On the Classification of Binding Mechanisms

R.Morrison, M.P.Atkinson[+], A.L.Brown & A.Dearle

Department of Computational Science, University of St Andrews, North Haugh,

St Andrews, Scotland KY16 9SS

[+] Department of Computer Science, University of Glasgow,

Lillybank Gardens, Glasgow, Scotland G12 8QQ

Keywords binding, persistence, operating systems, programming languages, databases

1. Introduction

In traditional programming languages, database management systems, file systems and operating systems there are a number of, often conflicting, binding mechanisms for composing sub-systems, programs and data. In our experiments in designing, building and using a persistent information space architecture (PISA) [3] we have encountered these binding mechanisms and wish to report on them here.

We wish to build a total system capable of providing for all programming activity. Our traditional view of the persistent information space is that it will subsume the functions of a plethora of mechanisms currently supported by components such as command languages, editors, file systems, compilers and interpreters, linkage editors and binders, debuggers, DBMS sublanguages and graphics libraries[1]. The information space is composed of objects, which may be simple or highly structured, defined by the universe of discourse of the type system

of the PISA architecture. To build sub-systems or other objects out of the information space requires mechanisms to compose these components. Since we wish to subsume the activities of the programming language, database management system, file system and operating system, the equivalent power of the binding mechanisms in these systems must be provided.

A further requirement on our information space is that the evolution of the data should be controllable. Since the uses of data cannot be predicted it is necessary to support the construction of new software systems which make use of existing data even when the data was defined independently. For large scale, widely used or continuously used systems any alteration to the system should not necessarily require total rebuilding. A mechanism is required to control this rebuilding.

2. The nature of binding mechanisms

Traditionally a binding consists of a name-value pair [13]. That is, a value is bound to a name for some period during the evaluation of a program. This has been extended by Burstall & Lampson to include a type [4] and further by Atkinson & Morrison to constancy [2]. A binding mechanism, therefore, has four components: a name, a value, a type and an indication as to whether the value is mutable or not. To further complicate the issue, bindings may be constructed statically by the compiler or dynamically by the run time system. Indeed bindings can be made at intermediate stages but only the extremes are of interest here since non-static bindings are always dynamic in some sense. A binding mechanism has four components, listed below.

1. R-value or L-value bindings?
2. When is the binding performed?
3. What scoping is involved?
4. When is type checking performed?

2.1 R-value or L-value bindings?

Bindings may be made to immutable values, that is constant values that do not alter during the period of the binding, or to mutable values (locations) where the binding does not change although the value refered to may. These kinds of bindings are traditionally known as R-value (for immutables) and L-value (for mutables) bindings in programming language semantics parlance [13]. The manifest constants of BCPL[11] or Pascal [15] are examples of R-value bindings. On the other hand Pascal variables are examples of L-value bindings where the compiler may bind the name to a location but not to a particular value since that may vary at run time.

2.2 When is the binding performed?

Some bindings can be performed statically (usually by a compiler) and others require to be performed dynamically (by the run time system). Again the manifest constants of BCPL or Pascal are examples of static R-value bindings. In contrast to this the variables of Fortran can be statically bound by the compiler and so constitute static L-value bindings. That is, the compiler can statically allocate the location for the variable, thus setting up the binding between name and L-value.

In block structured languages the binding of name to value is established when declaration is encountered. That is, although an abstract stack address may be statically determined, the actual location, or value is not known, and the binding not performed, until the block is entered and the code for the binding executed. Thus, the variables of Pascal are examples of dynamic L-value bindings. Procedure parameters are also bound dynamically since a procedure activation is equivalent to entering a block with the procedure parameters as the initial declarations. Constants whose values are calculated dynamically, that is when the constant is created, which can be seen in S-algol[9] and the simple constant values of Ada[6] are examples of dynamic R-value bindings. The advantage of this type of constant is that it may be read in or calculated from other data objects and then protected from change.

Part of the difficulty in classifying these bindings is the many mechanisms used to construct and control them. For example, in the above a compiler normally establishes an L-value binding, allocating the object an address in the logical address space of the program. This address may be altered by a linkage editor at load time and may be further translated by dynamic relocation mechanisms such as paging and segmentation. For the present, we will treat the later translations as variants of dynamic systems and concentrate what can be performed by static analysis and what cannot. For example, a Fortran or Cobol compiler running on a bare machine could calculate the physical addresses of all its objects. On the other hand, because of block structure and recursion, a Pascal compiler could not. This is irrespective of mechanisms such as linkage editors and paging.

2.3 What scoping is involved?

A binding is always performed with reference to a particular environment. That is, a binding will be part of a particular environment and may use other bindings in that environment to establish its own. The scope of the binding determines where it may be used. There are two common forms of scoping in programming systems, static and dynamic. In static scoping, the scope of the bound name can be detected by static analysis of the programs. The algol 60 scoping rule is static and this allows duplicate bindings to be detected statically. In dynamic scoping, the binding in scope is the one with the correct identifier that was last defined in the dynamic evaluation of the program. Dynamic scoping can be seen in Lisp[7] and in the segment binding mechanism of Multics [5].

In persistent systems some form of dynamic scoping is usual. A binding of a file name to a file in an open statement is usually resolved in the dynamic environment of the program. This is even although the file name may become statically scoped. Indeed this is the desired form of binding since different invocations of the program may be required to bind to different files. For example,

when running the mail command in UNIX [12] user running the command wishes to bind to their own dynamic environment.

2.4 When is type checking performed?

Type checking, assuming it is performed at all, can be performed statically by a compiler or by the run time system. Dynamic type checking occurs when the run time system executes code to ensure that the data is of the correct type. This typically occurs even in so called statically checked languages with read statements and in projections out of a union. Some languages such as SASL[14] deliberately choose run time type checking to facilitate polymorphism.

The binding mechanisms described above can be summarised in the following table.

	Static Typing Static Scoping	Static Typing Dynamic Scoping	Dynamic Typing Static Scoping	Dynamic Typing Dynamic Scoping
Static R-value	1	2	3	4
Static L-value	5	6	7	8
Dynamic R-value	9	10	11	12
Dynamic L-value	13	14	15	16

There are 16 different methods of binding based on the four binding choices given above. The most static form is a static R-value binding with static type checking and static scoping. The most dynamic form is a dynamic L-value with dynamic type checking and scoping. It is interesting to note that even within one particular language there is often more than one binding category. For example, in Pascal category 1 describes **const** values, category 13, variables, category 15, variant projections and category 16, to bind file names (although once the binding is established it now belongs to category 13).

We use the phrase Flexible Incremental Binding Set (FIBS) to describe the mixture of bindings required in a particular system. We search for a FIBS that is sufficient for a persistent environment.

3. Further Comments on the Binding Categories

If we examine the columns of the above matrix we can see that the first column, categories 1, 5, 9 and 13 represent the bindings normally found in strongly typed languages. Pascal uses categories 1 and 13 for its manifest constants and variables, Fortran category 5 for its variables and PS-algol [10] categories 9 and 13 for its dynamic constants and variables respectively.

In column 2, categories 2, 6, 10 and 14, it is difficult to find examples of such bindings. For this column every unique name in the system must have the same type otherwise the type could not be checked statically. Exception mechanisms, such as Ada's, sometimes have this kind of binding. That is, although the exception name has a static scope, the particular handler that is invoked to service the exception is determined by the dynamic evaluation of the program. Thus the binding of the exception name to the handler is dynamically scoped.

Column 3, categories 3, 7, 11 and 15 allow the typing to be dynamic and the scoping to be static. SASL uses category 11 to bind its objects where the scoping is static but the type checking is not performed until the operator is applied to the operand. By delaying the type checking, a form of polymorphism is achieved. For example, the self apply function $\lambda x.xx$, can be written in SASL

$$\text{self } x = x\ x$$

Such functions are more difficult if not impossible to write in strongly typed languages such as ML [8]. Category 15 is typically the binding mechanism used in

read statements, in projections out of a union and to bind file names to external files. For example, the statement

let a = readi ()

forms a binding between the name 'a' and the value read in. We use the procedure to indicate that an integer is expected so that the type checking in the compiler can continue. However, the type checking is actually performed as the object is read in and can therefore be regarded as part of the binding. By this technique the type checking can be dynamic but the static scoping may be preserved. Similarly, projection from a union is normally performed during dynamic evaluation of the program. The type check involves determining which particular member of the union is involved in the projection.

Finally column 4, categories 4, 8, 12 and 16 are the most dynamic of all. As said before category 16 is the mechanism used to bind Multics segments, the objects of Lisp and in establishing file bindings. An applicative Lisp would use category 12. These bindings may be summarised in the following table.

	Static Typing Static Scoping	Static Typing Dynamic Scoping	Dynamic Typing Static Scoping	Dynamic Typing Dynamic Scoping
Static R-value	Manifest constants of BCPL, Pascal Ada generic			
Static L-value	Manifest variables Fortran			
Dynamic R-value	Dynamic constants Ada, PS-algol	exceptions Ada	SASL Ada sub-types	Applicative Lisp
Dynamic L-value	Block structure and parameters Pascal, Algols		projection bindings using the dynamic environment	Lisp Multics segment

4. Safety v Flexibility

In determining the appropriate binding mechanisms for a particular system, the designer is is faced with the problem of balancing safety against flexibility. The safety in the system is derived from being able to say (even prove) something about the program before it runs (ie statically) in order to improve confidence that it is correct. This explains the wish by most language designers to employ static type checking as one of the devices for static checking.

A second aspect of static checking is that the programs so checked are usually more efficient. By performing the checking statically the need for dynamic checking is removed making the run time representation of the program execute faster and in less space.

Finally an aspect of static checking that is often overlooked in programming systems is that of the use of source code as program documentation. If a compiler can

statically check a program then so can another user. Thus statically checked programs have better documentation properties and consequently better cost properties throughout the life cycle of the programs.

Taken to extreme statically checked systems such as those described in category 1 are not very interesting. Statically typed and scoped, static R-values cannot accommodate change in the system. New values cannot be calculated and this category of binding is only of interest as a subset of a more general FIBS. Even the static (applicative) languages have more binding mechanisms than this. It is, however, a very safe mechanism.

At the other extreme, totally dynamic systems are just as unacceptable. Category 16 defines dynamic L-values that are dynamically type checked and scoped. Reasoning about the bindings in such a system is difficult since the particular bindings that occur depend upon the dynamic evaluation of the program. The system is extremely flexible since the program can calculate which binding will occur next but it is much less safe.

5. Conclusions

We have presented a classification of binding mechanisms that have been traditional in programming languages, operating systems, database management systems and file systems. In this we have extended the notion of a binding to be a name, value, type and constancy quadruple. From this categorisation we have shown that the binding set for any language may be described leading to a greater understanding to the facilities of the particular language.

6. Acknowledgements

This work is supported by S.E.R.C. grants GR/D 4326.6 and GR/D 4325.9 and a grant from International Computers Ltd.

7. References

1. Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. "An approach t

2. Atkinson, M.P. & Morrison, R. "Types, bindings and parameters". Proc of the Appin worksho

3. Atkinson, M.P., Morrison, R. & Pratten, G. "Designing a persistent information space architecture". 10th IFIP World Congress, Dublin (September 1986),115-120.

North-Holland, Amsterdam.

4. Burstall, R. & Lampson, B. "A kernal language for abstract data types and modules". Proc

5. Dennis, J.R. "Segmentation and the design of multiprogramming computer systems". JACM

6. Ichbiah et al. **The Programming Language Ada Reference Manual**. ANSI/MIL-STD-1815A-1983. (1983).

7. McCarthy, J. et al. **Lisp 1.5 Programmers manual**. M.I.T. Press Cambridge Mass. (1962).

8. Milner, R. "A proposal for standard Ml". Technical Report CSR-157-83. University of Edinburgh. (1983).

9. Morrison, R. "S-algol language reference manual". University of St Andrews CS/79/1 (1979)

10. PS-algol Reference Manual. Universities of Glasgow and St Andrews, PPRR-12-87.

11. Richards, M. "BCPL a tool for compiler writing and systems programming". AFIPS SJCC 3

12. Ritchie, D.M. & Thompson, K. "The UNIX timesharing system". Comm.ACM 17, 7 (1974), 365-375.

13. Strachey, C. "Fundamental concepts in programming languages". Oxford University Pres

14. Turner, D.A. "SASL language manual". University of St.Andrews CS/79/3 (1979).

15. Wirth, N. "The programming language Pascal". Acta Informatica 1,1 (1971),35-63.