IBM Research

A NOTE ON THE CONCEPT OF BINDING

George Radin

March 17, 1971

RC 3287

Yorktown Heights, New York

San Jose, California

Zurich, Switzerland

A NOTE ON THE CONCEPT OF BINDING

George Radin
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT: This paper illustrates the notion of binding with examples of different kinds and different times between program preparation and execution. It proposes a definition of the term which encompasses those examples.

RC 3287 (#15045)
March 17, 1971
Programming and
programming languages

The notion of "binding", once an esoteric word in compiler writer's jargon, is gaining use in other areas of computer science (e.g. data management, CPU design). Like many words in this non-rigorous field, it is being used in the absence of a careful definition. Often it is taken as being synonymous with a subset of its reasonable domain - and not always the same subset. The purpose of this paper is to define the term and describe some of the more interesting system functions which it encompasses. We choose to postpone an explicit definition until the end of the paper, when the reader's intuition will have been adequately prepared to evaluate it.

A. What do we Bind?

I. Attribute Binding

Consider an automaton which can, among other things, perform addition. What must be presented to it is

a) a request to perform the addition,
b) the two values to be added, and
c) the location in which to store the result.

This can be given as an "instruction":
ADD 3 to 4, RESULT IN LOC 102.
This instruction is said to be "executable" by this automaton. Conversely, it may be that only instructions of this form are executable by this automaton.

Suppose, now, that we wish to write a program to perform some specific algorithm on a range of input. If the values of the operands to this add instruction are ultimately dependent on the input, we would like this instruction to be more general with respect to its operands so that the program need not be rewritten each time the input changes. We might choose the strategy that regardless of their values, the operands will be found in locations 100 and 101. The instruction now reads:

ADD VALUE (LOC 100) TO VALUE (LOC 101), RESULT IN LOC 102

This is a much more useful instruction in the sense that the program can now run correctly for a wider range of input. The only problem is that in fact the program cannot run at all, because this new form of the instruction is not executable by this automaton!

In order to run this program, we must build a facility (either by hardware or software) which takes this instruction, examines locations 100 and 101, and presents to the automaton a new instruction, in which the values found have been bound to the ADD instruction. Note that we have paid a cost and/or performance penalty for this improved program generality.

When can this binding operation be performed? Clearly, the latest time for doing the binding is just prior to execution of the instruction. Postponing it to "execution time" is always a possible strategy, and has many advantages. Since the program remains general longest, changes to the values of the operands

can be made freely, leaving the program correct. Also, the program can be re-used without change. Note, however, that each time the instruction is executed, this binding facility must be invoked, even if the values of the operands have not, in fact, changed. Thus, we have paid an additional penalty for this improved program flexibility.

Most instructions in existing computers have, in fact, chosen this "late binding" strategy with respect to the value of their operands although there are still many "immediate" - operand instructions available). The reasoning generally is that if the same operation is to be performed several times on the same operands it should, in fact, have only been requested once. Thus the burden is placed on the programmer (or the optimizing compiler) to eliminate this redundant computation.

Now, the earliest time for doing this binding is just after the value has been changed. If the value changes after the binding, the program is incorrect. Historically, the most common activity of an optimizing compiler is to analyze the control structure and execution paths of a program in order to discover the earliest possible "binding time" for an instruction consistent with all permissible input values. (In fact, the instruction is actually executed just once at this binding time, but the effect is exactly as though just the values were bound there.)

Since the compiler must choose the latest binding time required

for all input values, the program may not in fact be optimized for any but this "worst case". A strategy to overcome this is called dynamic binding, in which the operand locations are monitored and a new binding (or execution) invoked each time the values change. This strategy clearly provides the advantages of both earliest and latest binding strategies. The cost of implementing the monitor facility, however, often significantly outweighs these advantages.

Notice that the operation which the binder performed, regardless of when it was invoked, was the evaluation of the function VALUE, defined on a domain of locations. The implication is that there is a single, built-in VALUE function for all operands. Suppose, however, that we choose to make the binder more flexible so that it can execute more than one value-obtaining function. (Examples might be to retrieve different encodings of the values at the locations, to do conversions, to retrieve a subset of the data at the location, e.g. an element of an array:) Similarly, the assumption that there is a single "value-storing" function can be generalized as well. Thus, the addition instruction, as input to the binder, might read:

ADD FLOATING.POINT.VALUE (LOC 100) TO INTEGER.VALUE (LOC 101),
RESULT IS FLOATING POINT IN LOC 102.

(Notice that this is exactly the same (except for syntax) as defining a different instruction name for different operand types: e.g. ADD1 adds a floating point number to an integer, returning

a floating point number:

ADD1 LOC 100 TO LOC 101, RESULT IN LOC 102

Now again I might want to a write a program which was more general than this (i.e. one which executed correctly regardless of the encoding of the values at the locations.) I could choose the strategy that the name of the VALUE function would be found in a specified location:

ADD VALUE.FN.IN.LOC. 200 (LOC 100) TO VALUE.FN.IN.LOC. 201
(LOC 101), RESULT IS VALUE.STORING.FN.IN.LOC.202 (LOC 102)

Now the program is much more generally useful. It can execute correctly on different kinds of data. The penalty is precisely that an additional binding facility is required - to determine the representational (and organizational) aspects of the operands. Notice that these two bindings need not be executed at the same time. It is only necessary that the value-obtaining function be bound before the value is obtained, and that the value-storing function be bound before the result is stored. Highly interactive languages, such as APL, often bind value functions just before execution to permit dynamic modification of data attributes. Most conventional computer instructions, however, assume that these functions have already been bound before hardware execution. Similarly, most programming languages designed for optimized execution, restrict attribute modifications so that value function can be done at compile time. This, in fact,

is probably the single characteristic which most distinguishes between interactive and compiler oriented languages. (Some languages, such as PL/I, attempt to accommodate both facilities as desired.)

Generally the location of an element of data, its representation and its organization are all stored contiguously in what is called a "dictionary entry" for that element. In this case, only one location, that of this dictionary entry, need appear in the instruction:

```
ADD DICT.ENTRY (LOC 300) TO DICT.ENTRY (LOC 301),
      RESULT IN DICT.ENTRY (302).
LOC 300:  FLOATING POINT, SCALAR, LOC 100.
LOC 301:  INTEGER, SCALAR, LOC 101.
LOC 302:  FLOATING.POINT,SCALAR,LOC 102.
```

This is a useful approach in "data base" applications where the location, representation, and organizational description of the data is best stored outside the problem program itself. It not only makes the program more general and impervious to changes in these attributes of the data, but also permits protection from unauthorized use to be built into the binder - a system facility.

Again it is possible to optimize binding by monitoring the dictionary entries and re-binding an executable program when these attributes change. This is more easily done than monitoring data value changes, at least in systems where access to the dictionary is controlled. Thus, even in systems where dynamic

value binding is unfeasible, dynamic attribute binding may be the optimum strategy. This is especially true where the modifications required to "re-bind" a program to a different set of attributes is extensive.

In the examples above of value-obtaining (and storing) functions, two implicit restrictions were made:

a) the functions required only one argument, namely the location of the operand (or result);
b) the functions were known to the binder - i.e. seeing its name, the binder would know how to execute the function.

Both of these restrictions can be relaxed, permitting even greater generality of programs, again at the expense of a more complex binder. A common example of relaxing the first restriction is the value-obtaining function for elements of a vector (or array, or structure). Suppose, for instance, the second attribute in the dictionary entry at location 300 were ARRAY. Suppose it required two arguments for two dimensional arrays. Then the first operand in the instruction might read:

ADD DICT.ENTRY (LOC 300) (3,4)...

Notice that now, while the instruction is not bound to a specific value-obtaining function, it is bound to that subset which accepts two arguments. It is also bound to the values 3,4 as indices. We can repeat the same exercize for these indices as for the

operand value, to unbind the instruction from specific index values.

The second restriction can be relaxed by structuring the binder so that it can blindly invoke a function named in the dictionary entry. For example, we could choose to substitute the value-obtaining function SPARSE.ARRAY instead of ARRAY. The program, being unbound with respect to the function name will remain correct. What is required is an explicit interface definition between the binder and these value-obtaining (and storing) functions so that new ones can be written. (Our conservation principle still holds, since explicit interfaces never come free.)

To summarize then, given an automaton which can perform additions on two operands and return a result to a specified location, we wish to write a program which is more general with respect to permissible values than that required by the automaton. Therefore, we must provide facilities which, some time before execution, obtain the required values. We can write a program which is "unbound" only with respect to the value of the operand (and evaluate it by a fixed function on a specified location) or we can choose to write a program which is even unbound with respect to the value-obtaining (and storing) functions. The degree to which the program is unbound and the time at which the bindings occur trade directly with cost/performance/generality/flexibility/security aspects of the system.

II. Location Binding

In Section I, we discussed binding of values and other attributes.

We assumed, that the location of the value, or of the value-obtaining function, was explicit, hence bound, in the instruction.

However, already in the example of the dictionary entry, the location of the value was not bound to the instruction. Thus, until binding time, we could change this field in the dictionary entry and the program would still be correct. This kind of unbound addressing is called indirect addressing. It has three major advantages:

a) many instructions can "indirect" from the same dictionary entry;

b) other information (e.g. attributes) can be stored together with the location;

c) the data validity, security, etc. can be managed more easily by the system than if references to locations were explicitly spread out through the program.

Indirect addresses can be bound early by a compiler which can examine the flow of a program and determine that the dictionary entry will not change for all executions of a program. Or they can be bound somewhat later by a loader which discovers (usually by being told) that the dictionary entry will not change for this execution of a program. Quite often, however, indirect addresses are bound dynamically at each instruction execution. An example of this is the "base register" addressing of S/360 computers.

Here an instruction is bound, not to an operand address, but to an indirect address, the name of the base register. The effective address is computed by adding the contents of this base register to an explicit offset in the instruction just before each execution. Thus all the advantages of late address binding occur. Unfortunately, since there are far too few base registers available for all the different operands in a program, these must be shared by executing "LOAD" and "STORE" instructions explicitly in the program, thereby removing by one level many of the advantages of indirect addressing. (We are now faced with the requirement of indirect addressing the "LOAD" and "STORE"s.)

An increasingly popular solution to this difficulty is to partially bind addresses early, and to greatly increase the number of base registers so that they need never be shared explicitly by the program. This is accomplished by a large vector of base registers called a page table. The low order part of an address is explicitly contained in the instruction (inserted directly by the programmer or by some early binder). The instruction is unbound with respect to the high order part of the address. Instead a page table entry is given. Directly before execution of the instruction the page table entry is examined and the high order part of the address obtained. Since there are at least as many page table entries as high order addresses, the program need never explicitly LOAD and STORE these. (In fact, these page table entries are shared between users of the system. But this is done only by the system itself.)

The page table approach to late address binding, while quite useful, has two serious difficulties:

a) the page table becomes quite large, especially when each user has his own version which gets loaded when he is dispatched;

b) if users wish to share pages, then many page table entries are pointing to the same page address. When this address is changed (i.e. the information is moved to a different address) all these page table entries must be updated. This leads to significant system overhead.

The same solution fortunately addresses both problems. We simply increase the level of indirection by one for a part of the page table entry name, and call it a segment table entry name. Thus the operand address in the instruction now reads

seg name. page name. displacement

Since the page table is generally sparsely filled for a single user, this permits the size of the table to be significantly reduced. And if we make the assumption that sharing is done only by segment, we must change more than one table entry only when the page table is moved, an occurrence far less frequent than page moving.

We see then, that binding of a program to the address of its operands can be postponed until execution time usually only by binding the program to the address of a dictionary entry, register

name, page name, etc. The advantages gained are the following:

a) sharing of data can be accomplished with only one address update required when the data is moved;

b) all addressing of data is channelled through a single system-controlled table, permitting authorization checking, recovery, etc.;

c) the programs remain correct longer, since generally these tables move less frequently than the data.

In order to make this approach feasible, however, access to the tables must be very fast. In most systems, this is accomplished by storing the table entries in a contiguous address space, and addressing them by offset from the table origin. The programmer thus sees an address space which is multi-linear (e.g. S/360 Base Register Number, Index Register Number, Displacement). In some systems (e.g. S/360 model 67) the address space can also be thought of as being linear, where an address is simply the concatenation of segment number, page number, displacement. Since this latter looks to the programmer as simply an address in a different memory, it is generally called a virtual memory addressing scheme.

Thus the instruction is bound at execution time to real addresses, but is bound early to virtual addresses. This early binding can occur at program creation time, if the programmer writes in

Assembly language and maps his data onto the virtual memory. Often, however, the programmer writes in some more symbolic language, where he names his operands in a mnemonic way. In order to permit execution time binding to be done quickly (i.e. by offset in current technologies), some binding facility must be inserted between program preparation time and program execution time. (This facility is generally accomplished by a compiler, or a link editor, or both.)

B. When do we Bind?

Consider our automaton again which can perform addition by executing an instruction like

ADD1 3 TO 4, RESULT IN LOC 102.

Consider now a programmer who wishes to write an instruction like

Z = X + Y;

Let us follow this instruction through various possible binding times as they have been defined on existing systems.

I. Pre-Compile Time

It may be that X,Y and/or Z are data which exist (and are accessible) more globally thus just this program. They could be catalogued by the system, which has a dictionary entry giving their attributes and locations. If this is the case, then any processor which needs these attributes in order to perform its function, must bind them to the program.

Now most compiler-oriented languages require declarations of

the attributes of all variables before compilation can proceed correctly. (If these attributes are not given explicitly, default assumptions are usually made.) Thus a program containing globally declared X,Y,Z must, before compilation, be bound to the dictionary entries for these variables. If these dictionary entries change, the program becomes incorrect and we must repeat the pre-compile binding process.

An example of such a pre-compile binding facility is the % INCLUDE feature of the PL/I "Compile-time" capability. This facility is quite useful if one wants the programmer to be unaware of the attributes of data, but also wants the optimization advantages of compilation.

Note that the process can be automatic provided that

1) the system knows precisely which programs are bound to which parts of which dictionary entries, and
2) the system will not change a dictionary entry while a program which is bound to it is executing. (This implies that when the binding is later, and the program is, therefore, bound to the dictionary entry for a shorter time, dictionary updates may be made more quickly. If all programs are execution-time bound to all data and dictionaries, changes can be made between instruction executions. Note that even then one must wait until all processes have completed their instructions, and not allow them to start new instructions which could be bound to the changing dictionary entry.)

II. Compile time

Assume that we now have explicit declarations of the attributes for X,Y and Z. (These could have been supplied by the programmer, or bound by a pre-compile binder). One of the prime functions of a compiler is to examine a program, discover all the permissible binding which it can do, and perform this at compile time. The extent of this binding clearly depends on the language, and on the intelligence of the compiler. Some examples may illustrate this best:

1) Value binding:

Suppose there were two other instructions in this program

```
X = 3;
Y = 4;
```

and that the compiler could determine that they were executed before the instruction `Z = X + Y;` and that, furthermore, no subsequent executions could possibly change the values of X and Y until this third instruction was executed. Then the compiler could bind the values of X and Y to this instruction at compile time, and produce

```
Z = 3 + 4;
```

(Of course, most compilers would even do the addition at compile time, producing `Z = 7`, but from a binding point of view this is the same.

2) Attribute binding:

Suppose the declarations for X,Y and Z in this program

assured the compiler (by the rules of the language) that, for all executions of the instruction, X had the attribute floating point, Y integer, and Z floating point. Then the compiler could bind the instruction to these attributes, producing

ADD1 X TO Y, RESULT IN Z;

(using the definition of ADD1 given in Section A-I).

3) Location binding:

Suppose X,Y and Z were "local" variables to this program (i.e. accessible only during execution of this program). Then the compiler could assign addresses for the values of these variables, and bind the instruction to the addresses:

ADD1 VALUE (LOC 100) TO VALUE (LOC 101), RESULT IN LOC 102.

Many (so-called COMPILE -AND - GO) compilers do indeed bind to this level. The problem is that now this program can execute correctly only when X,Y and Z are actually stored at these locations. This is undesirable in multi-programming systems where the job should be executable in any available memory, or for AUTOMATIC (PL/I) or LOCAL (ALGOL) variables where one wants a new set of addresses for each invocation of a PROCEDURE or BLOCK.

But even in these cases, the compiler can do more than just pass on the symbols X,Y and Z. Since it knows that new addresses for these variables are always required together,

it can partially bind the addresses (e.g. the low order part) and reduce the high order parts to a single unbound symbol:

```
ADD1  VALUE (LOC "BASE" - displacement 0) TO
      VALUE (LOC "BASE" - displacement 1), RESULT IN
      LOC "BASE" - displacement 2.
```

We now require only a single additional binding of a partial address (i.e. BASE) in order to have three fully bound addresses.

A compiler normally does not see the entire program at compile time. Generally it gets a single subroutine. Local variables then can be more or less tightly bound, as illustrated above. Variables which are accessible to other parts of the program, however, cannot generally be bound quite as tightly. For instance, if X,Y and Z were not local, the compiler could not have chosen to bind these displacements in the order given in Section II. For, when compiling another subroutine, a different order could have been chosen (e.g. YZX, or even XQYZ, where Q is used in the second subroutine).

Thus the compiler normally has no choice but to avoid all binding, and just compile the symbolic names intact. Since this is inefficient generally, many languages have introduced restrictions in order to permit some compile-time binding. PL/I, for instance, requires that attributes of EXTERNAL variables match exactly in all procedures in which they appear, thereby permitting attribute binding at compile time. FORTRAN goes even further, requiring that all

variables in the same COMMON area be declared in the same order in all subroutines, thereby allowing partial address binding of global variables.

III. Link Edit and Load Time

Thus we see that the output of a compiler still generally contains partially unbound addresses. When these addresses are, in fact, changeable during execution of a program (e.g. AUTOMATIC variables in PL/I) no further binding can occur prior to execution. But often two other early binding possibilities exist:

a) Suppose a program consists of several subroutines each containing the unbound symbols X, BASE1, Q. Now a processor can be invoked after compilation which gathers up all of these symbols and again partially binds them, creating say

NEW.BASE - displacement 0 for X

NEW.BASE - displacement 1 for BASE1

and NEW.BASE - displacement 2 for Q.

Now the program is still "relocatable" each time it is to be executed, but the number of symbols to be bound is reduced again. This post-compilation, pre-loader processor is called a Link Editor.

b) Finally, when a program is loaded into memory, a processor generally called a Relocatable Loader is invoked to bind the program to all other addresses which will not change

during this execution.

(Note that, while the examples in this paper are all data variables, the same binding considerations apply to values, attributes, and addresses of sub-programs and instructions. Thus, the program itself normally has one or more unbound symbols which are bound by the Relocatable Loader.)

IV. CALL Time

Many languages whose programs are organized into procedures (or subroutines, or blocks) carefully specify that certain attributes or locations may be bound when a procedure is called. Thus a binding process (called a Prologue) is invoked at each CALL. It does the following kinds of binding:

a) In Section II we described AUTOMATIC or LOCAL variables which required new addresses at each procedure invocation. The prologue will obtain a block of storage from the system and bind the variable references to the address of this block. For instance, if X,Y and Z were AUTOMATIC and partially bound to

BASE - displacement 0,1,2 respectively,

the prologue might bind BASE to Loc. 100, which effectively translates our instruction to

ADD1 VALUE (LOC 100) TO etc.

for the duration of this invocation of the procedure.

In addition some languages permit attribute binding for

AUTOMATIC variables to be postponed until CALL time. For instance, in PL/I the dimensions (but not the dimensionality) of an array and the length of a character string may be bound by the prologue. Normally, however, languages which allow attributes to be unbound after compilation go all the way and bind at execution time. This is because, in order to take advantage of CALL time attribute binding generally, the Prologue would have to assume much of the capability of a compiler.

b) The most common form of CALL time binding is of variables declared as parameters in the procedure. Suppose our addition instruction were contained in a procedure and X,Y and Z were declared to be parameters to this procedure:

e.g., in PL/I:

```
Q: PROCEDURE (X,Y,Z);
   Z = X + Y;
```

Suppose Q were called in another procedure by the PL/I statements

```
A = 1;
B = 2;
CALL Q(A,B+A,A);
```

Then at CALL time, depending on the specific language rules and on the declarations, the following kinds of binding occur:

1) Value Binding:

At call time, X could be given to the value which A had when the CALL was executed. Then, since there are

no intervening assignments to X, our addition instruction is bound to this value. Notice that no subsequent change in the value of A effects this binding. For instance, after the addition instruction is executed, X will still have the value 1, regardless of whether A's value has changed.

2) Address Binding:

All three parameters could be bound by address at prologue time. This implies that at the CALL B+A was evaluated and stored somewhere, say loc. 100. Suppose A was in loc. 101. Then, after binding our instruction reads:

```
ADD1 VALUE (LOC 101) TO VALUE (LOC 100), RESULT
            IN LOC 101.
```

Notice that now, after execution of the addition instruction, X has had its value changed. Languages (such as PL/I) which permit address binding of parameters often prevent much optimization at compile time, since the compiler must always expect that this kind of implicit "aliasing" might occur.

3) Evaluation binding:

Suppose the calling procedure, instead of evaluating B + A, prepared a value-obtaining function which,

when invoked, returned the value of B + A. We would (as in By-name calls in ALGOL) bind the add instruction to this function, as:

ADD1 VALUE (LOC 101) TO FUNC, RESULT IN LOC 101.

Notice that now, after execution of this instruction, not only X, but also Y has had its value changed.

Although it may appear to be an esoteric form of binding, this facility is quite useful in so-called "Data Base" systems where it is desirable that a special value-obtaining function be invoked at execution. It might be, for instance, that the value must be converted dynamically, or that the system can manage simultaneous access - change requests.

(Note that, unfortunately, the programmer cannot ignore which kind of binding the system has chosen to perform since, as we have seen, the results differ widely.)

4) Symbol binding:

Finally, we can specify a symbol-to-symbol CALL time binding where, for instance, the actual symbol A is substituted for the symbol X. This results in the necessity, at execution time, to bind the operand completely from its completely unbound symbolic form. It is the latest possible binding and requires a capability almost equivalent to a machine which can directly execute source

language programs. While this is generally too expensive to be feasible in most programming situations, there are applications where it has been found very useful (e.g. highly interactive program creation, where the user wishes to be able to modify the program in strange ways very dynamically).

V. Pre-Execution Time:

There are many languages which permit binding operations to be performed after the Prologue, but prior to instruction execution. These are generally triggered by explicit instructions, such as:

a) LOAD, STORE base (or index) registers in S/360 machine language;

b) ALLOCATE, FREE controlled storage in PL/I, which creates a new address for the allocated variable;

c) OPEN, CLOSE in OS/360 assembly language and in PL/I which bind the attributes of data on secondary storage earlier than READ/WRITE time so that these latter can be more efficiently performed.

VI Execution Time:

Finally, of course, in order to execute correctly, the instruction

must be in the form

ADD1 3 TO 4, RESULT IN LOC 102.

Thus, whatever binding has been postponed must be performed at this time.

C. What is Binding?

We now propose a definition of binding which, hopefully, will include in a reasonable way all instances hitherto discussed.

Consider an automaton which we ask to execute a program (perhaps as small as one statement). If it executes this program correctly, we say that this program is bound with respect to this automaton.

If the automaton cannot execute the program because it does not have sufficient information about the operands of the program, we say that the program is unbound in those aspects with respect to this automaton. We define binding as the process of augmenting or modifying the program by supplying information about the operands so that the automaton can execute the program correctly.

Notice that we cannot say whether a program is bound or unbound without defining the automaton which is to execute it. For instance, an APL program may be bound with respect to an APL interpretor but unbound with respect to a S/360 computer.

Notice also that whether a program is bound or unbound depends upon whether the automaton executes "correctly" (that is, gives us an answer which we are prepared to accept).

Thus the process of binding a program depends on how we have split functions between the automaton and the program, and how we have split operand information between the program and the data. It is a good name for the concept because it is in fact the glue which reseals these separate components which have been built separately in order to make them more generally useful.