Dynamic Binding of Separately Compiled Objects Under Program Control

Rex E. Gantenbein
Department of Computer Science
University of Wyoming
Laramie, WY 82071

Douglas W. Jones
Department of Computer Science
University of Iowa
Iowa City, IA 52242

Abstract

Most high-level programming languages do not have the ability to control the bindings between internal names and external implementations of those names at run time. This facility is necessary for the use of the language in a monolingual programming environment. The paper describes the semantics of a "dynamic binding" feature for a block-structured, strongly typed language that supports such control and presents an experimental implementation of Pascal that contains dynamic binding.

Introduction

An area of software engineering receiving increased attention lately is the "programming environment", a variation on the operating system that is primarily concerned with the support of high-level language development. A common thread through much programming environment research is the trend toward "monolingual" environments, in which no system-specific job control language or shell is needed to provide the connections between the programs making up a large application. In [Heering85], the convergence of programming and control languages is shown to depend upon the ability of the unified language to treat (1) procedures and programs and (2) files and types as equivalent. Many high-level programming languages have this ability, some of which are mentioned in the article; however, any language that is to be used to write a command interpreter must provide some sort of dynamic binding facility to link programs together.

Although most operating systems provide dynamic linkage services that allow one running program to cause other programs to be loaded and run, such services are not usually provided in high-level languages. Instead, control over binding at execution time usually involves access of the binding facilities of the underlying operating system. The Multics dynamic linker [Honeywell75], for example, is automatically called whenever a program references an unbound external name; after the linker has bound the name, the program may call a system service to terminate the binding so that the next reference to that name will relink it, possibly to a different object. It should be noted that the system services are outside the semantics of the programming languages involved; the linkers do not perform type checking and programs using these linkers are not portable.

There appears to be no high-level language that provides complete control over bindings between names declared within a program and procedural objects external to the program while assuring the consistency and correctness of the bindings. Algol 68 [vanWijngaarden76] allows strongly typed dynamic binding within the confines of a single program. In this language, it is legal to define procedure variables, or pointers to procedures; these pointers may be bound to any compatible procedure (where compatibility is determined by the structural equivalence of the argument list and result), but no provision is made for binding a procedure name to an external object. Mesa [Mitchell79] allows somewhat more flexibility; here, the language definition includes separate compilation, with strong type checking enforced across compilation-unit boundaries by the Mesa binder. Furthermore, as in Multics, the binder can be dynamically invoked to add new code to a running program. Unfortunately, once a binding has been established, it cannot be safely broken. As a result, running Mesa systems tend to grow and must be periodically restarted to prune off unneeded linkages.

A complete implementation of dynamic binding in a strongly typed language would involve all of the aspects described above. The language would have to allow programs to be constructed from separately compiled pieces and it would have to include constructs for dynamically controlling the binding between internal names and these external pieces. If the problems illustrated by Mesa are to be avoided, the language should provide some method of unbinding or rebinding a previously bound name. In addition, the data typing and data sharing rules should not depend on whether the program pieces were bound together at compile time or run time.

The following program fragment illustrates how a dynamic binding mechanism can provide a monolingual programming environment:

```
program Shell ( ... );
type a = ... ;
    ...
var b: a;
    FileName: string;
    ...
procedure c (var b: a ); dynamic;
begin
  repeat
    ...
    link 'c' to FileName;
    c;
  until ...;
end {Shell }.
```

This example is written in a version of Pascal extended to support dynamic binding. The program repeatedly binds the dynamically linkable procedure "c" to implementations stored in object files named by successive values of the variable "FileName". As such, "Shell" serves as a control program to mediate the communication between the procedure or program implementations linked to the name "c"; since these are written in the same language, this is a monolingual environment. Furthermore, the global variable "b" has a lifetime determined by the lifetime of the program "Shell", not by the duration of a call to "c" or the lifetimes of the object files containing implementations of "c". In this way, dynamic binding can achieve a form of control over persistent data (data that outlives the program that creates it), which has been shown to be one way of implementing typed files [Atkinson82]. There is nothing to prevent the recompilation or replacement of implementations of "c" between calls, and given an appropriate checkpoint mechanism, "Shell" can run forever.

Semantics of Dynamic Binding

Program development in a high-level language can be viewed as a three-stage process. The programmer writes the source code; the compiler translates the high-level code into machine language; and the run-time system executes the resultant code. In a modular language, these steps may involve several pieces of code, in which references across the compilation unit boundaries must eventually be resolved.

Separate compilation provides the support for the programmer's development of a modular program. A linker supports the compiler's translation of separate source code units by allowing the resolution of external references to be postponed. In most systems, the linkage mechanism resolves all external references at the time of linking. Those systems that support run-time modularity (i.e., dynamic binding) generally depend on operating system services, not on the language and its run-time facilities. Adding dynamic binding to a language moves this run-time modularity into the semantic domain of the language. This section explores the desired semantics of dynamic binding in the context of a strongly typed, block-structured language.

The compile-time characteristics of a language with dynamic binding should closely resemble those of the language without it. A language should not be fundamentally changed by the inclusion or exclusion of this feature; in particular, programs not using an available feature should be semantically identical to those constructed without access to the feature. Binding can therefore be controlled by a statement in the language that specifies the name and object to be bound.

For a language to achieve run-time modularity, it is essential that the language support compile-time modularity with separate compilation. This facility is no longer exotic, having been included in many languages, most notably Ada* and Modula-2; the feature of separate compilation of particular importance to the current discussion is the ability of a compiler to completely type check the separately compiled units. This type checking must be performed for the local identifiers, for external references to identifiers, and for the interface by which a unit specifies an object to be dynamically linked to an internal name. Complete type checking is necessary if there is to be any guarantee that an object correctly implements the name to which it will be bound. Of course, run-time type checking is a possible alternative, but the expenses of this approach are so high that it should be avoided if possible.

---

* Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

The rules for legal use of separately compiled, dynamically bound objects are clearly related to type equivalence rules. Thus, the choice must be made between structural equivalence and name equivalence. If structural equivalence is used, each compilation unit must include a structural definition of the context in which it may be used, and this must be compared with the actual context when linkage is attempted. Name equivalence requires that compilation units be compiled for use in some previously defined context. The Ada "subprogram is separate" clause and "package" specification [DoD83] and the Mesa "definitions module" [Mitchell79] may be considered to be definitions of contexts for this use.

When a dynamic binding mechanism is called on to link a name "N" to some object "o" at run time, the following steps are involved:

1. Determine if it is safe to rebind "N".
2. Decouple "N" from its current binding if it has one.
3. Locate the object "o".
4. Determine if binding "N" to "o" is legal.
5. Bind "N" to "o".

These steps can be described in terms of three relatively independent semantic issues:

a. The dynamic binding mechanism (steps 2 and 5).
b. An object-locating scheme (step 3).
c. Replacement safety and consistency (steps 1 and 4).

These issues are independent in the sense that there are many different ways of approaching each issue, and the particular approach taken in dealing with one issue has little effect on the approaches that may be used to deal with the other two. The following discussion addresses some of the possible design decisions in each of these areas.

What things can be dynamically bound? This is one of the first questions that must be asked when designing a dynamic binding mechanism. A natural way to answer the question is to list all the kinds of things that have names and determine which of them can be dynamically bound. Procedures, functions, and abstract data type implementations are all reasonable candidates since each is a natural unit of separate compilation. Simple variables may be viewed as dynamically bound constants, and pointer variables may be viewed as dynamically bound variables. Languages with label or type variables may be considered to support dynamic binding of labels and types.

At the heart of any dynamic binding mechanism, there must be a loader to include the separately compiled material in the address space of the running program, as well as a linkage mechanism to connect dynamically bound names with newly loaded objects. If the linkage is made through an indirect link, as in the Multics "combined linkage region", dynamic bindings are easily changed. If the linkage is made by modifying the address fields of each instruction that references the dynamically bound object, bindings are harder to change. If user programs may obtain and store actual code addresses in user-managed data structures, bindings are almost impossible to change. If the indirect links are stored in activation records, many bindings of a name may coexist (perhaps as many as there are activations of the declaring block). Of course, multiple bindings may only coexist if the loader is able to load different bindings of a name into different memory locations.

Locating an object to be bound by the dynamic binding mechanism on most computers undoubtedly means locating an object-code file produced by the compiler. Thus, the locating scheme must involve searching the file directory structure for a file name specified by the linkage instruction. Many operating systems provide a system service that does this searching, but this solution takes the problem out of the semantic domain of the language. An approach more suited to the stated intent of this work would be to include in the run-time support facilities of the language a searching mechanism that returns the location of the desired file. This location could be a simple address; a more sophisticated approach might make use of capabilities, given an operating system that supports capability-based addressing [Fabry74]. A third alternative is to put the burden on the programmer; by requiring that the access path to the file be completely specified as part of the linkage statement, the run-time support system could locate the object file directly without any need to search directories.

Replacement and consistency problems may arise in languages where names can be dynamically re-bound after one binding has already been established. In [Brosgol76], two problems are identified: that of converting instantiations from one form to another when rebinding occurs, and that of type identity when multiple bindings are allowed to coexist and interact. A patented software system developed by Bell Laboratories that permits individual procedures to be dynamically replaced adopts a very simple solution: No binding may be changed except when all instances of dynamically bound objects have been deleted [Bishop84]. This is described as

the "clear stack" condition, and it is up to the programmer to indicate when this condition is true in a program. More complex solutions may involve canonical forms for instantiations so that conversion is possible. An intermediately complex solution involves disallowing the rebinding of names when any instances of that name exist; although some run-time checking will undoubtedly be required, compile-time checking may be able to partially support such a solution.

An Experimental Implementation

In order to more fully explore the semantic issues involved in dynamic binding, a programming language incorporating dynamic binding was implemented. For the purpose of this experiment, the methodology described in [Marlin83] was used; that is, a base language was selected, the semantics of the feature were designed before any syntactic commitments were made, the semantic design problem was broken into largely orthogonal pieces, and specific abstract models were used to help evaluate the alternatives for each piece.

Pascal was selected as the base language for this experiment because it is well known and because easily modified partial implementations such as Pascal-P [Barron81] are available. The experimental language incorporating dynamic binding, "db-Pascal", was implemented by modifying the Pascal-P compiler to support separate compilation and compile-time safety checks, and by modifying the P-code interpreter to include dynamic loading and linking, as well as run-time safety checking.

Of the named objects supported by Pascal, procedures and functions are the obvious candidates for dynamic linkage. An important question is: Do results obtained for procedures in Pascal apply to abstract data type implementations such as those provided by Ada packages or Mesa and Modula-2 modules? In [Linden76], it is argued that procedures and modules are equivalent, while [JonesD81] argues that modular and object-oriented encapsulation mechanisms provide equivalent support for abstract data type implementation. Intuitively, this equivalence can be seen by noting that the operations on an abstract data type may all be formulated as procedures, and that rebinding the implementation of the abstract data type is equivalent to rebinding all of the procedures (including those that allocate space for instances of the type).

The mechanisms already in Pascal for forward procedure declaration provided useful direction for the introduction of separate compilation into the language. The Pascal forward declaration allows the compiler to check the correctness of code that calls a procedure or function before the compiler has encountered the procedure or function body. Dynamically linked procedures can use the same facility, but an additional mechanism is needed to allow the compiler to check that the separately compiled body conforms to the definition used by the calling program.

The "compiler suspension" model of [LeBlanc79] describes the semantics of separate compilation in db-Pascal. In this model, the (abstract) compiler executes a "fork" operation when it encounters a stub defining a separately compiled object. One copy of the forked compiler continues compilation of the source program containing the stub, and the other copy is suspended, ready to begin compilation of the body of the separately compiled object. (Separately compiled object modules are stored as P-code assembly language files.) The result is that the semantics of the separately compiled object are the same as they would have been, had the text of the object been inserted immediately after the stub. Thus, stubs may appear at any nesting level.

In order to implement separate compilation and dynamic binding together, a new compiler directive was added to Pascal, dynamic. (Note that forward is the only predefined compiler directive in standard Pascal.) When the compiler encounteres the dynamic directive, it dumps its symbol table into a file with the same name as the stub in a directory associated with the source file. This symbol table contains definitions of all label, type, variable, procedure, and function names that are globally accessible from within implementations bound to the stub as well as the formal parameter list of the stub. Each compilation unit begins with an "environment heading" indicating the stub for which a body is being compiled. When the compiler encounters this header, it initializes its symbol table from the associated file. The standard Pascal "program heading" can be viewed as an environment heading for the "user program" stub of the system command language interpreter.

A new simple statement, link, provides program control over the binding between an internal procedure stub and an external procedure body. This statement specifies the name for the procedure and the access path to the file containing the object code for the procedure body. The access path to the object file is assumed to start in the directory in which the source code is located. The compiler stores the access path as part of the compiled code for the link statement; the use of the path name eliminates the need for a run-time locating mechanism.

The extensions of Pascal-P that characterize db-Pascal are summarized in the syntax specifications below. Additions to the original specifications are underscored. The notation is that of [IEEE83], whose notion of "extension" is also used here.

```
simple-statement = empty-statement |
                   assignment-statement |
                   procedure-statement |
                   goto-statement |
                   link-statement

link-statement = "link" procedure-name-str
                 "to" procedure-body-str

procedure-name-str = apostrophe-image
                     procedure-identifier
                     apostrophe-image

procedure-body-str = apostrophe-image
                     path-name
                     apostrophe-image

path-name = {directory-identifier "/"}
            file-name

directory-identifier = identifier

file-name = identifier
```

Note that, in this specification, directives are not explicitly included; this is consistent with [IEEE83], which does not specify directives except to say that forward is required in all implementations.

At run time, execution of the link statement initiates the dynamic linkage mechanism. The P-code interpreter assembles the indicated object file (via the path name specified by the link statement in the original source) into an unused code segment in its internal code array. Db-Pascal only permits one implementation of a given procedure to exist at any time, so re-linking a previously linked name can re-use the space used by the previous implementation in the code array. A global indirect link suffices to transfer control to dynamically linked routines.

The only threat to safety in db-Pascal involves attempts to rebind procedures that are active. The decision to allow only one implementation of any stub at any time is the root cause of this threat. If a procedure were to rebind itself or call another procedure that rebinds it, chaos would result. The db-Pascal compiler contains static checks to prevent a procedure from rebinding itself or any statically enclosing procedure. The db-Pascal P-code interpreter maintains, with each indirect link to a dynamically bound routine, a count of the number of activation records currently in existence for that routine; attempts to rebind a routine with a non-zero count cause a fatal run-time error.

This implementation was carried out on a VAX 11/780 computer under Berkeley 4.2 Unix* during the summer of 1985. The introduction of separate compilation into the Pascal-P compiler required approximately two man-weeks of work. Another four man-weeks were spent to implement the dynamic binding mechanism in the P-code interpreter and to make the associated changes for compile-time safety checks in the compiler. The new implementation is neither complete nor efficient, but it demonstrates the relative ease with which dynamic binding can be added to an existing language.

The major limitation of db-Pascal is that only procedures can be dynamically bound; the implementation also suffers from many problems inherent in P-code systems. It does, however, indicate that dynamic binding is a feasible extension for a strongly typed, block-structured language. The implementation described here allows programs to be written along the lines of the example at the beginning of this paper; if a checkpoint mechanism were included in the language, it would also allow persistent programming as described in [Atkinson82]. A complete monolingual environment would require that an exception handling mechanism, like that presented in [Turba85], be defined so that execution errors do not cause an exit from the environment; this mechanism can be easily implemented with nonlocal gotos, which unfortunately are not implemented in the Pascal-P compiler from which the system was built. Also required would be a global command interpreter, written in db-Pascal, under which all user programs (including the db-Pascal compiler) could run.

---

* Unix is a registered trademark of Bell Laboratories.

References

[Atkinson82] Atkinson, M. P., et al. "PS-Algol: an Algol with a persistent heap," SIGPLAN Notices 17, 7 (July 1982) 24-31.

[Barron81] Barron, D.W., ed. Pascal: The Language and its Implementation, John Wiley and Sons, Chichester (1981).

[Bishop84] Bishop, T. "Online transaction processing," U. of Iowa Dept. of Computer Science Colloqium (28 September 1984).

[Brosgol76] Brosgol, B. M. "Some issues in data types and type checking," Design and Implementation of Programming Languages, Springer-Verlag Lecture Notes in Computer Science 54 (1976).

[DoD83] U. S. Dept. of Defense. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A (1983).

[Fabry74] Fabry, R. S. "Capability-based addressing," Comm. of the ACM 17, 7 (July 1974) 403-412.

[Heering85] Heering, J., and Klint, P. "Towards monolingual programming environments," ACM TOPLAS 7, 2 (April 1985) 183-213.

[Honeywell75] Honeywell Information Systems. Multics Programmers' Manual and Reference Guide, Document Order No. AG91 (December 1975).

[IEEE83] IEEE Standard Pascal Computer Programming Language, Inst. of Electrical and Electronic Engineers, New York (1983).

[JonesD81] Jones, D. W. The Systematic Design of a Protection Mechanism to Support a High-Level Language, Ph.D. Thesis, U. of Illinois (1980). Published as U. of Iowa Technical Report TR 83-05 (August 1983).

[LeBlanc79] LeBlanc, R. J., and Fischer, C. N. "On implementing separate compilation in block-structured languages," SIGPLAN Notices 14, 6 (August 1979) 139-143.

[Linden76] Linden, T. A. "The use of abstract data types to simplify program modification," SIGPLAN Notices 11, Special Issue (1976) 12-23.

[Marlin83] Marlin, C. D. "A methodical approach to the design of programming languages and its application to the design of a coroutine language," U. of Iowa Technical Report TR 83-05 (August 1983).

[Mitchell79] Mitchell, J. G., Maybury, W., and Sweet, R. Mesa Reference Manual, Version 5.0, Xerox Corp. (April 1979).

[Turba85] Turba, T. N., and Whitelaw, M. W. "The Pascal exception handling proposal," SIGPLAN Notices 20, 8 (August 1985), 99-106.

[vanWijngaarden76] van Wijngaarden, A., et al. Revised Report on the Algorithmic Language Algol 68, Springer-Verlag (1976).

Association for Computing Machinery

CSC '86

CINCINNATI

COMPUTER SCIENCE IN FOCUS: 1986

1986 ACM

Fourteenth Annual

COMPUTER SCIENCE CONFERENCE®

February 4-6, 1986

Cincinnati, Ohio

PROCEEDINGS

acm