" cannot be given "t" as its type. To get round the problem a pair must be written as a type constructed by

```
let intpair ==
  type
  let t == type extends integer end;
  let x == t$up(2)
  end;
```

This has constructed a pair consisting of a type (integer) and a value of that type (2), and the values can be extracted by "intpair$t" and "intpair$x". "t" had to be written using a type constructor so that "2" could be converted from "integer" to type "t" by "t$up(x)". Without it "x" would have had type "integer" and not "t".

5.2.4 Structures

Records are seldom used as objects on their own but more often as part of a structure such as a list or tree. It is possible to construct a list using a recursive union or using variables. Since this kind of structure is often associated with records it may be useful to define a slightly different kind of record constructor in addition to the basic one. This would essentially treat the record value produced as a pointer to a node. The record type produced would have an "nil" value and operations to compare the pointers. For instance, a list could be defined by

```
letrec list == struct(hd: integer; tl: list);
```

which would result in list having a specification

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

This constructor is not completely necessary since its effect can be achieved by other means. Equally, it should not be the standard record constructor since there are many circumstances in which no nil value is needed and hence the selecting expressions do not need to raise exceptions.

This constructor was implemented in a testing version of Poly because it was found to be useful when translating the Poly compiler from Pascal into Poly. Only experience will show whether it is a necessary part of the language or can be dispensed with.

5.3 Unions and Variant Records

Unions and variant records can be considered as examples of types where only a subset of the possible operations are available at any time, that subset being dependent on the history (or state) of the object. For a union of types, injection from any of the component types is possible at any time, but projection into a particular component type can only be performed if the last injection was from that type. A union type contains functions which indicate the permissible projection and so it is possible to consider these "guard functions" as pre-conditions for the projection operations. Variant records are an extension of unions where certain fields are accessible irrespective of the value of the guard functions.

5.3.1 Unions as Finite-State Machines

The generalisation of a union is therefore a type where certain of the operations are permissible only in certain states. It differs from, say a counter where the increment operation is not permitted when the counter is about to overflow, in that all operations which change the state lead to states which are dependent only on the operation and not on the state before the operation. (i.e. the post-condition is not a function of the input state).

A union can be represented as a finite-state machine. Consider the operations on a union of types S,T,U.

```
S:{ProjS => S; InjS => S; InjT => T; InjU => U}
T:{ProjT => T; InjS => S; InjT => T; InjU => U}
U:{ProjU => U; InjS => S; InjT => T; InjU => U}
```

This relatively low-level concept can be useful when dealing with abstract objects. For instance a disc-file can have states Closed, Reading and Writing. The state-transitions on it are similar to the union and can be written as:

```
Closed:{OpenToRead => Reading; OpenToWrite => Writing}
Reading:{Read => Reading; Close => Closed}
Writing:{Write => Writing; Close => Closed}
```

The restrictions are more severe than for a union of types Closed, Reading and Writing, in that there is no way to move directly from Reading to Writing.

5.3.2 Unions as Variants

Essentially the above example is a union where the state changes are part of the abstraction. Another use of a union or variant record is for objects which may be created as one of several types or forms, and once created they retain that type for the rest of their lives. Unlike the "state-changing" examples, most operations on the abstract object will be valid for all states, but the implementation will have to discriminate between the variants. The variants are much more closely associated with the value of the object and cannot be separated out as the state of a finite-state machine.

This view of unions is closer to the variant record where a record has several forms which are distinguished by a "tag field". It is possible to represent a variant record either as a record with a union as one of its fields or as a union of records all of which have at least one field with the same type. Declaring suitable additional functions to extract the variant fields directly allows the variant record to be seen as a single-level object. The choice of representation depends on whether access to the fixed fields or to the variant fields is more important.

5.3.3 Unions in Poly

As with records a union could be constructed by repeated application of a suitable pairwise uniting function, but as with records, it is inconvenient. In Poly there is a union constructor which takes a list of "tags" and specifications.

e.g. union(a: T; b: S)

The tags (a and b) are simply names which are used to construct the names of the operations of the resulting union type. For each of the tags there are three operations whose names are made by appending "inj_", "proj_" and "is_" before the tag. These are respectively injection into the union, projection out and a test for the variant. The injection operation is simple and takes a value of the appropriate specification (Poly allows a union to be constructed out of procedures and type values as well as the more conventional type values) and returns a value of the union type. The projection operation is a reverse of this but is complicated by the fact that the value contained in the argument may not have the same specification as the result of the projection operation. There are two solutions to this. One is to require that the projection operation only be applied after a test has ensured that it is valid. The alternative is to have the projection operation raise an exception or return an undefined value of the appropriate specification. In Poly the projection operations raise an exception if they are applied to an invalid value. There are also explicit tests for the variant (the "is_" operations) and a clever compiler could check that a projection operation was used validly in many cases.

5.4 Control Flow

Control flow abstractions have been fairly well covered in other languages and so Poly generally follows their conventions.

5.4.1 **If**

A conditional is fundamental in any language which uses call-by-value. A language which uses call-by-need can do without it but such a language is not easy to implement on a conventional von Neumann machine. Since Poly was intended to be implemented on conventional machines, it follows that an if-statement or similar is required. The simplest form is the "if conditional then statement" form. This can be improved by allowing the "if/then/else" form and by allowing the if-statement to return a value. Having the if-statement return a value complicates the compilation somewhat because it is necessary to coerce both arms to a common specification. If, for instance, both arms return types then the if-statement must return a type whose specification is the common factor of the two types.

There is the question of whether there should be a closing symbol to an if-statement ("endif" or "fi"). There seems to be no convincing argument either for or against so the decision seems to be a matter of taste. Poly does not use a closing symbol because it requires slightly more writing in the simple case. This decision does lead to the so-called "dangling else" problem where the syntax may be ambiguous. The solution to this is well established (an "else" is always paired with the nearest unpaired "then") and so cannot really be considered a problem.

5.4.2 **While**

The while loop is the simplest form of repetitive command. It is not strictly essential since it can be written as a procedure.

```
letrec while == proc(test: proc()boolean; body: proc())
   begin
   if test() then begin body(); while(test(), body()) end
   end;
```

However the need to create procedures when it is called, for instance

```
while(proc()boolean (x<10), proc() (x:=succ(x)));
```

makes it somewhat tedious to use. It could be argued that this a reason for providing a coercion from an expression to a procedure so that the call could be written as

```
while(x<10, x:=succ(x));
```

however coercions are dangerous because they increase the possibilities for ambiguity and reduce the chances of detecting errors so there must be a strong case to justify them.

Another argument against treating a while-loop as a procedure is efficiency. A reasonably efficient implementation would require that the compiler be able to remove tail recursion. For these reasons a while-statement is built into the language.

Since an expression in Poly can be built out of an arbitrary number of statements the whole body of the while-loop can be placed between the while and do. It then becomes equivalent to a "repeat...until" loop which is therefore not required in Poly.

e.g. `while (....; x<10) do ();`

The same arguments about whether a closing symbol ("enddo" or "od") are needed apply as for the if-statement. The decision in Poly was again to omit one.

5.5 Initialisation and Finalisation

In a purely functional language values are only ever initialised and finalisation is not necessary. In an imperative language initialisation and finalisation of variables may or may not be provided.

5.5.1 Initialisation

Many languages provide means of giving initial values to variables. Very few insist that all variables must be initialised. The usual argument is that if the first value to be assigned is computed at run-time then initialising the variable is just an unnecessary overhead. This is true in languages where variables can only be initialised by values computable at compile-time, but not when they can be initialised by run-time values.

The implementation of variables in Poly ensures that they are initialised by making the creation of a variable a procedure which takes a parameter. Not only does this parameter supply the initial value but its type gives the "type" of the variable.

5.5.2 Finalisation

There have been several suggestions (e.g. [SCH81]) that languages should provide the facility to perform the inverse of initialisation, that is an operation which is performed on an object when it is no longer accessible for any other operation. This is usually referred to as finalisation. These suggestions generally assume that variables are implemented on a stack and so have a definite life-time. Finalisation of objects on a stack can be done simply by executing the code just before the procedure returns. In a language such as Poly, where variables are not allocated on a stack, finalisation is much more of a problem. It would be possible to devise a scheme whereby objects were finalised by the garbage collector when they were found to be inaccessible, however this would not be wholly satisfactory. Finalisation is not essential for the security of the language in the same way as initialisation, so it is not provided in Poly.

5.6 "Standard" Definitions

The above sections have described the facilities for constructing types and manipulating objects in Poly itself. We now consider the types and procedures which, though not actually part of the language, must be provided to make the system usable. They also illustrate some applications of the type system. It is important to realise that by not building them into the language they can always be redefined by the user.

5.7 Variables

Variables have been part of programming languages since the earliest high-level languages, largely because they represent the idea of a storage location in a von Neumann machine. Some languages, the functional or applicative languages, have omitted variables completely, but they still appear in modern programming languages. Undoubtedly a major reason is efficiency, but this cannot account completely for their popularity. The concept of an object which can change the value associated with it can be considered as a reasonable abstraction to want to use and so for that reason it should not be omitted from the language.

5.7.1 References

There are two approaches to the interpretation of expressions involving abstract objects, where the object can be changed by the operations on it. One approach, exemplified by Algol68, is to distinguish references to an object and the value of an object. The reference to an object can be used to obtain the value, but once the value is obtained there is no actual object holding the value. The meaning of assignment in this system is that the object referred to by the value on the left hand side of the assignment has the value on the right hand side placed in it.

5.7.2 Variables and Abstract Types

An alternative approach, found in SIMULA and CLU, is that any value must be contained within an object. New objects may be created dynamically and thrown away when they are no longer required. Assignment may then have two meanings. Either the identifiers on both sides of the assignment refer to the same object, or certain operations take a copy of the object so that the assignment associates a new object which is a copy of the object on the right hand side, with the identifier on the left.

An important distinction between the two approaches is that the object case has no concept of a reference type. Call-by-reference is then taken to mean that the two identifiers refer to the same object while call-by-value involves creating a new object which is a copy of the parameter. Essentially the difference between the approaches is that the former considers identifiers to be bound to constants whether of reference types, in the case of "variables", or to values of some other type, while the latter considers all objects to be variables, but with some having the property that they can be assigned to only once ("constants").

5.7.3 References in Poly

The reference concept has a significant advantage as far as implementation in Poly is concerned. That is that it can be added on to the system without having to be built into the language. A variable constructing function can be provided which returns a reference type. The simplest specification of such a function would be

```
ref: proc (t: type end)
       type (ref)
       initial: proc(t)ref;
       assign: proc(ref; t);
       content: proc(ref)t
       end
```

i.e. the function takes a type as a parameter and returns a type with initialise, assign and content functions. A polymorphic infix operator

":=" could be written using implied type parameters so that expressions like

```
let refint == ref(integer);
let x == refint$initial(3);
x := 2;
```

would work. The initialise procedure is required to create the variable, since in Poly every object must be created by a procedure.

5.7.4 Coercions

A problem with this scheme is that we need to write

```
refint$content(x)
```

rather than simply "x", when we want its value. It would be convenient if the compiler could insert a call to the "content" procedure automatically. Since "ref" is not built into the language this requires either user-definable coercions, or a general rule which would be applicable to any type, but which worked in the required way for variables. User-definable coercions could cause enormous problems unless they were very restricted, and even coercions built into the compiler have to be very carefully framed.

The obvious coercion rule to apply in this case would be

- If a value x of type t is found when a value of any other type is required then try replacing x by t$content(x).

This will deal with most cases, but unfortunately there are problems when a variable is used as the parameter to a polymorphic procedure with implied type parameters. For instance, using the definition of "print" above, we might want to print the current value of a variable. However "print(x)" will not work because "x" is of type "refint" which does not have a "print" procedure. It is possible to construct other examples where this coercion rule will not give the expected results. The only way of solving the problem is to make a variable into a separate class of specification from a simple value. The solution in Russell, though required for a different reason, is to have a special "var" specification.

This is not the solution adopted in Poly because the language already contains a mechanism for solving it.

5.7.5 Variables as Pairs of Procedures

Treating variables as store locations is not always convenient, since an object which behaves like a variable may not always have a store location associated with it. Consider a sparse array in which most of the elements are zero and only the non-zero elements are actually held in the structure. There will be operations to give elements new values and to extract the value of an element, indeed such a structure will behave exactly like a conventional array except that an element of it is not a true variable. This is because it is not possible to subscript the array and obtain a variable which can then be passed to a procedure to modify it. If the procedure uses a call-by-value/result or call-by-name scheme then it is possible for an element of a sparse array to be treated as a variable, but not if it uses call-by-reference.

This suggests that representing an abstract variable by an address is not general enough and that a better representation is as a pair of procedures, one which extracts the current value and one which updates it. Both of them have bound into them the location of the "variable" itself. This is not just syntactic sugaring as in previous proposals along these lines [GES75] but an actual implementation.

The untyped language Gedanken [REY70] had a facility for using pairs of procedures as what were called "implicit references". An implicit reference was a pair of functions, a "setting function" which was executed when a value was assigned to the reference and an "evaluating function" which was executed when the reference was evaluated. The reference itself was created by packaging up the pair of functions with a built-in function IMPREF. The reference (or variable) could then be used exactly like a simple reference but the appropriate setting or evaluating function would be called.

This scheme can be implemented in Poly without requiring built-in functions by using a type as a way of representing a pair of procedures. In addition the abstract variable can be statically type-checked. In Poly a variable is a type consisting of two procedures, "assign" and "content". Variables are constructed by "new" which has the following specification.

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

This definition allows an unambiguous coercion rule:

- If a type "t" is found where a value is required, replace "t" with "t$content()".

This rule does not suffer from the problems of the previous rule and does not require a special "var" specification. The definition ensures that all variables are initialised, avoiding the need to check for uninitialised variables, and the initial value is used to find the base type of the variable. An assignment operator can be declared as

```
let := == proc [base: type end]
            (varbl: type assign: proc(base) end; valu: base)
            begin varbl$assign(valu) end;
```

It is then possible to write

```
let x == new(0);
x := 2;
print(x);
```

The definition of ":=" illustrates a further use of this scheme. In that definition only "assign" is passed in the type. Since the variable will only be updates "content" is not required. "assign" and "content" behave very much like "out" and "in" in Ada, except that they correspond to call-by-reference rather than call-by-value/result. Passing "content" by itself rather than a value of the base type is only required when side-effects may cause the variable to change. It is possible to tell from the specification of a procedure whether it reads, writes or modifies a variable parameter.

5.7.6 An Example of Abstract Variables in Poly

As an example of how this variables as pairs of procedures works in Poly a definition of a sparse array is given. It is not a particularly efficient implementation but it illustrates the principles. A better implementation could be made.

```
let sparse ==
   proc [base:                                        { implied argument }
           type
            = : proc infix (base; base)boolean
           end ]
        (zero: const base)                             { actual argument }
                                                    { returns a procedure }

          proc (integer)
            type
            assign: proc(base);
            content: proc()base
            end
  begin
  let indexpair == pair(integer, base);
  let baselist == list(indexpair);
  let root == new(baselist$nil);                    { the head of the list }
  proc(index: integer)
    type
    assign: proc(base);
    content: proc()base
    end
    begin
                    { lookup returns the value associated with the index }
    letrec lookup ==
      proc(r: baselist)base
      begin
      if null(r) then zero
      else if first(car(r)) = index
      then second(car(r))
      else lookup(cdr(r))
      end;
```

```
                                        { The variable returned is a type }
        type
        let assign ==
          proc(val: base)
          begin
          if ~(lookup(root) = val)
          then root := baselist$cons(
                 indexpair$make(index, val), root)
          end;
        let content ==
          proc()base
          begin
          lookup(root)
          end
        end
      end
    end;
```

The procedure takes a value (zero) which is the most common value. Only elements with a different value are actually stored. The type of this value (base) is inferred from it and is the type of all the elements of the array. The type must have an equality operator so that a test for zero can be made. 'Sparse' returns a procedure which represents the array. Calling this procedure is syntactically the same as indexing an array in other languages. This call returns a variable which can then be assigned to or evaluated. The array is represented by a list of non-zero values with their index values. "pair" and "list" are both procedures which return types. They can both be written in Poly but are omitted for simplicity. Similarly polymorphic procedures on pairs and lists are assumed to have been declared.

The procedure can be used in the following way. A particular sparse array can be constructed by writing

```
let sa == sparse(0);
```

which constructs a sparse array in which all unset elements have the integer value of zero and all the elements are integer. Expressions involving the array can be written.

```
sa(2) := 3;
sa(1) := (sa(2) + 4);
```

5.7.7 Abstract Variables in Russell

Despite having types as values this scheme cannot be used in Russell because the import rule forbids two procedures from sharing a common variable which is not a parameter. A procedure, such as "content" which is parameterless, must always return the same value. The sparse array is given as an example in one of the papers on Russell [DEM79a]. The example is quite complicated, involving use of a special element type, and the result is not precisely the same as the simple array. They admit that "implementing a sparse array type with precisely the same signature as an ordinary array appears to be a difficult problem, and is the subject of current research".

5.7.8 Efficiency

A major argument in favour of references rather than procedure pairs is that of the cost of the implementation. However using inline expansion of "new", "assign" and "content" could make this scheme as efficient as a conventional implementation of references. In practice, in a language like Pascal, variables are often used to hold intermediate result and are only assigned to once. Since Poly allows an arbitrary expression to be bound to an identifier, such cases can be replaced by a simple declaration, allowing the compiler to make better optimisations than in Pascal.

In conclusion it should be noted that any other definition of a variable can be made by the user. It is only the coercion which is actually built in.

5.8 Vectors and Strings

Vectors have been provided in programming languages for many years. The simplest form of a vector is as a linear array of storage indexed by an integer value in the range from either 0 or 1 to some positive value. Their attraction lies in the fact that they can easily be implemented on a conventional computer since the address of a particular element is calculated using simply a shift (or if necessary a multiplication) and an addition. Extensions of this scheme, multidimensional arrays, arrays with a index range starting other than at 0 or 1, can be defined using this primitive.

As an implementation of an abstract object an array corresponds closely to a square or rectangular matrix. It is a less satisfactory implementation of some other abstractions. In some applications it can be regarded simply as a storage-efficient implementation of a list. These kind of applications, the obvious example being a string, require the vector to be expandable as new values are added on.

An array can be regarded as a function which maps from the index value to a corresponding variable. The obvious implementation of an array in Poly is therefore as a function which takes a base type and a size and returns a vector which can hold that number of elements of the base type.

```
vect: proc (size: integer; base: type end)
   proc (subscript: integer)
      type
      content: proc ()base;
      assign: proc (base)
      end raises subscripterror;
```

So "vect(10,char)" returns a procedure which, when subscripted by a value in the range from 1 to 10 returns a variable which can either be assigned to by a character or return a character value. This implementation suffers from the problem that the values of the elements are initially undefined. To avoid this, the actual implementation of a vector provided in Poly is a function which takes a size and an initial value, inferring the base type from the value, and sets all the elements to this value. An alternative, used in CLU, is to regard an array as a sequence which is initially empty, but which can be extended dynamically by adding elements at either end. This avoids the problem of undefined values but at the expense of complication in common cases. It does allow an array to be created as a constant and is particularly suitable for objects like strings which are very close to the concept of a sequence.

5.9 Parallelism

Parallel processing was deliberately ignored in the basic design of the language because it was felt that an adequate basic language could be extended, if necessary, by adding parallel processing at a later stage. Attempting to add the necessary operations within the type-system is an interesting experiment. At present a simple coroutine scheme has been implemented, but full parallelism has not been attempted.

5.9.1 Coroutines

To consider the form the parallel processing library should take we will first consider a scheme which provides a form of parallelism, namely a coroutine. A coroutine is mid-way between a procedure and a process in that it can suspend itself part way through a computation so that when it is next executed it continues from where it left off. However, unlike a full process, it runs until it explicitly suspends itself, so it can operate on data shared between several coroutines with no need to use locks to protect critical operations.

In the coroutine scheme of BCPL [RIM80, RIW80] a coroutine is created from a procedure by calling "createco". The result of this call is a coroutine which can be entered by "callco" and this coroutine runs until it suspends itself with "cowait". "cowait" causes control to return from "callco". A subsequent "callco" on the coroutine appears to the coroutine as though "cowait" has returned. Note that this scheme is asymmetric in that one side, the parent, uses "callco" to run a particular coroutine, while the other side, the daughter, uses "cowait" to return to the parent. Symmetric schemes, like ports in Mesa, are bound to have problems when the coroutines are started because, unlike full processes, they cannot run in parallel in order to synchronise.

5.9.2 Coroutines in Poly

In BCPL the "callco" and "cowait" procedures both have one parameter which is passed when control is transferred and appears as the result of the other procedure. This causes no problems because BCPL is a typeless language. The problem with a similar scheme in Poly would be in ensuring that the specifications of corresponding "cowait" and "callco" calls matched. This essentially requires all "cowait" calls in a coroutine to have the same specification (say proc(s)t) and the corresponding "callco" calls to have the "inverse" specification (proc(t)s). The solution in Poly is to require any procedure which is to be made into a coroutine to have a procedure as a parameter which the "createco" routine sets to "cowait". "createco" then has the specification

```
proc [s, t: type end] (p: proc(init:s; cowait:proc(t)s))
   proc(s)t raises coerror
```

"createco" returns a procedure so avoiding the need for a "callco" routine. For instance if we declare a procedure

```
let writer == proc(init:char; nextch:proc(void)char)
   begin
   print(init);
   while (let ch == nextch(); print(ch); ch = endch)
   do ()
   end;
```

which prints a sequence of characters, the first being a parameter of "writer" and the subsequent being returned by calls to "nextch". This can be made into a coroutine by

```
let cowrite == createco(writer);
```

which declares "cowrite" with specification

```
proc(char) raises coerror
```

We can now call "cowrite" with a sequence of characters. The first time it is called the character is passed as "init" since "writer" is not waiting for a return from "nextch". If "writer" tries to return (i.e. when the character "endch" has been written) then "cowrite" will raise an exception ("coerror") as will any further calls. "cowrite" cannot raise any other exceptions because the specification of "createco" will only take a procedure which does not raise any. It it possible for the user to define new versions of "createco" which, for instance, start the coroutine as well as creating it.

5.9.3 **Full Parallelism**

The coroutine system can be used as the basis of a true parallel processing system. Indeed simply by changing "createco" so that instead of creating a coroutine it started a process and making the calls which transfer control in the coroutine system into a rendezvous system, processes could be provided. The initial parameter to the coroutine could be dispensed with in the process version because the new process can run in parallel with the old one until they are ready to rendezvous. This scheme is similar to the rendezvous in Ada or Hoare's CSP system [HOA78], but it allows one process to have only one entry. Multiple entries to one process would be possible using channels or ports, as in Occam [TAY82], which could be given a specification in Poly. There is however a problem when a process is required to wait for a message on one of a number of channels.

An alternative would be to provide a procedure to start a process (fork) and semaphores. "fork" could have specification

```
proc[s, t: type end] (proc(s)t; s)proc()t
```

where the result of the "fork" is a procedure which when called returns the result, waiting for it if necessary (i.e. "join"). The provision of semaphores rather than some higher level system such as messages or monitors seems rather like providing "goto" rather than "while" or "if/then/else", though in fact a monitor-based system still requires semaphores as well. A monitor mechanism cannot be provided without building it into the language since it requires the compiler to insert calls to set and clear semaphores in a collection of procedures. It is of course possible to write a monitor by explicitly writing calls to set and clear a semaphore, or a procedure which would take a procedure "p" and a semaphore "s" and return a procedure equivalent to the sequence "wait(s);p(...);release(s)". However a procedure which would take a set of procedures and so this to all of them cannot be written.

5.9.4 Other Facilities for Parallelism

Parallel processing is still an important area of research in programming language design and several different approaches are possible. The type-system of Poly should be able to support most of the alternatives although some fit in better than others.

5.10 For and First Loops

Another example of the use of polymorphic operations is in the design of more complex functions for iteration than the "while" loop. The "for" and "first" loops of Alphard can be written as polymorphic functions.

5.10.1 The For Loop

A for-loop specifies that a given statement be executed with some object (the control variable) taking all the values in some, possibly empty, set. The number of times the statement is executed is equal to the cardinality of the set. The order in which the control variables takes the values from the set may or may not be defined, but for any given implementation it must be determined.

5.10.2 The First Loop

Frequently it is necessary to write a loop which searches a list for a particular item and exits either when the item is found or when the list has been searched. The "first" loop was introduced into Alphard to perform this function without the need for "goto" statements or unnecessarily contorted programming.

```
first i:over(vec) until not vec(i).allocated
   then allocate(vec, i)
   else error("All used").
```

This loop searches for the first non allocated item and allocates it. If no item is free it produces an error.

5.10.3 For and First Loops in Poly

For and first loops can both be written as procedures in Poly. The simplest for-loop, corresponding to

```
for i:= a to b
```

in Pascal, i.e. with an increment of one each time, can be written as

```
let for == proc(min, max: integer; body: proc(integer))
   begin
   let x == new(min);
   while x <= max
   do
      begin
      body(x);
      x := succ(x)
      end
   end;
```

This can be generalised to any type possessing ">=", and "succ" operators (Pascal allows a for-loop to iterate over characters and enumerated types).

```
let for == proc [base: type >= : proc(base; base)boolean;
                            succ: proc(base)base end]
                (min, max: base; body: proc(base))
```

The body is the same as previously. (We ignore the problems of exceptions to simplify the explanation).

5.10.4 Alternative Definitions

Because the for- and first- loops are written in Poly rather than built in they can easily be extended when necessary. The Alphard definition can be written in Poly as

```
let for ==
   proc [base: type end]
        (iter: type test: proc()boolean;
                    value: proc()base;
                    next: proc()
               end;
         test: proc(base)boolean; body: proc(base))
   begin
   while (if iter$test() then test(iter$value()) else false)
   do begin body(iter$value()); iter$next() end
   end;
```

A simple first-loop for searching a list would be

```
letrec first == [base: type end; restype: type end;
                list: type hd: proc(list)base;
                           tl: proc(list)list;
                           null: proc(list)boolean end]
               (head: list; test: proc(base)boolean;
                thenpt: proc(base)restype; elsept: proc()restype)
               restype
   begin
   if null(head) then elsept()
   else if test(hd(start)) then thenpt(hd(start))
   else first(tl(start), test, thenpt, elsept)
   end;
```

A version with the recursion replaced with a while-loop could be written fairly easily.

5.11 Syntactic Sugaring

There are several forms of expression which, though not strictly necessary, make writing programs much more convenient. We have already seen two examples of this in Poly. The use of implied type parameters means that the user does not have to write the parameter to a procedure explicitly when it can be inferred from the specifications of values. Another case is automatic coercion of a variable to a value by applying the "content" procedure from it. Two further examples will be described below. Literal constants are a convenient way of writing values in a program, and prefix and infix operators make expressions more readable.

5.12 Literals

A literal is a way of denoting a value by the actual characters rather by an explicit declaration. In most languages the literal constants are restricted to a few built-in types such as integer and string. Poly allows a literal constants for any type by, potentially at least, evaluating them at run-time.

5.12.1 Overloaded Literals

The expression a+1 has a well defined meaning whether 'a' is an integer, a real, a rational or an arbitrary precision number. There is no particular reason to believe that the constant represented by the symbol '1' should have any of these types. It could be argued that '1' is an integer constant in all these cases and that '+' should be defined for (integer, integer) => integer, (real, integer) => real etc. However, consider the problem of a+70000 on a machine where integers are 16-bits, but 'a' is an arbitrary precision integer variable. It is essential that the compiler does not attempt to convert this to an integer, and in general it is up to the user to define the conversion of the character string

"70000" into a suitable representation of a number in his system.

5.12.2 **Literals in Poly**

The solution adopted in Poly is to consider all literals as characters strings and have a procedure to convert the character form into the appropriate type. Hence a+1 is considered to be shorthand for a+convert("1"), and "convert" is a procedure which takes a string and returns an integer, real or any other type depending on its declaration. Since the type of the result depends on the type returned by "convert" rather than on the lexical form of the literal, three different kinds of literal are distinguished.

A numeric literal is any sequence of digits or letters starting with a digit. There are also two kinds of character sequence, those enclosed in single quotes and those enclosed in double quotes. This allows the compiler to insert "convertn", "convertc" and "converts" respectively so that suitable declarations can have the effect of returning integer, character or string values. These declarations can be overridden by explicitly prefixing a literal with a type name (e.g. arbitrary$32) or by redefining the appropriate conversion routine.

5.12.3 **Efficiency and "Early Evaluation"**

A problem with this scheme is that the cost of converting a character string, into, say an integer value, is a considerable overhead if it must be delayed until run-time. For this reason Poly allows a procedure to be declared as "early" which allows the compiler to execute it at compile-time, either by interpreting it or by code-generating it and running the code. Such a procedure can only be executed at compile time if all its parameters are constants (e.g. literals or the results of previous computations) and if it does not refer to any values outside it which are not constants.

This scheme also depends on the procedure terminating correctly. It is of course impossible to guarantee against infinite loops, but other faults, such as use of unassigned variables, which could have unpredictable effects on the compiler, are prevented by the design of the rest of the language.

5.12.4 Other Uses of "Early"

This scheme can also be used to allow other procedures involving constants to be evaluated at compile-time achieving the effect of "constant folding" (e.g. replacing "1+2" by "3") in a type-independent and user-controllable fashion. Finally it allows compiler directives to be written as procedures which are executed at compile-time. They are procedures written as part of the compiler and so can affect its data structures. This may cause problems in a multipass compiler since the "directive" may not be executed until a later pass and so an instruction to read from a file may not take effect at the correct point. It has, however, worked well in the current single-pass implementation of Poly.

5.13 Operators

Operators provide a convenient notation for writing procedure calls with one or two arguments. Monadic prefix operators could be considered as simply removing the need to put the argument in parentheses and so could be treated as a normal procedure call. Binary (infix) operators are more difficult since parsing an expression requires that operators be detected during the parse. Monadic postfix operators are possible but since they are seldom used they are probably not worth the additional trouble in the parser.

5.13.1 The Syntax of Operators

The most general definition of an operator would be to allow any sequence of symbols such as "+", "-", "*" etc. or an identifier. An identifier used as an operator symbol has the disadvantage that the lexical analyser cannot distinguish operators from other identifiers, so a multipass compiler cannot build up a parse tree until it has detected that an identifier is an operator. There are several solutions to this problem. ADA restricts operators to symbolic operators (i.e. "+" or "-" but not in or mod) which are already defined in the language. This is clearly too restrictive for Poly. An alternative, adopted by Algol-68, requires all operators to be lexically distinct from variables, e.g. by writing operators in a different character case. Essentially this is the same as treating operators as reserved words as they are in Pascal.

5.13.2 Operators in Poly

In Poly a procedure can be marked as an infix or prefix operator. This requires the parser to be able to find this information, which is possible in a one pass compiler but could cause problems in a multipass compiler which did not build up the symbol table until after the program had been parsed. In the present implementation of Poly there is only one precedence level for infix operators, so all expressions involving infix operators are evaluated from left to right. This has the unfortunate effect that "x:=a+b" is interpreted as "(x:=a)+b". It seems likely that a future revision of Poly will allow infix operators to be declared with a particular precedence.

5.14 Module Composition

We have already seen that types in Poly are very similar to modules or packages in other languages. There is no need for a separate construction to represent a unit of compilation. Procedures can be used as a way of handling the importing of modules and all module binding can be considered as normal procedure application. Treating modules in this way avoids the need for "definitions files" and means that a module which uses another module does not have to be changed if facilities in the second which it is does not use, are changed.

5.14.1 Module Binding in Mesa and Modula

The normal method of checking interfaces in strongly typed languages with inter-module linkage is by the use of definitions (or interface) modules. The mechanism for Mesa has already been described (chapter 2) and a similar system is used for Modula [WIR82]. A definitions module contains a description of the properties of a module which can be used by a module which imports it. It contains enough information for the compiler to type-check the use of the module. A module which imports an interface can make use of any of the objects defined in it.

5.14.2 The Problems with Interface Files

The major disadvantage is that any change to a definitions file will require all the using modules to be recompiled, even if the change is an addition to the interface or a change to something which is not used by a particular importer. For example, suppose that there is an error reporting module (E) which is used by many of the other modules in the system. For each possible error there is a distinct code and associated message in E. There is also a procedure (E.fault) which takes a code and puts out the associated message. E therefore has an interface which is something like the following.

```
fault: proc(code);
e1, e2, e3, e4....: code
```

Consider what happens if we wish to add another error code to the module. Since the interface has changed, all the modules which use E must be recompiled even though only one module, the one which actually uses the new code and of course E itself, have actually changed. Similarly any change to any of the definitions, for example a change of name, requires all the modules to be recompiled.

An importing module should only have to be recompiled if there has been a change to an interface which requires that the importing module be changed as well. The interface should be split between the importing and exporting modules so that the importing module gives a description of the properties it requires from the exporter and the exporter gives a description of the properties it will export. In the above example each of the modules would contain not just the name of E but the properties (i.e. "fault" plus any error codes) that it actually uses. When the modules are bound together the interfaces can be checked. Providing the exporter provides everything that the importer requires, and that they type-check, the binding will succeed.

5.14.3 **Module Binding in Poly**

This process is exactly the same as applying a procedure to a type in Poly. A procedure which takes a type as a parameter lists only those properties that it will actually use. It can take any type as an actual parameter providing it has the necessary objects in it.

A module in Poly can be represented as a procedure which takes some types as parameters and returns a type as its result. The parameters include only those properties which the module actually uses. The modules (procedures) which comprise a program can be compiled in any order because they are self-contained. It is only when they are applied to actual modules that the interfaces are checked. A module can be changed and the system rebound without having to recompile any of the others. If a module is changed in a way which changes its specification then other modules may have to be recompiled if they depend on it. Those which are unaffected by the change do not need to be recompiled. In particular a change which simply involves adding to the result specification will never require any of the using modules to be recompiled.

The System Modeller (SM) developed by E. E. Schmidt at Xerox also regards module binding as procedure application. Because it is based on Mesa it has to take into account interface files. In SM the interface file is considered as a type and an implementation of it as a value of the type. A module is therefore a polymorphic function which takes a type (an interface) at compile-time and generates a function which can be applied at binding-time to an implementation. This system still means that any changes to an interface require recompilation of the importer.

6. IMPLEMENTATION

The design of a fully optimising compiler for Poly would be a research topic in itself, involving considerable work in machine-independent optimisation. The present Poly compiler supports the full language and the examples in chapter 3 have all been run on it. Despite doing little optimisation it runs acceptably fast for interactive work. Much of the implementation is standard compiler technology and so will not be described in detail. The description in this chapter will concentrate on those aspects of the language which required novel or interesting techniques.

The Poly compiler was written in Pascal following a similar process as I had used when modifying the Zurich Pascal compiler for the IBM 370 [MAT78]. The compiler generates a tree structure representing the code to be generated, which is then code-generated in a second pass. The tree is a low-level representation and most of the work of the compiler, dealing with identifiers, specification checking etc., is done in the first pass. The first version of the implementation had an interpreter directly interpreting the tree. Later a code-generator was written to generate a low-level code (essentially a flattened version of the tree) together with an interpreter. Finally the code-generator was modified to generate VAX [DEC81] machine code.

6.1 Lexical Analyser and Parser

The lexical analyser and parser are fairly conventional. The parser uses recursive descent primarily because I had experience with it. The only problems in parsing occur in parameter lists, when the syntax requires an extra symbol of look-ahead, and with operators, where the parse depends on the declaration of identifiers. In a parameter list the syntax allows either a specification or a list of identifiers followed by a specification.

<identifier>†:<specification> | <specification>

Since a specification can consist of among other things, an identifier, there is potential ambiguity. If an identifier is read it cannot be declared or looked up until the next symbol has been examined. An operator in Poly is simply an identifier which has been declared as a procedure and marked as either an infix or prefix operator. Since the parse of a sequence of identifiers depends on separating the operators from the operands it is necessary to look up the declarations while parsing.

6.2 Identifiers and Specifications

Every identifier in Poly has a specification which describes the uses to which it may be put. The compiler represents a specification by a directed graph made up of pointers and nodes. Each identifier is a node in the graph pointed to by a symbol table if it is local to a block, and/or pointed to by another specification. An identifier is either a constant, a procedure or a type. Constants are defined by their type and so a constant identifier has a pointer to the type to which it belongs (this is always another identifier in Poly). Procedures have pointers to their argument list and the specification of the result (represented by a dummy identifier) and information about whether it is an operator. Types have a list of the identifiers which make up the attributes of the type arranged in alphabetical order.

6.2.1 Internal Representation

The structure making up the specifications is not a tree structure because the attributes of a type have specifications which are often in terms of the type itself. This makes the specification of a type into a looped structure. When a new identifier is bound to an existing type then the whole structure is copied to refer to the new type. For instance if a type "T" has specification

```
T: type p: proc(T)T end
```

then after binding "S" to "T" by writing

```
let S == T;
```

"S" has specification

```
S: type p: proc(S)S end
```

6.2.2 Specification Checking

The most important use of specifications is in checking the validity of operations. The specification checking rules are written as procedures which operate on the structures and return a result indicating either success or the reason for the check failing. Most of the checks are fairly simple, the only ones which could be difficult are those which involve renaming. For instance, to check whether

```
p(t, x)
```

is valid if the specifications of p, t and x are

```
p: proc (s: type q: proc(s)s; s end; s)s;
t: type n: t; q: proc(t)t; r: proc(t) end;
x: t;
```

requires that "s" be renamed with "t" throughout the parameter list. Renaming is done by having a list of pairs of identifiers such that any occurrence of the first identifier is assumed to refer to the second. When "s" and "t" are compared a pair consisting of "(s,t)" is created. The attribute lists of "s" and "t" are scanned ignoring any attributes in "t" but not in "s" and checking the remaining attributes using the renaming list. Note that the list means that any occurrence of the first is replaced by the second and not the other way round. It is possible for a type to appear several times as the second type of a pair, but it can only appear once as the first type. A procedure which takes two types as parameters may take the same actual parameter twice providing the type fits both contexts.

6.2.3 The Specification of Results

This renaming scheme applies both to subsequent arguments (e.g. x in the above example) and to the result. As the argument list of a procedure is matched to the actual arguments a renaming list is built up from the types which have been matched, and subsequent arguments are compared using it. The specification of the result is made by taking a copy of the specification of the result of the procedure and replacing any occurrences of the formal parameters with the actual parameter values found on the renaming list. Implied parameters are dealt with in a similar manner to explicit parameters. For instance using the same definitions of "t" and "x" as before, but making "p" take an implied parameter

p: <u>proc</u> [s: <u>type</u> q: <u>proc</u>(s)s <u>end</u>](s)s

then "p(x)" is a valid call. It is checked by first constructing a renaming list for all the implied parameters with their "second" value set to "not yet found" (i.e. in this example the list would be (s,nyf)). When checking the types of two constants the renaming list is always checked first to see if the "first" type is on the list. If it is set to "nyf" then the entry is replaced with the second type, provided the types themselves match (e.g. while checking "x" against the actual parameter of type "s" "s" is found on the list paired with "nyf". "nyf" is replaced with "t" since "t" has all the attributes of "s" and their specifications match). From then on the renaming rules work as for explicit parameters. If any implied parameters are left unmatched then it is an error (this could in fact be detected when the procedure is declared).

6.2.4 The Type Constructor

One further problem with specifications is in the type constructor. The type constructor takes a type and adds new operations on to it. This first of all requires the type to be copied and renamed with the new type name. Any new declarations must be added on to this type as they are declared, replacing any existing attribute of the same name. The newly declared identifiers also become local values in the block and can be used explicitly as such as well as being available as attributes of the new type.

6.3 The Poly Abstract Machine

The tree structure generated by the first pass of the compiler corresponds to the code of a stack machine, indeed at one time the tree was code-generated into a low-level code for such a machine. This abstract machine is mapped on to the actual machine by the code-generator. Before describing the actual implementation on the VAX the representation of the high-level objects in the abstract machine will be described.

6.3.1 The Abstract Machine

The memory of the machine is a sequence of words each of which is large enough to hold an address to any other location. A word can also hold other information. In addition every word has a tag which identifies the data as either a pointer or not. A pointer always points to the start of an object which contains information such as its length. There is at least one stack which is used to hold link information and local values for procedure calls. The values below the top of the stack are always constant (i.e. once a value has been produced it is never changed) though of course when a stack frame is deleted the old values can be overwritten. The stacks run upwards in memory. The arguments are stored in order on the stack below the link information and the local values are above the link information. Because the values on the stack are always constants all operations which return results simply leave their results on the stack. A declaration is just a result whose offset from the base of the frame is known to the compiler.

6.3.2 Constants

All constants occupy one word in memory so that a polymorphic procedure can operate on an object of any type. This means that objects, such as records, which require more words have to be implemented on the heap and treated as pointers. It would be possible for the compiler to use more than one word for a record and only use a pointer when an object was being passed to a polymorphic routine. This would, however, lead to much greater complication in the compiler.

6.3.3 Procedures

A procedure is represented by a pointer to some code and the non-local values referred to in the procedure. When a procedure is compiled a note is made of any non-local values and these are copied into a vector when the procedure value is constructed at run-time. This ensures that if a procedure is returned as the result of a procedure that the non-locals it uses are still available even though the stack frame that contained them has disappeared. This scheme works because all local values are constants (variables are held on the heap) so that a value can be copied and variables will still work. Values global to all procedures do not need to be copied into the closure, they are in fact treated as constant values and included as literal values in the procedure text. The present implementation uses a single word for a procedure, which points to the vector of non-local references. The first word of this is the address of the code. This requires a double indirection to find the code and so slows down the procedure call. If a pair of words were used, one pointing to the code, and one to the non-locals then the overhead would be reduced. This would, however, cause complications when a procedure was used, say, as a field of a record.

There are certain procedures, those which operate on primitive types like integer, which have to be directly interpreted by the machine. These can be provided by using values of the procedure address which are trapped by the procedure call instruction. Alternatively they can be code-generated into normal procedures.

6.3.4 Types

A type is represented by the address of a vector containing a word for each attribute of the type. As an improvement on this, types containing only one attribute are represented by the attribute itself. It would be possible to do without the indirection and have the type as the vector rather than its address, since the number of entries in the vector can be found from the specification, but it would cause complications when, for instance, a type was returned from a procedure.

6.3.5 Exceptions

Exceptions are handled by two instructions, one which raises an exception whose name is given by a string, and the other which traps all exceptions in the containing block. They can either be implemented by using jumps from the raise instruction to the handler, or by having a fixed location (or register) holding the current handler. The situation is complicated because uncaught exceptions must be propagated back to the calling procedure. This requires either that all procedure calls be followed by a jump to the handler, the jump being ignored if no exception is raised or else procedure frames must be unwound until a handler is found.

6.3.6 Refining Types

In Poly a new type may be constructed from an existing one by selecting a subset of its attributes. This is known as refining the type. The refine instruction causes the attributes to be loaded from fixed locations in the existing type and loaded into a newly created vector. If the ordering of the attributes in both old and new types followed a standard pattern (e.g. they were in lexicographic order) then a bit map could be used to identify which attributes of the old type were to be loaded into successive locations in the new one. Unfortunately, for various reasons, the lexicographic ordering cannot be guaranteed for all types so the refine instruction has to be given a list with the location of each attribute.

A further complication is that a type contained within the type may be refined at the same time. This "nested refine" requires the inner type to be refined first to generate a new type which can then be stored in the new type containing it. If the vector representing an inner type were contained as a continuous sequence of words in the containing type, rather than as a pointer to a separate vector, then this problem would not arise. However there is then the problem that other references to this type would break the rule that a pointer may point only to the start of an object and not into the middle of one.

The need to refine a type presents one other difficulty. Suppose we have a procedure which returns a type as its result. A simple application of the specification matching rules would suggest that it could be passed as parameter to another procedure provided that the specification of the parameter was for a procedure with the same or fewer attributes in its result type. For instance, suppose p and r have specifications

```
p: proc(q: proc () type x:... end)
r: proc() type x,y:... end
```

It should be possible to call "p" with "r" as parameter since, when "q" is called in the body of "p", "r" will return a type with both "x" and "y". Unfortunately this will not work properly because the call to "q" will expect a type with only one attribute and unless this attribute happens to be in the expected place in the vector it will pick out the wrong attribute from the vector. The only solution to this is to construct a function which is passed in place of "r" which calls "r" directly but on the return refines the resulting type so that the expected type is returned. A similar problem occurs with a procedure which takes a type as a parameter. Since these cases occur very infrequently the present compiler will not allow them and insists that a procedure which is being passed as a parameter takes types as parameters or returns types which are exactly the same as the specification of the parameter.

6.4 Garbage Collection

Some form of garbage collection is needed since objects can be created on the heap in an apparently random fashion. A use-counting scheme like that used for the Cedar system could be used, where the garbage collector is needed only for looped structures. It is however implemented using microcoded instructions and would not be satisfactory on a machine without microcode support.

6.4.1 Implementation of a Garbage Collector

Garbage collection requires that all pointers in an object can be found and followed so that all the accessible objects can be marked. The Poly machine is essentially untyped in that a pointer is simply a collection of bits which is interpreted as a pointer by particular operations. Two solutions to finding pointers can be used and both have been tried. One solution is to use maps of objects such that the map shows the location of the pointers and the map for the object pointed to. The other solution is to tag pointers so that pointers can be distinguished from non-pointers in all circumstances. The size of an object must be able to be found either from the tag or from the object itself.

6.4.2 Maps of Objects

Maps have been used in various language systems particularly Algol68 [PEC71] and have the advantage that they can be "added on" to the system and do not require special treatment of pointers in the code-generator. They are particularly appropriate for implementing a language on a conventional computer where all the bits of a word are significant as an address. They do, however, require extra storage to hold the maps. They are simpler to use with Algol68 than with Poly because the type (mode) of a value in Algol68 gives the structure of an object, whereas in Poly a constant is regarded as unstructured so, for instance in a polymorphic operation, the only way to find the map of an object is by indirecting through the vector which represents its type. A map of a constant can be regarded, and even implemented, as an extra, hidden marking attribute of every type.

6.4.3 Tag Bits

The alternative system, used in the current implementation of Poly, is to use tag bits. This is considerably simpler to implement but does lead to some overheads at run-time. It is particularly convenient on machines which can be microprogrammed to support the language or where some bits of a word are ignored when it is used as an address (e.g. the IBM 360 and 370 use only the 3 lower-order bytes of a 4 byte word as the address and ignore the high-order byte).

6.4.4 The Current Implementation

The present implementation uses a simple recursive marking algorithm to mark all the objects which are in use. The marks are held as bits in a separate vector. This is more convenient than using a mark bit in every word. The free store is simply put back onto the free chain and no compaction is used.

6.4.5 Asynchronous Garbage Collection

At present the garbage collector runs whenever the storage allocator receives a request that it cannot satisfy. It would be convenient if it could run asynchronously as a low priority process so that, hopefully, there would always be store available. This is particularly important if Poly were used as part of an operating system where delays in responding to interrupts are unacceptable. The major problem with an asynchronous scheme is that a user process running during the marking phase of the garbage collection may cause an object to be missed if, for instance, the object is assigned into a location that the marker has already passed, and all other references to the object are overwritten. The solutions proposed [STE75, DIJ78] essentially require the assignment operation to be made more complicated than it would otherwise be, so that, for instance, it informs the garbage collector if an unmarked object is assigned into a marked variable.

6.5 The VAX Implementation

The VAX provides a very good environment for implementing the Poly machine. It provides a stack and the standard procedure call instruction corresponds closely with the requirements of the call mechanism in the Poly machine. The stack runs downwards, but this causes no problems as the code-generator can easily change the sign of the offsets from the frame-pointer.

6.5.1 The Garbage Collector

The major problem is in implementing the tag bits. The VAX uses byte addressing and an address occupies a longword (4 bytes). The solution was to treat the top two bits of every word as tag bits and regard all addresses as longword addresses. Every address must therefore be shifted two places to the left before indirection which removes the tag bits and converts the address to a byte address. This is not as much of a problem as it might appear because the VAX provides an index mode in which a value contained in a register is shifted before being added to a displacement. The amount of the shift depends on the size of the operand (1, 2 or 4 byte). Since every operation in Poly operates on a longword value this works quite well. The indexing mode can only be applied to a displacement so the scheme used is to have a register (r11) pointing to the base of the memory space used by Poly, and have all addresses as word offsets from that.

Using two tag bits should allow up to four kinds of value to be distinguished. In practice it is convenient to reserve 00 and 11 for numerical values so that a positive or negative numerical value can be fetched without requiring sign extension. This also means that only one bit of precision is lost. Tag bit values of 01 and 10 can be used to indicate pointers. A value of 01 is used to indicate that the object pointed at occupies only one word, 10 indicates that it occupies more than one word and that the word preceding it contains the length (low order 3 bytes) and a code (high order byte). The code is used to allow the garbage collector to distinguish segments of bytes (i.e. strings or instructions) which might appear to be tagged words but are in fact not.

An alternative to using a length field would be to have an extra bit which could be used to mark the last word of a segment. This has the advantage of not needing a whole word for the length of a segment, at the expense of requiring an extra bit for every word in memory. It is therefore cheaper if most segments have fewer words than the number of bits in a word. It does require this bit to be masked out when a word is loaded from a segment and set correctly when the last word is stored. Apart from the problems of finding an extra bit in a word on the VAX the need for masking and setting the bit mean that a length word is the better method.

6.5.2 Operating System Interface

The VAX implementation relies on the UNIX Pascal system to provide most of its input/output. A few functions, such as block I/O and allocating storage are done by direct calls to the UNIX kernel.

6.5.3 Exceptions

The VAX hardware and procedure call conventions support an exception mechanism which is similar to but not identical to that required for Poly. The VAX convention is that a procedure may have a handler associated which can optionally continue from an exception. The address of the handler is contained in the first word of a procedure activation record. When an exception is raised the nearest handler is found and called. The handler can either continue from the exception or unwind the stack frame and reraise the exception. Exceptions in Poly can never be continued from so this scheme is not entirely satisfactory. It also does not allow a name to be passed to the handler. The present implementation uses the address at the base of the frame as the address of the currently active handler for that procedure. The VAX procedure call instruction sets this word to zero on entry and this is taken, in the Poly system, to mean that no handler has been set up. On entry to a block which contains a handler the address of the handler is stored in this location, after saving the current handler address if necessary. When an exception is raised the frames are unwound until a handler is found. On entering a handler or exiting from a block the previous handler is reinstated.

6.6 The Poly Environment

The Poly compiler is intended to be one function in a general Poly programming environment. Poly allows complex objects to be given specifications which can be checked when they are bound together. It is possible to achieve the same kind of protection afforded by, say a capability architecture, by checking the interfaces when objects are bound together without the overhead of repeating the checks on every reference. These kinds of checks are also more flexible than hardware checks which are often limited to read/write/execute permissions. It does of course require all programs to be written in Poly, or if not to be given specifications in a similar way to Poly routines and to be very carefully checked.

6.6.1 The Current Poly System

The present system is arranged around a piece of store which represents the memory of the Poly machine. All Poly objects, programs as well as data, are contained in this store. New objects can be created by Poly programs or by the code-generator. There is one vector which contains all the global data and their specifications. The memory can be written out to disc at the end of a session and read in at the start of the next so that declarations can be preserved from one session to the next.

The memory is initialised by the compiler with certain standard types boolean, integer etc. which are treated specially by the code-generator, and other declarations are read from an initialisation file. From then on the compiler will accept input from the terminal or from a file, compile it and execute it. All objects in the Poly memory and all operations on them must therefore have been checked by the compiler. When a global object is declared a structure representing its specification is created as well. The present compiler, written in Pascal, keeps its own information about the specification of global values, but the representation of the specification in the Poly memory can be used by Poly programs. In particular there is an operator which can look up a declaration and print its specification. This representation is also used by the compiler to build up its own information if the memory is read in from disc.

6.6.2 **Future Development**

At the moment some parts of the compiler, the lexical analyser and the code-generator, have been translated from Pascal into Poly. It is intended that the whole compiler should be a Poly procedure with a Poly specification. Eventually a full programming environment will be developed with every object having a Poly specification.

6.7 Further Implementation Work

The present implementation of Poly does not perform much optimisation. It is intended, once the compiler has been translated into Poly, to do some simple high-level optimisation. The most important is probably to allow in-line insertion of the text of procedures so as to avoid the overheads of procedure calls. This is particularly important in procedures which are used to achieve the effect of overloading. For instance the polymorphic "print" procedure, which merely calls the "print" operation of the type which is passed as its parameter, could be inserted in-line. It would generate no more code than calling the polymorphic "print", but would save an extra call every time it was used. Since there are many polymorphic operations like "print" this would be a big saving. There are several similar kinds of optimisations which have been described in this chapter and which may be implemented if time permits.

It would be interesting to implement Poly on a machine which could be microprogrammed to give specific support. Storage management in particular would benefit from microcoding. An asynchronous garbage collector would be a practical possibility.

7. SUMMARY AND FUTURE RESEARCH

This dissertation describes the design and implementation of a programming language, Poly. The design was the result of a decision to pare down the requirements of the language to essentials and not include anything unless there was a clear justification. Polymorphism was a result of this decision, not a "fancy feature" added on because it might be useful.

Many languages include generic types or procedures, and overloading of operators. Treating types as values means that all these can be considered simply as special cases of procedure calling. This means there is only one mechanism for parameterisation. It also allows the user of the language more control, since there are fewer rules built into the compiler and more of them are incorporated in the definitions of procedures which can easily be overridden or changed.

Making the decision to treat types as values can lead to problems with static type checking. The only other language to take this view, Russell, required that all procedures be "variable-free" (i.e. import no global variables) and treated variables as very much part of the language. This made restrictions on parts of the language which were not concerned with types and are avoided in Poly. Poly makes a different restriction which affects only the use of types. Two values are considered to have the same type only if they both belong to the same named type. Some form of restriction is necessary because, if types are values which may not be computed until run-time, it is in general undecidable whether two expressions represent the same value. There must be some restriction on the forms of the expressions so that this is decidable at compile-time, and hence that the language can be type-checked. It is important to note that we are concerned only with whether the two expressions denote the same value and not with what that value is. A third alternative, which may be explored in a future version of Poly, is to separate variable-free and variable-importing procedures into separate classes and only allow the former in type expressions.

Another area which could be explored by a future version of Poly is type inference. The language ML is statically typed even though the user gives no type information in the program. The compiler finds the most general type for each declaration by examining the use of all the identifiers in an expression. This is very convenient, particularly for interactive working. Types in ML are not values, though they can be considered as "empty" types in Poly (i.e. those with no attributes). It seems likely that the same mechanism of inferring types could be used for a version of Poly, where types have attributes associated with them. It would require the unification algorithm used to assign types to expressions in ML to accumulate the attributes of unified types. In addition, it would allow implied type parameters to depend on the result of a procedure as well as on other explicit parameters. (In the present version of Poly type parameters may be inferred from other parameters, but not from the result). This would allow the effect of overloaded literals as well as overloaded procedures (as at present). A language combining the Poly view of types as modules and the simplicity of type-inference from ML would seem to have considerable advantages over many languages.

While efficiency of the resulting code is not of paramount concern in most applications, it is important if a language is to gain serious acceptance. [1] The decision to treat types as values and allow higher-order functions was taken with some reluctance because it would require a garbage collector to reclaim the free store. However the advantages for the language design were considerable. The problem then was to decide how best to compile the language to take advantage of the fact that most procedures do obey a simple stack discipline, so that providing

---

[1] It is interesting that the advantages of Lisp outweigh, for a large section of the research community, the disadvantages of its poor speed performance (on a conventional machine) compared with statically typed languages.

generalised type values and higher-order functions does not severely compromise performance. Another potential inefficiency is that all operations, even those as basic as integer addition, are treated as procedures. To gain a reasonable level of efficiency without losing generality the compiler can use in-line insertion and a simple peep-hole optimiser. These examples illustrate a general point, that thinking about the implementation of a language while designing it is fine, but the compiler-writer's job is to implement the language, not to design it.

Poly was designed as a general purpose programming language. Only with experience of its use will it be possible to say whether it has succeeded. In any case it has shown that treating types as values is practical and a serious alternative to generics and overloading.

8. REFERENCES

[BIR73]
Birtwistle G.M. et al. "Simula Begin"
Auerbach Publ., Philadelphia, 1973.

[DAH66]
Dahl O.J. and Nygaard K. "Simula-An Algol-based simulation language"
Comm ACM 9,9 Sept 1966, 671-678.

[DEC81]
"VAX Architecture Handbook"
Digital Equipment Corporation, 1981.

[DEM79a]
Demers A. and Donahue J. "Report on the Programming Language Russell"
TR79-131, Department of Computer Science, Cornell University, 1979.

[DEM79b]
Demers A. and Donahue J. "Revised Report on Russell"
TR79-389, Department of Computer Science, Cornell University, 1979.

[DEM80a]
Demers A. and Donahue J. "An Informal Description of Russell"
TR80-430, Department of Computer Science, Cornell University, 1980.

[DEM80b]
Demers A. and Donahue J. "The Russell Semantics: An Exercise in Abstract Data Types"
TR80-431, Department of Computer Science, Cornell University, 1980.

[DIJ78]
Dijkstra E.W. et al. "On-the-Fly Garbage Collection: An Exercise in Cooperation"
Comm ACM 21,11, Nov 1978, 966-975.

[DOD77]
U.S. Department of Defense. "Revised 'Ironman' "
Sigplan Notices 12,12 Dec 1977.

[DOD80]
U.S. Department of Defense. "Ada Reference Manual"
Forsvarets Forskninganstalt, Stockholm 1980

[GES75]
Geschke C.M. and Mitchell J.G. "On the Problem of Uniform References to Data Structures"
Xerox Palo Alto Research Center, 1975.

[GOR78]
Gordon M. et al. "A Metalanguage for Interactive Proof in LCF"
Fifth Annual Symposium on Principles of Programming Languages, Tucson, 1978.

[HOA78]
Hoare C.A.R. "Communicating Sequential Processes"
Comm ACM 21,8, Aug 1978, 666-677.

[HOA81]
Hoare C.A.R. "The Emperor's Old Clothes". 1980 Turing Award lecture.
Comm ACM 24,2 Feb 1981, 75-83.

[ICH79]
Ichbiah J.D. et al. "Rationale for the Design of the ADA Programming Language"
Sigplan Notices 14,6 Part B June 1979.

[JEN75]
Jensen K. and Wirth N. "Pascal - User Manual and Report"
Springer-Verlag, New York, 1975.

[LAM77]
Lampson B.W. et al. "Report on the Programming Language Euclid"
Sigplan Notices, Feb 1977.

[LAM82]
Lampson B. Privatė Communication. Nov 1982.

[LIS81]
Liskov B. et al. "CLU Reference Manual"
Springer-Verlag, Berlin 1981.

[MAT78]
Matthews D.C.J. "A Pascal Implementation for the IBM370"
Diploma dissertation. University of Cambridge. July 1978.

[MIL78]
Milner R. "A theory of type polymorphism in programming"
JCSS 17,3 348-375 1978.

[MIT79]
Mitchell J.G. et al. "Mesa Language Manual"
Xerox Palo Alto Research Center, 1979.

[PEC71]
Peck J.E.L. (ed.) "IFIP Working Conference on Algol68 implementation"
North-Holland Publishing Co. Amsterdam 1971.

[REY70]
Reynolds J.C. "Gedanken - A Simple Typeless Language Based on the
Principle of Completeness and the Reference Concept"
Comm ACM 13,5 May 1970, 308-319.

[RIM80]
Richards M. and Moody J.K.M. "A Coroutine Mechanism for BCPL"
Software-Practice and Experience, Oct 1980.

[RIW80]
Richards M & C Whitby-Strevens. "BCPL: The Language and its Compiler"
Cambridge University Press, 1979.

[SCH81]

Schwartz R.L. and Melliar-Smith P.M. "The Finalization Operation for Abstract Types"
Technical Report CSL-122, SRI International, Menlo Park, California, 1981.

[SCH82]

Schmidt E.E. "Controlling Large Software Development in a Distributed Environment"
Doctoral thesis, University of California, Berkeley, 1982.

[SHA81]

Shaw M. (ed.) "Alphard: Form and Content"
Springer-Verlag, New York 1981.

[STE75]

Steele Jr. G.L. "Multiprocessing Compactifying Garbage Collection"
Comm ACM 18,9, Sept 1975, 495-508.

[TAY82]

Taylor R. and Wilson P. "Occam"
Electronics 55,24, 30 Nov 1982, 89-95.

[WIJ76]

van Wijngaarden A. et al. "Revised Report on the Algorithmic Language Algol68"
Springer-Verlag, Berlin, 1976.

[WIR82]

Wirth N. "Programming in Modula 2"
Springer-Verlag, Berlin, 1982.