Bindings in Persistent Programming Languages

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

Abstract

In designing and building persistent object systems we are attempting to regularise the activities on data that are traditional in programming languages, operating systems, database management systems and file systems. We hypothesise that regularity and simplicity may be achieved by regarding the exercise as one of designing a language powerful enough to allow for all our programming needs and using some principles in the design of the language to achieve this regularity and simplicity.

In this paper we investigate the nature of binding mechanisms showing how some form of dynamic binding is necessary for persistence. The binding mechanisms of Ada, which has a traditional file based view of persistence, and of Napier, which has an object based view, are used as illustrations.

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 the units of reconstruction.

We report here, first on the nature of binding mechanisms, giving a classification of them, demonstrating that some method of dynamic binding is necessary in a persistent system and then describing the mixture of bindings allowed in the languages, Ada [9] and Napier [2].

2. The nature of binding mechanisms

Traditionally a binding consists of a name-value pair [15]. 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 performed 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 parts, listed below.

R-value or L-value bindings?

The first categorisation is that sometimes the bindings are to immutable values, that is constant values that do not alter during the period of the binding, and sometimes the bindings are to mutable values where the binding does not change although the value within it may. These kinds of bindings are traditionally known as R-value (for immutables) and L-value (for mutables) bindings in programming language semantics parlance [15]. The manifest constants of BCPL[13] or Pascal [17] are examples of R-value bindings and the compiler can make the binding statically. 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 it may vary at run time.

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 first declarations. Constants whose values are calculated dynamically, that is when the constant is created, which can be seen in S-algol[11] and the simple constant values of Ada[9] 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.

What scoping is involved?

A binding is always performed with reference to a particular environment. This may be performed statically or dynamically. The algol 60 scoping rule is static and allows duplicate bindings to be detected statically. Dynamic scoping can be seen in Lisp[10] and in the segment binding mechanism of Multics [8]. A binding of a file name to a file in an open statement is usually resolved in the dynamic environment of the program. 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 [14] user running the command wishes to bind to his or her own environment.

When is type checking performed?

Type checking, assuming it is performed at all, can be done 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 in read statements and in projections out of a union. Some languages such as SASL[16] deliberately choose run time type checking to facilitate polymorphism.

The binding mechanisms can be categorised by 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, file names.

We use the phrase Flexible Incremental Binding Set (FIBS) to describe the mixture of bindings required in a particular language. 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 [12] categories 9 and 13 for its dynamic constants and variables respectively.

In column 2, categories 2, 6, 10 and 14, it is difficult to perceive how such bindings may be used. For this column every unique name in the system must have the same type. If they did not we could not statically check the type.

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. In the bindings the scoping is static. However the type checking is not performed until the operator is applied to the operand. 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.

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 and the objects of Lisp. An applicative Lisp would use category 12.

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 source code on 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 statically about the bindings in such a system is impossible because the particular bindings that occur depend on 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. Persistent Systems

We have categorised a spectrum of binding mechanisms and assert that persistent systems need all of the spectrum to facilitate the needs of prospective users. We can consider the persistent store to be populated by objects which we would expect to be partially statically checked for safety. From this universe of objects we wish to allow

a the creation of new objects (and binding into the persistent store)
b the reuse of existing objects (program and data) by new objects
c new combinations of existing and/or new objects
d incremental construction of objects

We may wish to bind statically to a combination of objects in the persistent store in which case the objects are assembled into a new object bound together. On the other hand we may wish to obtain the latest version of the object and delay the binding until we actually use the object, as in Multics.

To perform any of the binding activities above, with a persistent store, requires a subject which has a description of the object to which it wishes to bind. For the

purpose of this paper we will assume that the description is provided in the form of a type although there is still much research in finding a type system that is rich enough for this purpose. Since the subject and object may be prepared separately, the types must match by some rule and structural equivalence is the appropriate type matching rule for this.

When an object is statically bound to a subject a static description of the object is required. For complete static binding throughout the system, a complete type description of the persistent store is required. This is clearly impracticable for large systems or for systems that are incrementally updated in parallel.

In general the persistent store forms a graph. To avoid having to describe the complete graph there must be nodes at which we can leave the description incomplete in static terms. In our experiments we have used a built-in infinite union of cross product types [12] to perform this action. Thus, in order to describe the part of the persistent store of interest, the user need only describe up to these infinite union nodes completely. To traverse these nodes, a projection out of the union (a dynamic bind) is necessary. This same technique is used in dynamic construct of the language Amber [5] which allows an infinite union of all types for this purpose.

It should be clear from the above that some form of dynamic binding is not only desirable but also necessary in persistent systems.

Some of our skill in using a persistent object store will be in deciding which objects are statically composed and never changed and which objects are dynamically composed. A judicious mixture of mechanisms has to be provided by the system and we would expect eager binding, in that, when it is appropriate to perform the binding, then it is done for safety and not delayed unnecessarily.

We will now investigate two languages to identify their flexible incremental binding sets and highlight their binding mechanisms. The first language, Ada, has a file system view of persistence whereas the second, Napier, has an object store view.

5.1 Bindings in Ada

Ada takes a very traditional view of program and data. Its philosophy is that program and data are quite separate and two different mechanisms, a library and a file system are used to provide the persistence in the system. The library contains all the statically bound program objects and uses the compiler and a loader to compose the library units into programs. The program may then dynamically create bindings and may bind to objects in the file store.

Sub-programs and packages are therefore composed statically into programs by the loader. These may, of course, contain code for dynamic bindings. Generics are a

good example of binding category 1. That is, a generic is a static R-value (to the compiler) which uses static typing and static scoping.

All data object binding is performed dynamically. Constants and variables are examples of binding categories 9 and 13 respectively. In the general case sub-type objects require a run-time check and therefore are bindings of categories 11 and 15. Selecting from a record by using a variant requires dynamic scoping to see if the record has the particular variant yielding binding category 15 only since constants are not allowed in records. Category 10 binding is used however to bind an exception to a particular name.

When we look at persistent data in files we see that the 'create' and 'open' statements act on an external file name binding it to a particular file. The file name, a string in this case, is evaluated in the dynamic environment, dynamically type checked and yields category 12 bindings for read file and category 16 bindings for write files. Thus, the FIBS for Ada is {1, 9, 10, 11, 12, 13, 15, 16}.

However, there are a number of hidden activities relating to binding within the files. The first is that any file has to be of a particular type. Ada has no union type and thus the whole of the file has to be completely described on use. Incremental building of the persistent store (a collection of file) is done by using the file itself effectively as the built-in infinite union which may be projected from by name. Thus sections of the persistent data are dynamically composed by program which knows about the particular files. No cross referencing between the files is possible.

It is also in the use of files that we expose the pretence of name equivalence. Each instantiation of the generic packages Sequential_io and Direct_io yields a unique file type. If this were the case we could not create a file in one program and use it in another because the file types would be different. The generic formal parameter type is converted on instantiation to the equivalent of an integer which is used to denote the file within the program. This is equivalent to structural equivalence of file type across programs.

5.2 Bindings in Napier

In Napier, all data is persistent. That is, data is kept for as long as it is usable. This we can determine from the fact that it is reachable by the computation of the transitive closure of objects from the persistence root, called PS. When a program terminates all of its data objects may be destroyed except those that the program has arranged to to reachable from PS. In general the persistent store will be a graph and objects which may be distributed over many machines.

Our preferred method of use of the persistent store is to navigate it with a software tool like a browser [7] that writes programs which bind to the persistent objects that we select. In such an environment the difference between compile time and

run time is blurred. Indeed our action in using the system is essentially setting up bindings and executing programs using these bindings. For persistent objects there is no notion of static R-values or static L-values and the bindings contained in rows 1 and 2 of the matrix do not apply. This should not be a surprise to us as rows 1 and 2 refer to bindings that are created by the compiler which may have disappeared. Such bindings may still be available if a compiler is employed to compile programs against the persistent store but will only be available for objects within the program and not persistent objects. For this reason they will not be discussed futher.

Simple bindings are introduced by a **let** declaration. For example

let a = readi ()

introduces the binding of 'a' to the integer value that will be read in. Since we have use '=' rather than ':=' to introduce the binding, the value is constant. This is an example of a dynamic R-value binding, category 9. Dynamic L-values, category 13, can also be introduced by

let a := 3.2

which binds the name a to the variable location which has the initial value '3.2'.

All simple object bindings are introduced in this manner and there is a one-one correspondence between this and the bindings introduced as procedure parameters. That is, the language obeys the Principle of Correspondence [15]. Before we can look at other bindings in the system, we must have a brief description of the universe of discourse of the language.

The Napier type system is loosely based on one suggested by Cardelli & Wegner[6]. All data objects in the system can be described by the following rules

1. the scalar data types are integer, real, boolean, string, pixel, picture and null.
2. image is the type of a rectangular matrix of pixels.
3. for any data type t, *t is the type of a vector with elements of type t.
4. for identifiers $I_1,..I_n$ and types $t_1,..t_n$, **structure**($I_1:t_1,...I_n:t_n$) is the type of a structure with fields I_i and corresponding types t_i, for i = 1..n.
5. for types $t_1,..t_n$, $t_1|...|t_n$ is the type of a union of the types t_i, for i = 1..n.
6. for any data types $t_1,..t_n$ and t, **proc**($t_1,...t_n$ -> t) is the type of a procedure with parameter
7. type parameterisation may be used in the type algebra.
8. for identifiers $I_1,..I_n$ and types $t_1,..t_n$, **env** is the type of an environment with entries I_i and corresponding types t_i, for i = 1..n.
9. for any identifier I and any type t, **abstype** I **with** t, is the type of an abstract data type.
10. any procedure type may be universally quantified.
11. type **any** is the infinite union of all types.

The universe of discourse of the language is defined by the closure of rules 1 and 2 under the recursive application of rules 3 to 11.

With regard to binding to the persistent store the data types **env** and **any** are the most important. Objects of type **env** are collections of bindings, that is quadruples of name, value, type and constancy. They differ from structures in that objects of this type belong to the infinite union of all such cross products. Furthermore we can add bindings to, or delete bindings from objects of type **env**. The distinguished point in the persistence graph, PS, has data type **env**. Type **any** is the infinite union of all types.

We can simulate the scoping mechanism of block structure using environments. We can collect the environments dynamically in some order that will be equivalent to entering blocks. We can then create new bindings in the environments in the same manner as we do from blocks. This is perhaps best explained by an example. We will write a program to place an integer and a procedure that operates on that integer into an environment and place it in the persistent store. Later we will retrieve the values and use them.

```
    let e = environment ()
    let rand := 2111 in e
```

This creates the dynamic L-value binding, category 13 and places the binding in the environment e. To use the binding we write

```
use e with rand : int in
begin
    let random = proc ( -> int)
    begin
        rand := (519 * rand) rem 8192
        rand
    end in e
end
```

We have a new example of binding here that is of interest. The **use** clause dynamically binds the name, type, constancy tuple to the environment expression. If the environment contains at least that name, type, constancy tuple then the binding succeeds and the name is available in the following clause. The binding, which occurs at run time, and is therefore dynamic, is similar to projecting out of a union. The difference here is that we only require a partial match. Other fields not mentioned in the **use** clause are invisible in the qualified clause and may not be used. Depending whether the value that we bind to is an R-value or an L-value we obtain binding categories 12 and 16 respectively. That is, in the L-value case, a dynamic L-value with dynamic type checking and dynamic scoping.

This kind of binding is the most dynamic form that can be found in a strongly typed system and is therefore sufficient for our needs. Notice that the binding once it is performed acts in the same manner that block structure does. That is, the environments may be collected in any order and act like nested blocks. Once the binding has been performed on entry to the **use** clause (block) then the scoping reverts to being static. This gives us the flexibility of constructing blocks either statically as we compose the program or dynamically as we run it.

6. 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 dynamic binding is necessary in persistent systems. Furthermore we have proposed structural equivalence as the appropriate type checking method and built-in infinite unions as the method of allowing a partial description of the persistent store.

To highlight the binding mechanisms we have described the binding in a traditional file based system, Ada, and an object based system, Napier.

7. 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.

8. References

1. Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. An approach tc
2. Atkinson, M.P. & Morrison, R.
   Types, bindings and parameters. Proc of the Appin workshop. Universities of Glasgow and S
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. international symposium on
5. Cardelli, L.
   Amber. A.T & T. Bell Laboratories (1984).
6. Cardelli, L. & Wegner, P.
   On understanding types, data abstraction and polymorphism. ACM.Computing Surveys 17, 4 (December 1985), 471-523.
7. Dearle, A. & Brown A.L.
   Safe Browsing in a strongly typed persistent environment.
   To be published in The Computer Journal 1988.
8. Dennis, J.R.

Segmentation and the design of multiprogramming computer systems. JACM 12, 4 (October 1965), 589-602.

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

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

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

12. PS-algol Reference Manual.
Universities of Glasgow and St Andrews, PPRR-12 (4thEdition).

13. Richards, M. BCPL a tool for compiler writing and systems programming.
AFIPS SJCC 34 (1969), 557-566.

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

15. Strachey, C.
Fundamental concepts in programming languages. Oxford University Press, Oxford (1967).

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

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