THE BINDING MODEL: A SEMANTIC BASE FOR MODULAR PROGRAMMING SYSTEMS

D. Austin Henderson, Jr.

Massachusetts Institute of Technology

Prepared for:

Office of Naval Research
Advanced Research Projects Agency
National Science Foundation

February 1975

BIBLIOGRAPHIC DATA SHEET	1. Report No. GJ34671 + N00014-70-A-0362-0001 MAC TR-145	2.	3. Recipient's Accession No. AD/A006 961
4. Title and Subtitle The Binding Model: A Semantic Base for Modular Programming Systems			5. Report Date: Issued February 1975 6.
7. Author(s) D. Austin Henderson, Jr.			8. Performing Organization Rept. No. MAC TR-145
9. Performing Organization Name and Address PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY: 545 Technology Square, Cambridge, Massachusetts 02139			10. Project/Task/Work Unit No. 11. Contract/Grant No. GJ34671 N00014-70-A-0362-0001
12. Sponsoring Organization Name and Address Office of Naval Research Department of the Navy Information Systems Program Arlington, Va 22217 Associate Program Director Office of Computing Activities National Science Foundation Washington, D. C. 20550			13. Type of Report & Period Covered: Interim Scientific Report 14.
15. Supplementary Notes			

16. Abstracts: A programming system is a computer system which supports a community of programmers who can make use of one another's work. A module is a (possibly complex) construction usually comprising both programs and data which will provide some service. A programming system is said to be modular if modules can be constructed within the system from existing modules using only knowledge about the behaviour of those existing modules. In such a system: mechanisms must exist which permit any module to be used by any other (the system must be flexible); modules must be responsible for supplying all the modules they need in order to realize their behaviours (modules must be self-sufficient); modules must not conflict with one another when used together (sets of modules must be compatible); and module behaviour must be independent of the identity of the modules which invoke the modules (modules must be non-discriminatory). The Binding Model is an abstract machine designed as an "ideal" kernel for modular programming systems.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

PRICES SUBJECT TO CHANGE

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Approved for Public Release; Distribution Unlimited	UNCLASSIFIED	285
	20. Security Class (This Page) UNCLASSIFIED	22. Price 8.75-2.25

MAC TR-145

THE BINDING MODEL:
A SEMANTIC BASE FOR MODULAR PROGRAMMING SYSTEMS

D. Austin Henderson, Jr.

February 1975

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

THE BINDING MODEL:
A SEMANTIC BASE FOR MODULAR PROGRAMMING SYSTEMS

by

Dugald Austin Henderson, Jr.

Submitted to the Department of Electrical Engineering on January 24, 1975 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

ABSTRACT

A programming system is a computer system which supports a community of programmers who can make use of one another's work. A module is a (possibly complex) construction usually comprising both programs and data which will provide some service. A programming system is said to be modular if modules can be constructed within the system from existing modules using only knowledge about the behaviour of those existing modules. In such a system: mechanisms must exist which permit any module to be used by any other (the system must be flexible); modules must be responsible for supplying all the modules they need in order to realize their behaviours (modules must be self-sufficient); modules must not conflict with one another when used together (sets of modules must be compatible); and module behaviour must be independent of the identity of the modules which invoke the modules (modules must be non-discriminatory).

For various reasons, existing programming systems are not modular.

The Binding Model is an abstract machine designed as an "ideal" kernel for modular programming systems. It is based on the notions of "location-less" data, unrestricted structuring, limited mutability, explicit binding, and controlled access to data.

The value of the model is established in two ways: Firstly, examples are used to demonstrate that common constructs of programming languages and operating systems are realizable in the model. Secondly, it is shown that two kinds of desirable special behaviour can be given formal definitions in the model, and that there are large, structurally-defined classes of programs which satisfy these definitions.

THESIS SUPERVISOR: Jack B. Dennis
TITLE: Professor of Electrical Engineering

Acknowledgments.

The creation of this document has taken me a long time. Without "a little help from my friends", I never would have made it:

From my masters at Upper Canada College, and my instructors at Queen's who helped me gain confidence in my ability to think, and learn the pleasure of doing so.

From my summer and part time employers: Canada Life, IBM Mount Royal, Avon Products of Canada, Argonne National Laboratory, Lincoln Laboratory, Massachusetts Computer Associates, and Bolt Beranek and Newman (BBN) who introduced me to computers and the wide world of their practical application to the needs of the users whom they serve in so many ways. Out of these experiences grew the appreciation of the difficulties which are the main thrust of this work.

From members of the Programming Linguistics Group at MIT, and in particular Arthur Evans, Jr., who helped equip me with the knowledge of programming languages which underlies the design of the Binding Model.

From Bob Thomas at BBN whose collaboration in some early experiments in using the ARPA Network helped crystalized my interest in the use of names.

From Pete Jessel, Greg Pfister, and Steve Zilles, fellow graduate students and co-conspirators in founding and maintaining "The Longest-Established, Permanent, Floating Bull Session in Project MAC". In addition to broadening my horizons, all of these did regular duty in picking up the discouraged pieces of Henderson, helping me re-group, and setting me going again. Steve in particular put in innumerable hours assisting me in getting my head straightened out on details of this work.

From Jack Dennis, my supervisor, who oversaw the whole project, suggested the study of the special behaviours, acted as continuing "devil's advocate" on the use of cells, and was an uncompromising editor of my many

attempts to write English. That this report is at all readable is largely Jack's fault.

From Barbara Liskov, a reader, and friendly critic. Always, and especially when the going was rough, her enthusiastic involvement in her own research was an inspiration, and her willingness to spend time with students, even when time was most at a premium, made her an invaluable resource.

From Bert (W.R.) Sutherland, my other reader and "old friend". Bert's persistent support over many years, through an aborted earlier piece of research, and through the formulation and execution of this work has been invaluable. His technical suggestions have forced me to keep taking fresh looks at my work; more than any other single factor, his advice on the process of doing research has been responsible for my finally having learned what doing research is all about, and as a result getting this piece of research completed; and his concern for my intellectual and spiritual well-being has on many occasions helped me to have faith enough in myself to believe that "despite my best efforts, progress was in fact being made".

From the people who "are" MIT. They have provided an educational environment flexible enough to permit a student to tap the many excellent resources for learning in Computer Science that exist in the Boston area. And, more specifically, for many years they have born with a sometimes not overly diligent and often intellectually hesitant student.

From the agencies which supported my work through their funding: ARPA, NSF and IBM (see below). They have provided me with "bed, bread, and beer".

From Marsha Baker and Marty Richardson who spent many hours at a terminal entering the original text of this work from fairly illegible manuscripts.

From Marty Richardson, who did the drawing in the figures.

From Lincoln Laboratory, and particularly from Jim Forgie, Jack Raffel, and Jack Mitchell. They have provided me with the services of the TX-2 for use in preparing the text of this document. This truly marvelous facility, while at times offering a significant distraction from my studies, has been formative in the ideas of the work and invaluable to one for whom the conversion of strings of words into English sentences conveying his intended meaning has been an highly incremental process.

From BBN. They permitted me to use their facilities for producing final copy of the text of this work, and have been morally supportive in the final push to get finished.

From all my relatives, who have given me roots; and particularly from my parents who have unflaggingly had faith in, and been supportive of, their firstborn; and from my sister, Mary, and cousin, Austin Hoyt, who live in Boston and have provided me with innumerable mornings on the court, afternoons at Cohasset, and evenings around the hearth.

From Marsha Baker who, in addition to many years of playing the role, unpaid, of my personal secretary, has been, through it all, a friend.

And finally, from all my many friends and playmates, particularly Marty Richardson and all the members of the Nutt House, who have not only pushed, nursed, cajoled and challenged me into "finishing the damn thing", but also have consistently provided me in the midst of all this high technology with the reminder and experience of other, more ancient realities.

This work was supported in part by the Advanced Research Agency of the Department of Defense under ARPA Order No. 433 monitored by ONR contract no. N00014-70-A-0362-0001, and in part by the National Science Foundation under Grant no. GJ34671, and in part by the International Business Machines Corporation, and in part (at Lincoln Laboratory) by the Advanced Research Projects Agency of the Department of Defense and the Department of the Air Force, and in part by Bolt Beranek and Newman Inc.

Note on Preparation of this Report.

From handwritten manuscripts, the "raw" text was entered into a TENEX system at Bolt Beranek and Newman Inc. (BBN). The File Transfer Protocol (FTP) on the ARPA Network was used to transmit the text to the TX-2 system at Lincoln Laboratory. Formatting information was added to the text using the scope editor (SE) and tablet editor (TE) subsystems. This resulted in acceptable input for the layout (LO, subsystem, which was used to produce formatted copy. Draft versions were printed on a Xerox Graphics Printer (called the LDX - Long Distance Xerox - at Lincoln). Changes were made to produce improved drafts using the same editing systems. The final draft was transmitted back to BBN again using the FTP on the ARPA Network, where it was typed on a Bedford Terminal. Finally, smooth curves in the figures were drawn by hand.

Table of Contents.

Chapter 1. Introduction.

Computers were originally conceived of as devices for carrying out calculations. However, as computers have evolved into computer systems, this image has changed: now, computer systems provide communities of users with the ability to store, manipulate, and interchange information.

This changing perception of computer systems has effected concomitant changes in the life styles of the communities which they serve.

One community of interest is that of computer programmers. With "on-line" storage, programs and data have come to be regarded as part of the computer system. With easy communication among programmers, it becomes possible to build on the work of others by using programs and data prepared by others in the creation of one's own programs and data. This saves much wasteful duplication of effort. It also makes possible the creation of truly large programs. Such programs are difficult to build without the (possibly inadvertent) contributions of many programmers. The complexity of big projects is likely to confound individuals, and the sheer size of big undertakings demands more time and effort than an individual has available. If every man had to build his own castle, there would be nothing but houses.

Unfortunately, in existing systems these savings have been only partially realized. This is due, in part[1], to what appear to be "merely"

---

1. There are other problems than those discussed in this work which must also be solved before the full potential of building on the work of others is achieved. In particular, precise means of specifying the behaviour of modules must be devised so that programmers can describe the modules which they create (see [Parnas, 72a], [Liskov, 74], and [Zilles, 75]). Also, ways of organizing and searching these descriptions must be provided so that programmers can discover what modules are available for use.

mechanical problems. It is these apparently simple "mechanical problems" which are the subject of the research reported in this document; the goal is to design a computer system supporting sophisticated programming in which these problems do not arise.

1.1. Modular Programming Systems.

The entities in a programming system which a programmer can use in the construction of other such entities will, in this report, be called modules. In many existing systems, modules are either pieces of program or pieces of data. In general, however, modules are (possibly complex) contructions, comprising both programs and data, which can be used to carry out some computation, or achieve some effect[2].

Ideally, a programmer constructing a new module using existing modules should only need to understand what those existing modules do; he should not need to know how they are constructed. That is, the programmer would like to treat existing modules as "black boxes". The term "module" has been chosen to convey this ideal. The ideal of being able to program with black boxes is called modular programming". These terms are intended to evoke the mental images of physical and electronic modules (eg. children's toys, cinder blocks, floor tiles, integrated circuits, plug-in circuit boards) and of construction with them. These images suggest easy planning and construction according to simple rules.

At this point it is helpful to introduce some terminology for expressing certain relationships between and about modules. A new module created from existing ones is said to employ the existing modules; the existing modules are said to serve the new module. The existing and new modules are therefore spoken of as the serving and employing modules respectively. An employing module is said to invoke a serving module when

---

2. In keeping with this notion of modules, 'pure data' is conceived of as being the combination of some representation of the data and the programs which manipulate it (see [Liskov, 74]).

it requests service from it. The activity which is started by an invocation of a server and which is terminated when the service has been rendered is called the activation of the server for that invocation. Humans who invoke the services of modules in a computer system are called users of that system. Humans who create modules in a computer system are called programmers. The programmer or module that causes a new module to come into being is called that module's creator.

In this report, the behaviour of a module is used to mean all the information which is needed to describe the services which that module provides. This information is organized around some abstract notion of what the module does. For example, the module might compute the square of a number, sort a file, make a reservation on a train, or give the current Dow-Jones average. The behaviour includes information about the expected representation of the inputs and outputs, the relationship between these, and the changes if any which the module makes to the inputs. These aspects of behaviour will be called a module's input/output characteristics.

Sometimes, invoking one module can cause changes to the input/output characteristics of other modules. In particular, use of a module may affect its own input/output characteristics. For example, use of a random number generator affects it further use: pushing a value on a stack affects subsequent popping operations; and recording changes to stock prices affects the Dow-Jones average. Such interactions which affect a module's input/output characteristics may also be important to employers of that module, and are therefore considered to be part of that module's behaviour.

The subject of changing input/output characteristics will receive considerable attention later in this report. In preparing for that dicsussion, the following terms are introduced: an module whose input/output characteristics change with time is called mutable; otherwise it is called constant[3].

---

3. Notice that these terms must be understood relative to a set of input/output characteristics, which are in turn understood in terms of a programmer's conception of what a module does. Thus, a module may be constant with respect to one conceptualization of its activities and mutable with respect to another.

As just noted, the behaviour of a module is all the information about that module which is needed to make full use of it. There is a great deal more information concerning a module however. This includes how it is coded, what internal structure it uses, and what other modules it employs to achieve its affects. It is this other information that modularity demands need not be known by the employer of the module.

Intuitively, then, a modular programming system is one in which modules can be constructed employing existing modules using only knowledge of the behaviour of those modules. In this work, this intuition is embodied in the four requirements developed in the following paragraphs.

Firstly, a modular programming system must provide mechanisms for employing any module in the construction of any other. Also some means of invoking an employed module is necessary. Ideally, these mechanisms should be easy to understand and use. Also, the decision to employ one module in another should not place any restrictions on the internal structure of employing module beyond the requirement to supply the serving module with appropriate input. In particular, employing a module should not affect the freedom of choice of names used in the employing module to identify entities of any kind. A system which enjoys these properties will be called flexible.

Secondly, modularity will be taken to require that the employer of a module not have to worry about the other modules which that module needs in order to realize its behaviour. Modules must therefore arrange to supply themselves with all such needed modules. In a word, modules must be self-sufficient.

Thirdly, as any module may be employed in creating a new one, any set of modules will be employed together in the service of some other module. In a modular programming system, this ought never to lead to conflicts of any sort between members of the set. If a set of modules which can be employed together is called "compatible"[4], the third requirement of

---

4. The term compatible has sometimes been used to imply that modules agree on the format of data. That is a stronger condition than is intended here, for to demand that every set of modules have reached such an agreement is

modularity can be stated: every set of modules must be compatible.

Fourthly, in many systems, modules can be passed as arguments to other modules. In such systems, it may be difficult or impossible to predict which modules will employ and invoke a given module. Modularity will therefore be taken to require that modules exhibit the same behaviour to all employing modules. In a word, modules must be non-discriminatory[5]. Modules which fail this test are, of course, discriminatory.

1.2. Goals of this Research.

The goal of the work reported here has been to designa system which is "ideal" for carrying out the modular construction of programs. Such a system should satisfy the four requirements of modularity discussed in the previous section. In addition, four other requirements are adopted to embody the notion of "ideal".

Firstly, the system should present the programmer with a language for creating modules which is adequate for expressing common constructs of current programming languages and operating systems. These constructs include means for logical and arithmetic calculations, arbitrary structuring of data including self-reference, creation and manipulation of mutable modules and data, provision for controlling access to modules and data.

Secondly, the system must make provision for the creation and co-ordination of multiple processes.

---

to demand representation-independence of all data in the system. Although this work demonstrates one way to achieve representation-independence in data, representation-independence is not regarded as a criterion for modularity.

5. Non-discriminatory behaviour should not be construed to imply constant behaviour. However all employers of a mutable module should see the same time-varying input/output characteristics.

Thirdly, the system should support definitions of certain classes of specially behaved modules: In deciding to employ a module, a programmer is presumed to understand that module's behaviour. Of special interest because of their simplicity are modules whose input-output mappings are time-invariant; that is, given similar inputs, they always produce similar outputs. An ideal modular programming system, therefore, should support a definition of a module's behaviour being repeatable which captures this notion of time-invariance.

Furthermore, a programmer would wish to have a test which can be applied to a module to determine whether it satisfies the definition. However it is too much to hope that any reasonable definition of repeatability would be decidable. So, instead, it is demanded that there be forms which guarantee repeatability; that is, modules constructed according to those forms are known to be repeatable.

There is another sort of behaviour which is also special. Certain modules, while not repeatable, appear to be repeatable when viewed as mappings on sequences; that is, given similar sequences of inputs, they always produce similar sequences of outputs. So, it is also demanded that there be a definition of a module's behaviour being sequence-repeatable which captures this intuition. Also there should exist forms which guarantee that modules exhibiting those forms are sequence-repeatable.

Fourthly, the system should be implementable with reasonable extensions of current hardware and software.

There are other requirements which could also be adopted for defining what makes a modular programming system "ideal", but only at the price of overly extending the scope of this work. While no attempt is made here to satisfy these additional requirements, some care is taken not to preclude their solution. They are left to be solved as additions to the system presented in this work. They include providing mechanisms within the system for creating and using descriptions of the behaviour of modules, for handling errors, and for supporting input and output.

1.3. Problems in Achieving Modularity.

Before reviewing existing programming systems, it is convenient to explore some of the problems encountered in achieving modularity. This will provide some concepts and terminology which will aid in the analysis of those systems.

There are two ways to arrange for one module to employ another: a copy of the serving module can be created and included in the employing module, or a name for the serving module may be included in the employing module.

In the first of these schemes, a module can to physically enclose copies of every module which it employs. The scheme is inadequate, however, because it does not permit two modules to both employ some other module which is Consider, for example, a module which yields the current Dow-Jones average. Assume that this module uses some data base of current stock prices. Assume also that there is another module which makes changes to this data base to keep it current. Both modules must employ the data base. Under the copying scheme, therefore, each module must include a copy of the data base. Then, however, changes to one copy will not be seen in the other copy. So the desired behaviour is not achievable. Thus the copying scheme does not satisfy the flexibility requirement of modularity, for it fails to provide a mechanism by which a mutable module can be employed by others.

The other scheme is to include a *name* for a serving module in an employing module. The paradigm for the use of names is as follows: A name is an element of a possibly-infinite set called a *namespace*. A *context* is a partial function from a namespace into modules of the system. To employ a serving module, a name is chosen, a context is created mapping that name into that serving module, and the name and context are both included in the employing module. The server is located by applying the context to the name. Arranging that a context map a name into a module is called *binding that name to that module in that context*. Using a context to locate a module from a name is called *resolving* that name in that context.

There are potential problems with using names to indicate modules. For one, contexts are often mutable. For example, directories often serve as contexts, and often directories permit the mapping for names to be deleted or changed. Employing one module in another by using a name and a mutable context often makes it impossible to insure that when the time comes to use that name and context the desired module will be accessed. Any scheme for accessing modules for which such uncertainty exists will be called unreliable. So mutable contexts often lead to unreliable naming schemes.

Another source of problems is the inclusion of an abbreviated form of a name in a module. If the full name cannot be reconstructed with certainty from the abbreviation, the possibility of incorrectly resolving the abbreviation arises.

Another problem arises from the need to include contexts in modules so that they may be used for resolving names used in the module. If contexts are themselves modules, then the loop closes: to employ a module, it is necessary to name it and then employ another module, the context, to resolve that name. Of course, the context can then be named. This requires employing yet another context. Ultimately this loop can be stopped by using a name which has a binding in a context which is assumed by the system. Resolving this name is undertaken by the system, so no mention of the assumed context need be made. For example, in many systems the root of the directory system is an assumed context for file names. Another example is the context associated with each process in a system for resolving the addresses of words in memory; this context is called the process's address space, and in a paged system is represented by a page table. That this context is assumed is demonstrated by the fact that a process usually has no need to access its page table.

A problem arises if the assumed context associated with a particular name in a particular module is incorrectly supplied by the system. This can happen if the system's mechanisms for selecting the assumed context for that name in that module is dependent upon anything other than that module and that name. For example, file names in some systems are resolved relative to the "current working directory"; often the method of selecting

the working directory is dependent upon information other than the filename to be resolved and the module containing it.

Another way to break the loop of naming contexts is to copy them and include copies in modules which use them. This will not work if contexts are mutable.

Another problem involving a loop can arise in implementing contexts, for contexts must somehow indicate modules. As modules cannot in general be copied, either each module must appear in at most one context or contexts must also use names to indicate modules. The first solution is quite acceptable. The second solution introduces another loop: the names used by a context to indicate modules must themselves be resolved in another context. In this case the first solution can be used to break the loop introduced by the second solution. Alternatively, a scheme of assumed contexts can be used. As before, this solution requires that care must be taken to insure that the assumed contexts are properly supplied.

The problem commonly known as conflict of names occurs when two or more programmers try to bind the same name in a single context. This can happen either before or after the name has been used in modules to indicate the modules being bound. If the conflict is discovered before the name is used, the programmer can then choose a new name. The only problem with this solution is the nuisance of choosing alternate names.

Discovering name conflict after uses have been made of the name is a more serious problem for it requires changing those uses. Making such changes may not be easy. A case in point can arise when an attempt is made to combine contexts. For example, putting two sets of programs together often produces conflicts of subprogram names. The contexts associating names to programs are being combined into a single context. When conflict arises, the programmers are not necessarily still available to locate and change the uses of the conflicting names.

These various problems arising out of the use of names lead to violations of the requirements of modularity. An attempt to employ a module in another by using a name whose resolution to the serving module is

not guaranteed must be regarded as a failure of the mechanism to employ the module. If there is any module in a system which cannot reliably be employed by some other module then the system must be deemed non-modular. Name conflicts often lead to incompatibility of sets of modules. Invoker-dependent resolution of names can produce discrimination unless the behaviour of modules includes a description of how names are resolved.

Violation of compatibility can also come from resource limitations. A common example of this is a limitation on the number of names that can be resolved by a single context. In particular, the limited size of "address space" of "core-image" systems often restricts what modules can be employed together in forming a new module.

Discrimination may occur in systems in which access to modules is dependent upon access rights. To avoid discrimination, this dependence must be described in each module's behaviour, an often difficult task for complex modules. For example, any module able to employ a directory system should be equally treated. However that treatment may include requiring that modules explicitly present proof of access rights.

If, to employ a module, a programmer must be concerned with modules which it in turn employs (other than those explicitly described in its behaviour), that module is not self-sufficient.

1.4. Existing Systems.

This section discusses some representative existing systems to see which (if any) of the problems of the previous section appear. Two types of system are examined: systems which support a single programming language, and operating systems with their (possibly many) supporting languages.

FORTRAN language systems are typical of many language systems. In them, subprograms play the part of modules. Each subprogram is given a name by its creator. When a set of subprograms is put together (an activity known as "loading"), a single context is created associating

subprograms with their names. Uses of names in the subprograms of the set are then resolved in this context.

Each program is included in the set individually. Therefore the creator of the set must arrange that the modules serving an included module are also included in the set. Modules in these systems are therefore not self-sufficient. (The solution of having a "linking loader" is discussed later in this section.)

Because a single context is used for all subprograms loaded together, two subprograms having the same name are incompatible. The common manifestation of this incompatibility is name conflicts discovered when two collections of subprograms, independently conceived and created, are brought together to serve some new program.

The act of loading subprograms involves making copies of them. As discussed in the previous section, this precludes employing mutable modules or data.

Also, the act of loading a set of subprograms does not create another subprogram. Rather, it creates a program which is not acceptable input to a further loading operation. This violates the system's flexibility, for a program cannot be employed by another program.

Some FORTRAN systems do not strictly follow the FORTRAN standard upon which the preceding discussion has been based. While still using creator-assigned names, they use other mechanisms for name resolution. These mechanisms are typically those of the operating system by which the FORTRAN system is supported. The analysis of the modularity of these FORTRAN systems is therefore dependent upon the mechanisms for name resolution in the supporting operating systems.

APL language systems (see [Falkoff, 68]) differ from the FORTRAN pattern. The role of module is played in APL systems by the APL function. Where FORTRAN creates a single context associated with each program for resolving names in all subprograms of that program, APL systems give each programmer a single context for resolving all the names used in all his functions. This single context is called the programmer's "workspace".

Where FORTRAN loads subprograms when creating programs, APL functionals are loaded into a workspace when they are created, or when they are copied from the workspace of another programmer.

The same problems arise in APL as do in FORTRAN: Name conflicts lead to incompatibility. Requiring the programmer to supply serving modules leads to violation of self-sufficiency. Copying modules from other workspaces precludes employing mutable modules.

APL has additional problems. For one, the names in a workspace may be re-bound. This leads to unreliable name resolutions. For another, modules in APL are often discriminatory. A function can create and name data which is treated as local to an activation. These names are temporarily added to the workspace for the duration of the activation. If another function is invoked during an activation which creates local data, use in that function of the names for the local data will resolve to that data. The behaviour of a function making use of names in this way will depend upon what local data has been created by the invoking function, or its invoker, and so forth. This "call chain name resolution" leads to discrimination.

LISP systems (see [McCarthy, 62]) are similar to APL systems in many ways. Each programmer has a single context for many LISP functions: the space of "atoms". Functions of other programmers must be copied into the atom space of an employing function to be located by the name resolution mechanism. The bindings of functions to atoms may be changed. Call chain name resolution is usual.

However, LISP has another naming mechanism which can be used to eliminate these problems within the limited scope of a single programmer's set of functions. The atoms, functions, and data of a single programmer are all represented as objects called "list elements". When a list element is created, it is bound to a "list address" in a single (per programmer) context called the "list addressing mechanism". (The implementation of this mechanism varies between operating systems. It usually is built on operating system main-memory addressing mechanisms and a garbage collector or compactor.) List addresses cannot be re-bound in the list addressing mechanism. List addresses can be used in list elements using the list

addressing mechanism as assumed context to achieve reliable references to (other) list elements.

The "FUNARG" mechanism in LISP (see [Moses, 70]) permits creation of objects called "closures". A closure comprises a function and a context which is called the "environment" of that closure. Closures are also modules of the system. When a closure is invoked, it resolves names appearing in its function by using its environment. The modules and data having bindings in the module's environment are named using list addresses. List addresses are also used in the closure to name the function and environment.

In many LISP systems the size of the name space of list addresses is small enough that it can be exhausted relatively quickly by even the modules of a single applications program. Thus potential[6] sets of closures can be incompatible because they would together exhaust the list address space.

As far as name conflicts are concerned, however, two closures are always compatible. With care, closures can be made self-sufficient. Because closures avoid call-chain name resolution, they are non-discriminatory. So within the confines of a single user's functions and data, LISP permits modular programming through exclusive, careful use of closures[7].

Most language systems, including those just discussed, have been designed to aid the single programmer in creating his own programs in isolation. It is only secondarily that they have concerned themselves with interactions between programmers in the creation of programs. A common

---

6. Because closures are created within the list address space, any set of existing closures by their very existence demonstrate that they do not exhaust the list address space. For this reason, the incompatibility of larger sets of closures manifests itself in exhaustion of the list address space when trying to create one of the closures of that set.

7. This particular discipline is not a common one among LISP programmers, however.

form of response to this concern is to create a "library system" for the language. Libraries may be regarded as contexts which map names into existing modules. When creating a program, copies of these modules are loaded from libraries in response to uses in other modules of the names appearing in the libraries. That loading implies copying precludes employing mutable modules. Often names are resolved into modules by searching a collection of libraries according to some set of conventions. If names are bound in more than one of these libraries, uncertain name resolution can result. Sometimes giving priority to the libraries is used to solve this problem. It often is difficult, and in the worst case is impossible, to choose a set of priorities which will insure the desired name resolution[8]. On top of this, names can usually be re-bound in libraries, which leads to even more uncertainty about name resolution.

The other form of response in languages to the need for interaction among programmers is to fall back on the name resolution mechanisms of the operating system by which the language system is supported. This leads to a discussion of the second type of programming system currently in existence: operating systems together with their supporting languages. In such systems, modules are often called "files". Names used in modules are resolved in a single system-wide context called the "file system". The names used to indicate files are consequently called "file-names". In some of these systems, it is not necessary to copy a module in order to use it; in these systems, mutable modules can be successfully employed by more than one module.

However, because all programmers use the same file system, conflict over the use of file-names can occur. Therefore it is common (for example, in CTSS [Crisman, 65], and APEX [Forgie, 65]) to partition the space of file-names giving part to each programmer. This is often done by assigning unique names to programmers and requiring that the first part of each file-name be the name of the programmer choosing that file-name.

---

8. See [Clingen, 69] for a detailed analysis of the related problem arising in the Multics system of resolving, at run time, names used in loaded and running modules.

For the convenience of a programmer creating a collection of related modules, file-names appearing in modules in the collection and indicating modules within the collection are often allowed to be abbreviated by omitting the programmer's name (for example, in CTSS [Crisman, 65], and APEX [Forgie, 65]). This requires a concomitent sophistication of the name resolution mechanisms of the file system.

Such abbreviation schemes, although convenient, must be used with care however. For example, if an abbreviated name is passed as a parameter to a module created by another programmer, the name resolution mechanisms of the file system will incorrectly extend it when generating the full name of the desired module. The problem in extending abbreviated names is a common source of problems in achieving reliable naming schemes.

As a programmer uses names in his partition of the namespace of file-names, he may eventually find that he has already used all the mnemonically satisfying names. This often leads to further subdivision and structuring of the namespace of file-names, supported by additional namespaces to name the partitions (for example, in Multics [Organick, 72]). Permitting more sophisticated abbreviations then leads to more sophisticated mechanisms for extending those abbreviations into full file-names (see [Clingen, 69]). This in turn leads to even more difficulty in guaranteeing reliable naming.

Many systems permit re-binding of a file-name in the file system, particularly when the programmer knows that the module currently bound to that name is no longer in use. However, one result of employing the modules of others is that the creator of a module may have no idea of whether that module is still named by other modules in the system. Systems which do not police re-binding are common; in such systems, relying on file-names is unwise.

The preceding review makes it sound as though systems of the sorts mentioned have severe problems. In actual fact, there exist such systems which serve sizable communities and receive extensive daily use.

There are two factors which make this possible. Communities tend to adopt protocols and conventions for system usage which help programmers to avoid trouble. Programming aids are often created which support restricted classes of programming. Thus sub-systems can support modularity in systems which do not support modularity very well by restricting and controlling programmers' activities. However, building such well-behaved sub-systems in non-supportive systems is often difficult.

The second mitigating factor is that, except at the command and language-library levels, very limited use is currently being made of modules written by other people. Thus the problems are not encountered as frequently as they would be if the practice of modules employing other modules were common.

Despite these mitigating factors, however, the violations of modularity do cause troubles in existing systems. That this is so is attested to by the feeling often expressed by programmers that "code rots", and by the smiles invoked by suggesting that one should rely on a module which has not been used for an extended period of time.

Thus, faced with solving the problems of employing the modules of others in systems which support modularity badly, programmers can perhaps be forgiven for feeling that "it's easier to write it myself". This attitude leads to an unfortunate vicious circle: Programmers avoid employing existing modules because the effort to do so in existing systems appears not worth their time. At the same time, system designers do not examine and remove the sources of these difficulties because programmers appear to get along pretty well the way things are.

The objective of the work reported here is to break into this circle by designing a system which is ideal for carrying out modular construction of programs.

1.5. Related Work.

The previous section has argued that existing systems do not meet the requirements set out for an ideal modular programming system. This failure is due primarily to the fact that these requirements have not been included among the design criteria for these systems. In turn, this exclusion results from the fact that these requirements have not been clearly stated until recently, as is shown by the following history of work on modularity and the design of systems demonstrating it.

Dijkstra [Dijkstra, 65a] stated "The principle of non-interference": the function of a program can be established from the external specification of the parts each considered independently of one another and of the further context in which they are used. Boebart [Boebart, 65] discussed the concept of modules as "conceptual entities" existing within a system, rather than as "partially complete programs". standing outside the system standing. He demanded that employment of a module not require that an individual "comprehend the internals of the module". He proposed a system in which modules were connected by queues, the resulting assemblages then becoming modules. The details of sharing, copying, and naming of modules, particularly data, were not made clear however.

Dennis [Dennis, 68] carried the ideas of Boebart further, stating some more precise requirements for modularity (called "programming generality" in this work). These requirements included the ability to "transmit elaborate information structures as arguments", and the ability of a module to call on modules unknown to the caller. The former ability is a requirement that the system be free of one of the forms of incompatibility between modules: the exhaustion of storage space. The second ability is a request for self-sufficiency. As with Boebart's work, the issues of program creation and module naming were not addressed.

Dennis [Dennis, 71] further developed the concept of modularity by defining it as "the ability of users to construct programs by linking together subprograms without knowledge of their internal operation". This work also reviewed the problems encountered by systems in using names, and

concluded that no good solutions to these problems was then known.

Dennis [Dennis, 72] also added to the notion of modularity the idea that the behavior (he said "correctness") of a module should be establishable without recourse to knowledge of the context of its use in building other modules. These notes went on to give a careful and detailed analysis of modularity in language systems for FORTRAN and ALGOL 60, and in the MULTICS operating system. It also proposed a system design (The Common Base language) which was much more thorough than the others mentioned thus far. It was based on a graph model of memory. While the details of its design are not complete, it has served as a basis for study of the semantics of programming languages (see, for example, [Amersinghe, 72] and [Ellis, 74]).

Vanderbilt [Vanderbilt, 69] developed a model in which nodes in the graph are differentiated by markings which indicate how they may be used (read, write, execute). Controlled sharing was achieved by adding access rights to the branches of the graph, and by viewing the processes of the system as not being included as part of the graph. This system is limited by the acyclic restriction on its graphs.

The models developed in these earlier studies were not designed to support studies of the special behaviours, and were not well-suited to doing so: Firstly, they were not completely enough specified, particularly in the aspect of program creation. Secondly, in these models no differentiation was made among the nodes in the graphs to indicate their intended use by the programs. As work reported here <u>is</u> concerned with the special behaviours, the model developed here differs from earlier models by being quite specific about program creation and by differentiating among the objects of the system on the basis of their intended use. It must be noted, however, that the earlier studies, while not meeting all the requirements of this research have been instrumental in providing ideas to the system presented here. These contributions will be discussed further in Chapter 8.

Another source of ideas upon which this work is based is the studies naming schemes in programming languages. The description of all languages

must express how names are used and resolved. Some language descriptions (for example, BASEL [Hammer, 69], and ALGOL 68 [Van Wijngaarden, 69]) have developed models and supporting terminology to aid in making these descriptions clear. The Contour Model [Johnson, 71] is designed primarily to make intuitive and explicit one particular class of scoping rules (including those for ALGOL 60). More generally applicable is the Cellular Storage Model [Lomet, 73a; Lomet, 73b]. It is based on a set of operators which act on names and memory structures to define the resolution of those names. Other languages (for example, VERS2 [Earley, 73a; Earley, 73b], EL1 [Wegbreit, 74]) have provided very powerful facilities for allowing the programmer to control the resolution of names within his programs.

Despite this extensive study of the use of names in programming languages, naming mechanisms in operating systems have received only limited attention. While the work in programming languages has escalated the programmer's expectations for programming expressive power, and has improved the understanding of name resolution mechanisms in programming languages, corresponding benefits for operating systems have not been sought. It is one of the purposes of this research to do so.

There is one further area of thought that deserves mention, primarily in order to avoid confusion over terminology and intent. The term "modularization" is often used to discuss the segmentation of a large program into smaller, more understandable parts (see especially [Parnas, 72b]). The distinction between segmentation and modularity should be noted: When segmenting a program, the intended usage of a module is well known. The concern of modularity is that modules be usable in places where their use has not been forseen at all. Of course, having a modular system will aid in the (re)assembly process of a segmented program, and to that extent it may be some aid to the segmentation process. Also, as pointed out by Rhodes [Rhodes, 73] in a review article on segmentation, segmentation leads to the need for modularity, because "the ultimate design objective is to build up a library of re-usable modules".

1.6. Preview.

The goal of the work reported here is to design an ideal modular programming system. In fact, the work defines a whole class of programming systems. The design of this class is embodied in the design of an abstract machine called the "Binding Model" which acts as a kernel (in the sense of [Wulf, 74]) for members of this class. One of these members is produced by writing an "initial program" for the model. Initial programs must provide certain operating system functions, such as creating the initial directory structure, creating the mechanisms for adding and deleting knowledge about programmers, and performing "logins" and "logouts". The abstract machine provides the primitive mechanisms from which these non-primitive facilities can be programmed, such as mechanisms for doing logic and arithmetic, for grouping, for providing for mutation, for interpreting data structures as programs, for name resolution, for recursion, for protection, and for multi-processing.

Chapter 2 introduces and informally describes a version of the Binding Model which has only a single process. Chapter 3 gives examples of the behaviour of this version to increase the reader's familiarity with the model and its concepts and to demonstrate how it supports some common programming constructions. Chapter 4 discusses a formal definition (given in the Vienna Definition Language [Lucas, 68]) of this version of the model. This definition appears in Appendices B and C. Chapter 5 discusses the definition of the special behaviours for this version of the model and some forms defining modules which satisfy them. Chapter 6 introduces a version of the model which permits the creation and co-ordination of processes. It includes an informal description of this extended model, some examples of its use including the design of one initial program for the extended model which demonstrates its ability to support a concept of "programmer", a description of the extensions necessary to the formal definition (also included in Appendices B and C), and discussion of how the

extension affects the definitions and forms for the special behaviours. Chapter 7 comprises thoughts on implementing the model. Chapter 8 reviews the model in light of the goals and related work discussed in this chapter. It also lists some topics which appear worthy of further study.

Chapter 2. Informal Description.

This chapter introduces the Binding Model. Included are all the ideas upon which the Binding Model is based except those involving the creation and co-ordination of processes; these appear in Chapter 6. This chapter is an informal but thorough presentation, designed to develop intuition. The next chapter furthers that development with some examples. Chapter 4 describes the formal definition which appears in Appendices B and C.

The Binding Model (called "the model" hereafter) is an abstract machine of sophistication comparable to that of the abstract machine implemented by an operating system. It would therefore probably be implemented in a combination of hardware and software. Implementation is discussed in Chapter 7.

2.1. States and Transitions.

The Binding Model can be thought of informally as a single active entity, the process, making changes to a finite collection of passive entities, the items. Each item can be thought of as a single piece of data. Together, all the items can be visualized as forming a directed graph with a finite number of nodes. Each item is a node of the graph together with a (possibly empty) collection of directed arcs emanating from it. These arcs terminate on the node portions of other items. Because of the importance of this derived graph, the collection of items is spoken of as the graph of items. Figure 2.1.1 depicts this schematically.

In addition to its arcs, each item of the graph has some further information. This information will be discussed in detail in the following sections of this chapter. However, it is helpful to anticipate that discussion here by noting that this information may include what can be

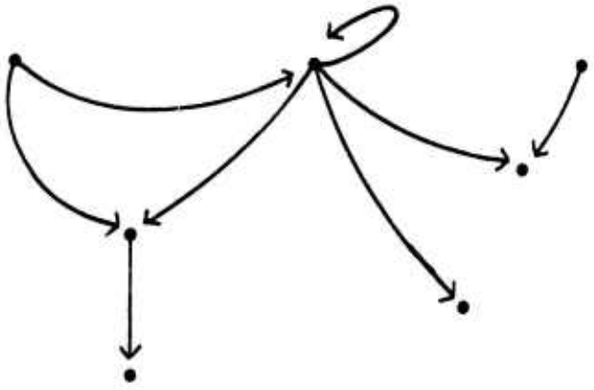


Figure 2.1.1. Schematic depiction of a graph of seven items.

informally thought of as "a machine language program for the Binding Model". Information of this kind is called a control structure. It is a tree of control directives which correspond very roughly to individual "machine instructions". A set of names, called identifiers, are used in these instructions to name the items in the graph which the instructions are to use as arguments. By carrying out these instructions, the process may effect two kinds of change in the graph: it may add an item (a node and some arcs) to the graph, or in certain special cases it may change the termination point of an arc in the graph. There are no changes which contract the graph; items, once created, are viewed as existing forever.

Items which contain control structures in their information are called functionals They play the role of "module" in the Binding Model.

The process is composed of a (LIFO) stack of activations, each of which is the record of the partial completion of the invocation of a functional. At any given time, the process is actually "running" in only the most recent of these activations. Each activation is composed of two parts. The first of these is a special arc indicating some control directive in some control structure. It is called the activation's control point, and can be thought of as the process's "program counter". The control point of an activation always remains within the same control structure; this control structure is therefore referred to as the control structure of the activation. The other part of an activation is a context which maps identifiers encountered in the control structure of the activation into

items in the graph. This context is called the environment of the activation.

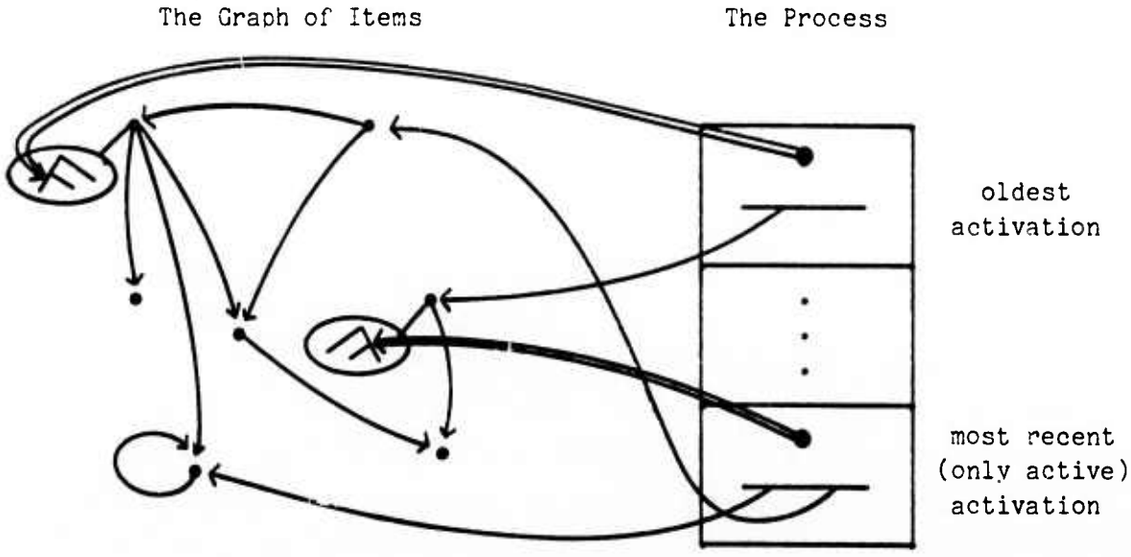


Figure 2.1.2. Schematic depiction of the graph and the process.

All of this is depicted schematically in Figure 2.1.1. In this figure, the process appears as a line of adjacent boxes, each of which depicts an activation. The most recent activation appears furthest down the page. The ovals depict control structures; that they are part of the items to which they are connected is suggested by the fact that the connections are not made with directed arcs. Doubled arcs depict control points. environments appear as horizontal lines.

At any given moment, a snapshot including the graph of items, the process, and their interconnection would capture the complete status of the model. Consequently such snapshots are called states of the model. To define the model more formally, it is convenient to describe the actions of the model in terms of "before" and "after" states: that is, in terms of state transitions. The activity of the model is defined by a state transition rule which defines for every state a successor state. The model starts in one of a collection of possible initial states. Transitions are

effected by repeatedly moving from the current state of the model to its successor state. Transitions continue until one of a collection of final states is encountered.

The state transition rule can be regarded as defining successor states to be the result of the process's taking the action indicated in the control structure by the control directive pointed at by the control point of the "current" state. In moving from state to state, the process is said to be evaluating[1] the control structure.

There are many possible initial states of the model. The graph of items in an initial (and indeed, in every) state of the model has only a finite number of items in it. In an initial state, the process has only one activation. The control point of this activation indicates a control directive which is the root of the tree of control directives constituting some control structure. Together the control structure and the environment of the single activation of the process in an initial state of the model are called the process's task. As just noted, no evaluation has taken place on the task of a process before the initial state. A final state of the model is either one in which, by interpretation, the task of the process has been completely evaluated, or the distinguished state called error which indicates that the processor has encountered a request for an ill-defined action during its attempt to evaluate its task.

The next few sections give the static characteristics of the graph of items. The primitive operations whose evaluation changes this graph are then discussed. Thereafter, the way in which these primitive operations are used in control structures is presented. The chapter finishes with discussions of generating cyclic paths of arcs in the graph of items, and a way to represent control structures with items.

---

1. The term "evaluate" as used in this report is not intended to imply the absence of "side effects". That is, the term is used not in the purely mathematical sense, but rather in the sense of "evaluating the arguments to a procedure".

2.2. The Graph of Items.

The previous section introduced the graph of items. Before proceding with further discussion of the individual items composing the graph, it is helpful to introduce some terminology. For clarity, a special term is used to indicate the test for equality of nodes (items) in the graph. Nodes (items) are spoken of as being identical if they are the same node (item) in the graph.

If one item has an arc emanating from it which terminates on another item, the first item is said to directly enclose the second. If a path of arcs leads from one item to another, the first is said to enclose the second; for later simplicity, it is convenient to extend "enclose" to say that every item encloses itself; that is the paths determining enclosure are permitted to have zero or more arcs in them.

As noted in the previous section, every item in the model has associated with it some information which defines its activity in the model. (This information includes all the item's arcs.) It called the item's information content. This section describes the nature of the information content of items. To aid understanding, this information will be expressed pictorially by giving depictions for the node parts of items that are more detailed than the dots which appeared in the figures of the previous section.

There are three dichotomies which are of interest in describing items: elementary/non-elementary, structured/unstructured, and mutable/constant. An elementary item is one whose information content determines its identity; that is, there are never two elementary items in the graph which have the same information content. There are three kinds of elementary item in the model: the logicals, the integers, and the symbols. The rest of the items are, the non-elementaries, are of five kinds: the keys, the sheaves, the bundles, the cells, and the functionals.

A structured item is one which has (more than zero) arcs emanating from it in the graph. All elementary items are unstructured; so are the keys and certain of the sheaves, bundles, and functionals.

A mutable item is one whose information content may be changed by actions of the process. More specifically, the termination of arcs emanating from mutable items may be altered. Cells are the only mutable items.

The logical items provide for logical computations. As noted above, they are elementary (and therefore are also unstructured and immutable). There are only two of them; they are depicted by the two words true and false.

The integer items permit arithmetic computation. Each has information content consisting of the fact that it is an integer item and the fact of which (mathematical) integer it represents. Because the graph in any state is finite, there are in any state only a finite number of integer items. Because the integer items are elementary, no two of them will ever represent the same (mathematical) integer. Integers items (hereafter often referred to just as "integers") are depicted by their decimal representations.

Symbols are used in the model for naming and for text manipulation. Each symbol may be thought of as a "string of characters". They are elementary, and are depicted as quoted(") sequences of characters.

Examples of depictions of these elementary items are given in Figure 2.2.1. As noted above, elementary items with the same information content are identical. However, reflecting this property of elementary items while depicting collections of items can lead to unpleasant complexity in those depictions (great "snarls" of arcs). Consequently the following convention is adopted: duplicate depictions of an elementary item are permitted, but they are to be understood as all representing a single node in the graph.

Keys provide for identity which cannot be counterfeited. They are unstructured and immutable. The information content of a key consists solely of the information that it is a key. Thus the information content of any two keys is the same. As keys are non-elementary, there can be many

true	-37	"here"
false	171	"there"

Figure 2.2.1. The depiction of some elementary items.

Figure 2.2.2. Depiction of a key.

of them, however. Keys are depicted as wedges[2]. An example is given in Figure 2.2.2.

The ability to group items together is provided in the model principally through use of items called sheaves. A sheaf can be thought of as a (possibly empty) collection of items each of which is named within the collection. The names used in these collections are elementary items and keys. Because these items are used to select items from the collections which are sheaves, the elementary items and keys are referred to collectively as the selectors.

The items included in the collection which is a sheaf are called the components of that sheaf. There are no restrictions on the kinds of items that can be components of a sheaf. Sheaves may therefore have sheaves as components. Thus arbitrarily large structures of items are permitted.

Sheaves are depicted as shown in Figure 2.2.3. If the sheaf has n components then there are n arrows emanating from the circle. An empty sheaf has no arrows, and therefore appears simply as a circle. Each of these arrows indicates the direct enclosure of a component; the direct enclosure of a selector for each component item is indicated by an arrow originating near the arrow for that component. (The information content for a sheaf is everything that is necessary to draw it: the identity of the selectors and components, and the correspondence between them.)

---

2. All non-elementary items will be depicted as closed shapes, possibly with additional related shapes and lines.

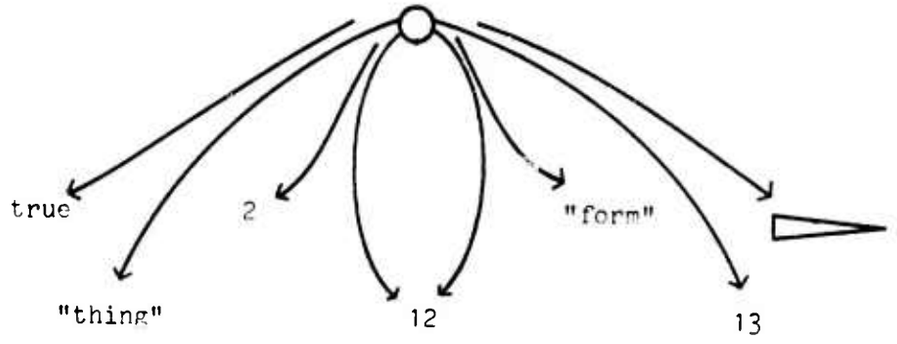


Figure 2.2.3. Depiction of a sheaf.

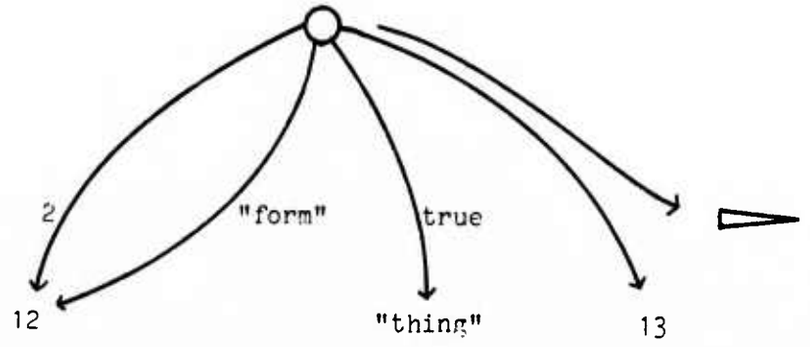


Figure 2.2.4. Abbreviated depiction of Figure 2.2.3.

Because selectors are usually elementary, a shorthand for direct enclosure of elementary items as selectors of sheaves is introduced: the arrow to an elementary selector can be omitted. Figure 2.2.4 depicts the sheaf of Figure 2.2.3 using this shorthand. Figures 2.2.3 and 2.2.4 also indicate that the ordering of components in the depiction of sheaves conveys no information.

Bundles are structured similarly to sheaves. However, unlike sheaves, bundles provide for controlled selection of their components. (Discussion of how this difference shows itself in the model must wait until the operations of sheaves and bundles are introduced in later sections.) For clarity, the selectors of bundles are called the bundle's extractors. Bundles are depicted just as sheaves are, except that a hexagon is used instead of a circle. See Figure 2.2.5.

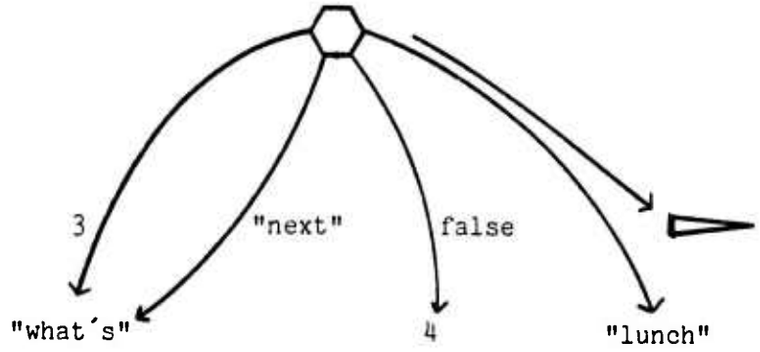


Figure 2.2.5. Depiction of a bundle.

Figure 2.2.6. Depiction of a cell.

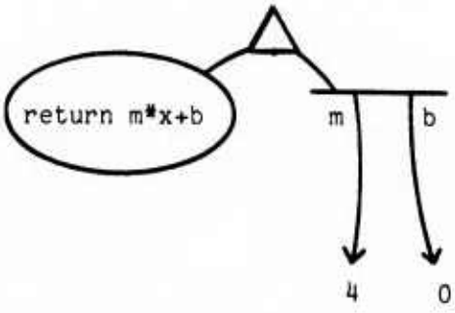


Figure 2.2.7. Depiction of a functional.

Cells provide for mutation in the model, as will be explained later. Structurally, a cell directly encloses some single item called the cell's contents. There are no limitations on what items may be the contents of a cell. In particular, a cell may be its own contents. Furthermore, two cells may have the same item as contents. Therefore a cell is not thought of as containing its contents in a physical sense. Cells, like all items, are abstract and are not viewed as having physical location. A cell is depicted as shown in Figure 2.2.6.

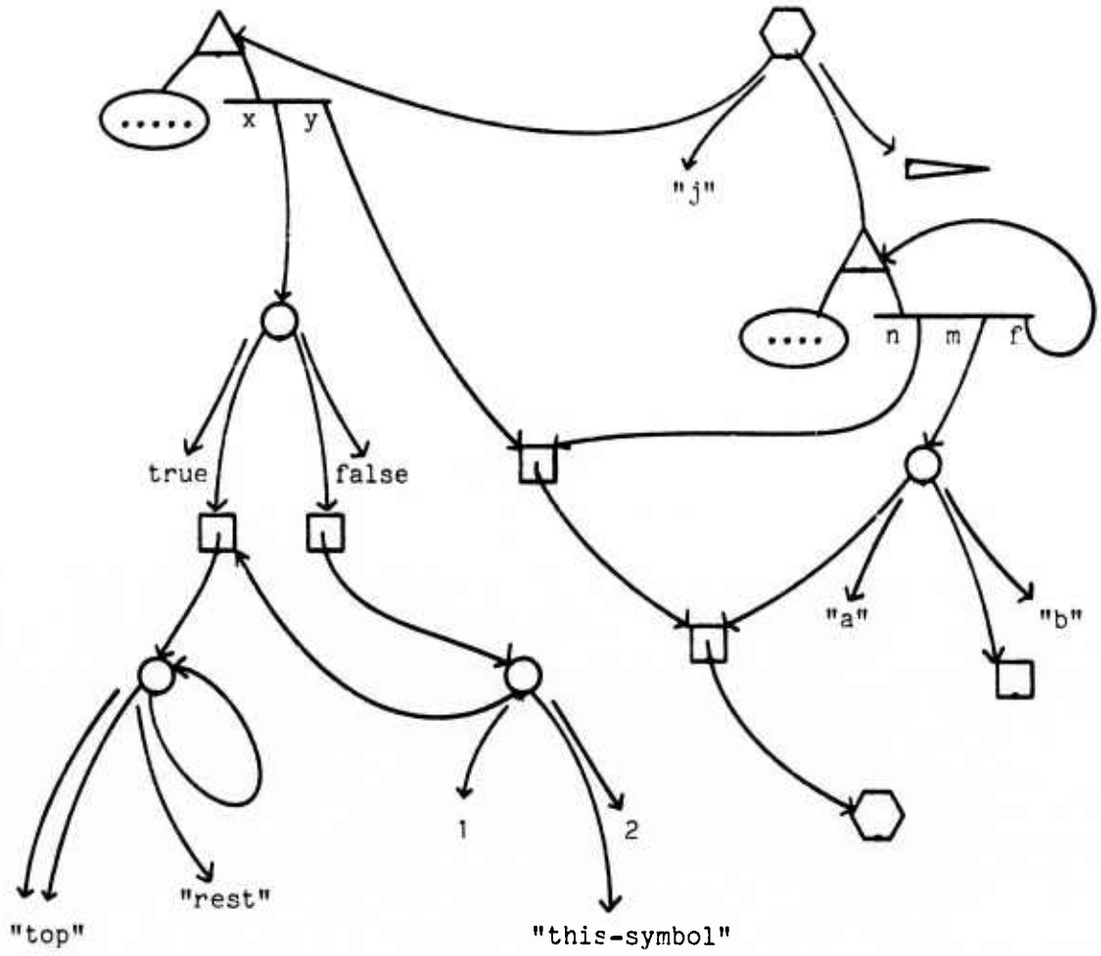


Figure 2.2.8. A collection of items.

Functionals provide the model with functions and procedures. As discussed in the previous section, each has a control structure which may be thought of as the "code" of the functional, and an environment which is a context used for resolving the identifiers which appear in the control structure. As in other contexts, the associations in environments of functionals between identifiers and items are called bindings. Not all identifiers appearing in the control structure of a functional need have bindings in the environment of the functional.

A functional is depicted as shown in Figure 2.2.7. The oval surrounds a linear representation of the control structure. The horizontal line depicts the environment. Each binding in the environment is depicted by an identifier appearing beside an arrow descending from the line. Notice that the identifier x has no binding in the environment of this functional.

The collection of items in a state of the model can be depicted using these pictures. Such depictions would emphasize the graph of dependencies of items upon one another. Figure 2.2.8 demonstrates this for a small collection of items.

2.3. References.

As Figure 2.2.8 indicates, it is possible to have an item which directly encloses itself, or which directly encloses some other item which in turn directly encloses it. Also an item may be directly enclosed by more than one item.

To support these complex patterns of enclosure, the model uses a namespace whose members, called references, may be thought of as the model's "internal" names for items. A single context called the reference context is used to resolve references when it is desired to access items from references. A reference bound to an item in the reference context is said to refer to that item.

When, at the request of the process, the model adds an item to the graph, a reference which has no binding in the reference context is chosen. This reference is then bound to the item in the reference context. The reference can then be used in other items which depend upon this item.

Bindings in the reference context are never altered or deleted. Including a reference in an item is therefore a reliable way to support the dependence of that item on the item referenced.

Because the reference context has bindings added but not deleted, it simply continues to grow. An exception to this pattern of continuous growth is discussed in Section 2.13. Implementation considerations are discussed in Chapter 7.

In the pictures introduced above, the use of an arrow can be interpreted as a depiction of a use of a reference and of its binding in the reference context. These pictures can be thought of as representing references as specifications of locations on the page (for example, as co-ordinates in 2-space), and as implementing the reference context by using the ability of a reader to find a location on the page from its specification. The arrow may be viewed as a shorthand indicating a specification of the location of that arrow's head present at the location of that arrow's tail.

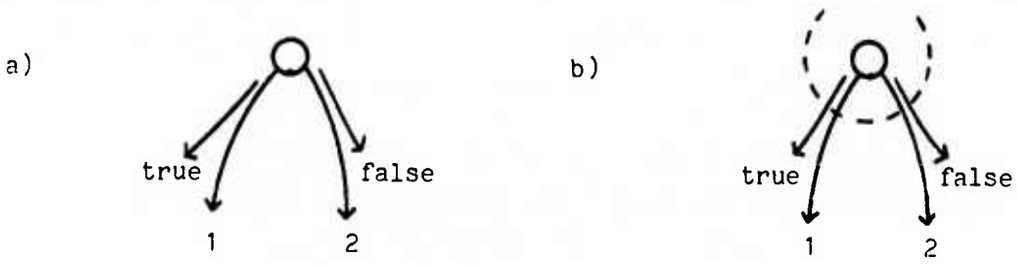


Figure 2.3.1. Re-interpretation of the depiction of a sheaf.

To make this re-interpretation exact, the depictions of sheaves, bundles and functionals must also be somewhat re-interpreted. Consider, for example, the sheaf of part a of Figure 2.3.1. The information content of this item can be broken into two parts. The first part is "shape": the

fact that the item is a sheaf, and that it has two components. In the modified redrawing of the item in part b, the shape is depicted by the portion inside the dotted line. This shape has four "holes". The four references which fill these holes are the second part of the information content of the sheaf. They are depicted in part b by the portion outside the dotted line: that is, by the four arrows.

2.4. Types and Primitive Operations.

The concept of "type" is an old but usually informal idea in programming languages and operating systems. Loosely speaking, objects of different types behave differently. Recently (see [Liskov, 74] and [Zilles, 75]), the notion has been formalized by characterizing a type by the set of operations that can be performed upon objects of that type.

In the model, each kind of item has a different set of operations which can be performed on items of that kind. Thus, each different kind of item defines a different type. Because these types describe entities — items — which are fundamental to the model, they are called the primitive types of the model. The operations which can be performed on items are called the primitive operations (PO's) of the model. That part of a control structure which specifies that a PO is to be used is called a PO invocation. In carrying out a PO invocation, the process is said to execute the PO.

All manipulations and changes to the graph of items are viewed as the result of invocations of the PO's. The various PO's require from 0 to 3 references to indicate which items in the graph they are to operate upon. References used in this way are called operands of PO's. The terms anadic, monadic, diadic and triadic are used to indicate the number of operands expected. Each PO yields a list of 0 or more results which are also references to items. Some PO's add items to the graph of items in the current state; others interrogate existing items; and one causes a change to be made to the dependency of an existing item upon other items. If the action of a PO adds an item to the state, and (a reference to) that item is returned, then that PO is said to have yielded a new item.

The next few sections describe the PO's of the model. Later sections explain how PO's are implanted in control structures, how their operands are determined, and what becomes of their results.

The PO's of the model each expect a fixed number of operands. In most cases, the PO's are defined only when their operands are of specific types or groups of types. It is therefore considered an error if an operand is not of the expected type(s). Mention of expected types in the following descriptions of PO's should be taken as indications that such "type errors" will arise when the operands are not of those expected types.

The detection of an error causes the model to select as its next state the distinguished final state error. No further activity takes place. It is left for further study to design an error-handling mechanism which permits further activity following the detection of errors.

log	logical
int	integer
sym	symbol
key	key
shf	sheaf
cel	cell
fnl	functional
bnd	bundle
sel	selector (log,int,sym,key)
any	any (log,int,sym,key,shf,cel,fnl,bnd)

Figure 2.4.1. Identifiers used to indicate item types in PO forms.

It will be helpful to have a linear notation to represent control structures of the model. Such a notation will be developed when new parts of control structures are introduced. The linear notation for an invocation of a PO has the form

PO-name{operand-1,operand-2,...}

where operand-1, etc. are identifiers which are resolved into references to items during evaluations of the PO invocation. The identifiers used in these forms will indicate the types expected (see Figure 2.4.1).

To illustrate the use of this notation, nine PO's which can never produce operand type errors are now introduced:

identical{any,any}
is-logical{any}
is-integer{any}
is-symbol{any}
is-key{any}
is-sheaf{any}
is-cell{any}
is-functional{any}
is-bundle{any} .

The first of these implements the equality test for nodes in the graph. It returns a reference to a logical item; a reference to true indicates that the two references which were the operands to identical both referred to the same node in the graph; a reference to false indicates that they did not.

The other PO's of this group implement tests for the types of items. They return references to logical items indicating whether or not the references which were their operands referred to items of the type sought.

It is tedious to keep mentioning that operands and results are references to items. Consequently, subsequent descriptions of PO's will speak of passing and returning items with the understanding that it is really references to those items that are being discussed.

2.5. Operations on Elementary Types.

There are three operations on logical items:

not(log)
and(log,log)
or(log,log) .

Not expects one operand, which is (a reference) to a logical item. If the operand refers to the item true, not yields the item false. If the

operand refers to the item false, not yields the item true. Thus the PO not implements the common logical function "not".

The PO's and and or implement the common logical functions "and" and "or". Each expects two operands and returns a single logical item.

As the logical items are elementary, there is at most one item true in any given state of the model. Suppose not is used with true as an operand. Then if the item false is not in the state it must be added in the next state; this new item is returned. If false is present, then no new item is added; the existing one is returned.

Four primitive operations on integers

add{int,int}
subtract{int,int}
multiply{int,int}
divide{int,int}

implement the operations of the ring of integers. The first three yield single results. The last yields two results: the quotient and the remainder of the division. As with the logical items, it is arranged that integers which are alike are kept identical.

There are two operations on symbols:

compose{shf}
decompose{sym} .

Both yield single results.

It is helpful at this point to single out a subset of the sheaves. A sheaf with n components is called a "row" if the n selectors of these components are the integer items 1 through n. Rows are expected in a number of places as, or enclosed in, the operands of the PO's.

Compose expects a row of symbols; it returns a single symbol which is the concatenation of all the symbols in the row. An error results from invoking compose with an operand which is not a row of symbols. Decompose expects a symbol; it returns a row of symbols each of which has only a single character. Symbols, like the other elementary items, when alike are identical.

2.6. Operations on Keys and Selectors.

Keys are included in the model to provide an unbounded set of distinguishable items. The only PO for keys is

new-key{} .

It yields a single result which is a new key. (That is, each evaluation of new-key yields an item which is not identical to any existing item.)

There are two PO's which apply to any pair of selectors (logicals, integers, symbols, and keys):

equal{sel,sel}

less-than{sel,sel} .

Each yields a single logical item. Equal implements the test for identical. Less-than provides an ordering for all the selectors. This ordering is first by type: logical < integer < symbol < key. Within each type the orderings are: false < true; the usual ordering of integers; lexicographical ordering on symbols using some assumed ordering of characters (ASCII character codes, for example); order of creation of keys. The next section indicates why such a general ordering relation is provided.

2.7. Operations on Sheaves and Bundles.

The PO

empty-sheaf{}

yields a new empty sheaf.

The PO augment is triadic:

augment{shf,sel,any} .

It returns a new sheaf which has as components all the components of its first operand and in addition the third operand. The selectors of these components are the selectors of the first operand, and the second operand. Use of augment does not alter its first operand in any way. Figure 2.7.1

gives an example of the action of augment[3].

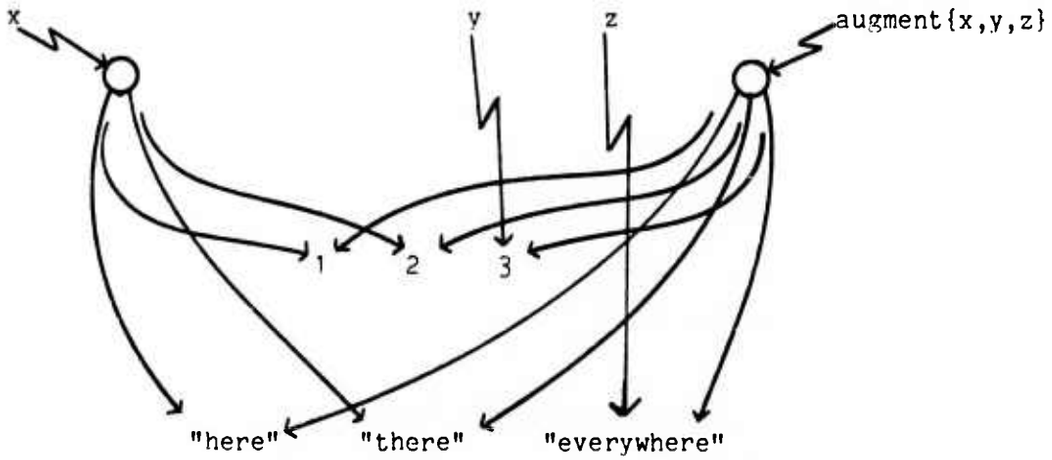


Figure 2.7.1. Action of the PO augment.

Two PO's allow accessing of the components of a sheaf:

select{shf,sel}

is-selector{shf,sel} .

Select returns the component associated in that sheaf with its second operand. It is considered an error if the second operand is not a selector of the sheaf. The PO is-selector is provided to protect against such errors. It permits testing for the presence of a particular selector item as a selector in a particular sheaf. It yields a logical item.

Two PO's are also provided for enumerating all the selectors of a sheaf:

length{shf}

selector{shf,int} .

Length yields an integer representing the number of components in its operand. The second operand of selector must be an integer, say k, between 1 and the length of the sheaf. Selector returns the k-th selector of the sheaf, where selectors are ordered by the relation implemented by the PO

---

3. In this and subsequent figures, "lightening bolts" are used to label the items; they are not to be taken as part of the graph of items.

less-than. It is for this reason that less-than is defined over all the selectors, not just the integers.

A bundle is a sheaf access to whose components is controlled: there are no PO's for enumerating the extractors of a bundle. The PO

new-bundle{shf}

yields a new bundle having components and extractors which are the components and selectors of its operand. The PO's

extract{bnd,sel}

is-extractor{bnd,sel}

manipulate bundles in the same way that select and is-selector manipulate sheaves.

Because no analogue to the PO's length and selector exist for bundles, the only way to obtain an extractor from a bundle is by trial and error using is-selector. Since the elementary items can be enumerated, all elementary extractors of a bundle can eventually be found. It is not possible to enumerate the keys however. Consequently there is no way to obtain from a bundle a key which is one of that bundle's extractors. Thus by using keys as extractors, the components of a bundle can be protected from those able to reference the bundle.

2.8. Operations on Cells.

There are three PO's on cells:

new-cell{any}

contents{cel}

update{cel,any} .

New-cell yields a new cell whose contents is its operand.

Contents takes a cell and returns its contents.

Update yields no results. It is invoked purely for its effect: following the resulting execution, the contents of the cell is the second operand of update. See Figure 2.8.1.

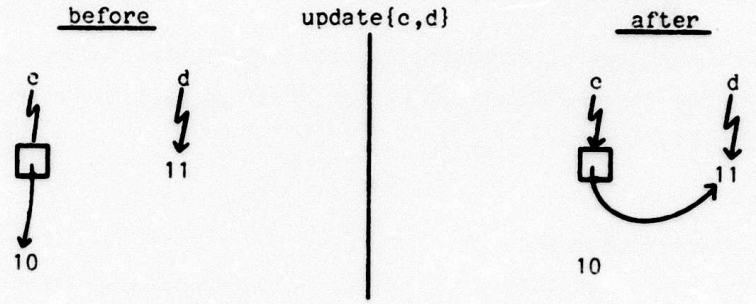


Figure 2.8.1. Updating a cell.

All items other than cells are immutable. Their structure and the items which they directly enclose are the same in all states. The contents of a cell however can be changed by using update. Thus cells are the sole source of mutability in the model.

Because of the side-effects which updating a cell can have, it is important to be able to ascertain where a cell is referenced. To help in this, it is convenient to have a PO which will permit the testing of "two" cells for identity. The PO identical serves the same purpose. However, it will be convenient (see Chapter 5) to be able to restrict the use of identical which is a completely general identity test and still retain the the ability to test the identity of cells. Consequently,the PO

same{cel,cel}

is provided for testing the identity of cells. (Notice the similarity between this limited identity-testing PO and the one for selectors: equal. The argument for the existence of equal is the same as this argument for the existence of same.)

2.9. Operations on Functionals.

A functional with an empty environment can be created using

install{shf} .

This PO expects an item whose closure is a control structure representation (CSR). The structure of CSR's is discussed in a subsequent section after the nature of control structures has been explained. Install yields a

functional whose eontrol structure is that represented by the CSR and whose environment has no bindings. The major job of install is to translate the CSR into a control strueture.

The PO

bind{fnl,sym,any}

also creates new funetionals. It yields a new funetional whose control structure is the same as that of its first operand, and whose environment is that of its first operand extended with a binding for the symbol to the item (see Figure 2.9.1). As with augment on structures, bind does not alter its first operand; it ereates a new functional which has an extended environment. It is an error to attempt to bind a symbol using a funetional whose environment already eontains a binding for that symbol.

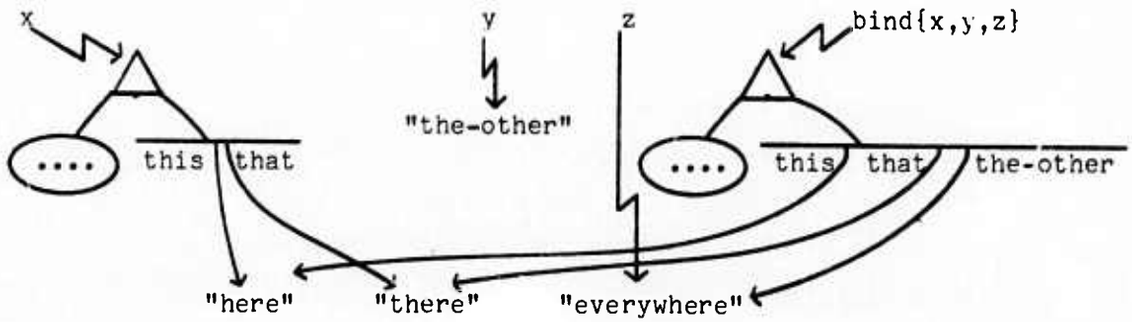


Figure 2.9.1. Aetion of the PO bind.

The purpose of binding a symbol to an item in the environment of a funetional is to give meaning to uses of a eertain identifier in the eontrol strueture of the functional. That identifier is the identifier represented by the symbol being bound in the CSR from which that control strueture was ereated. Subsequent diseussion of the evaluation of funetionals will indieate how this purpose is aehieved.

Notiee that bind creates only one new binding. Therefore, the environment of a functional is created ineremental ly. As the examples of the next ehapter will demonstrate, incremental binding of the identifiers of funetionals is very useful: among other things, it is used to model the distinetion often made in programming languages between external (or

non-local) identifiers and parameters. However, since binding is incremental, and new functionals result when binding takes place, it is possible to drop this distinction in favour of the attitude that the parameters of a functional are all those unbound identifiers of the function which must be bound before the functional is evaluated. This attitude must be supported by another: in addition to being used for evaluation, a functional can be used to create other functionals through binging.

Also, because bind creates functionals which enclose the items used in the bindings, bind is the means of achieving self-sufficiency in the model. To employ one functional in another, the identifier which is to name the serving functional is bound to that serving functional. The resulting functional encloses the serving functional; it therefore need not be a concern of any user of that new functional.

The last PO which is applicable to functionals is

eval{fnl} .

It is the mechanism for invoking the evaluation of a functional. In essence, evaluation involves carrying out the activities specified by the control structure of the functional using the environment of the functional for resolving identifiers. Subsequent sections discuss evaluation in detail. In doing so, they explain how the evaluation of a functional terminates and specifies a list of items to be "returned". These items become the result of the invocation of eval. Thus the number of items yielded by eval is dependent upon the functional which is its operand.

The fact that there are no PO's for analyzing functionals enables sophisticated protection: A functional can only be bound or evaluated; there is no way to determine how it is constructed. Consequently, sensitive data bases of items may be enclosed in "caretaker" functionals by binding, and those functionals may be widely diseminated without endangering those data bases.

2.10. The Process and Activations.

The task of the process appears in the initial state of the model in the form of a reference to a functional. The state transition rule produces from this initial state other states in which the process parts are records of partial evaluations of this task.

During the evaluation of its task, the process may invoke the PO eval with some functional as operand. When this happens, the process evaluates the operand of eval before continuing the evaluation of the task. It does this by creating a new activation which starts with an environment that is a copy of the environment of the functional, and which has a control point that is the root control directive of the functional's control structure. Since this activation is now the most recent activation, the process continues by evaluating the control structure of the new activation, using the environment of the new activation for resolving the identifiers in that control structure. During this activity, eval may be invoked again, in which case this activation is interrupted, and yet another activation is created. Thus at any time, the process is viewed as evaluating the top element of a stack of partially completed evaluations of functionals; the stack grows when eval is invoked, and shrinks when an evaluation is completed. Figure 2.10.1 depicts this.

The initial state has an empty stack of activations. It also has a task to be evaluated. An activation for the task is placed on the stack. This activation is carried out, possibly being interrupted by other activations. When it is complete, the model is considered to be in a final state. Thus states with empty activation stacks are either initial or final. The only other final state of the model is error.

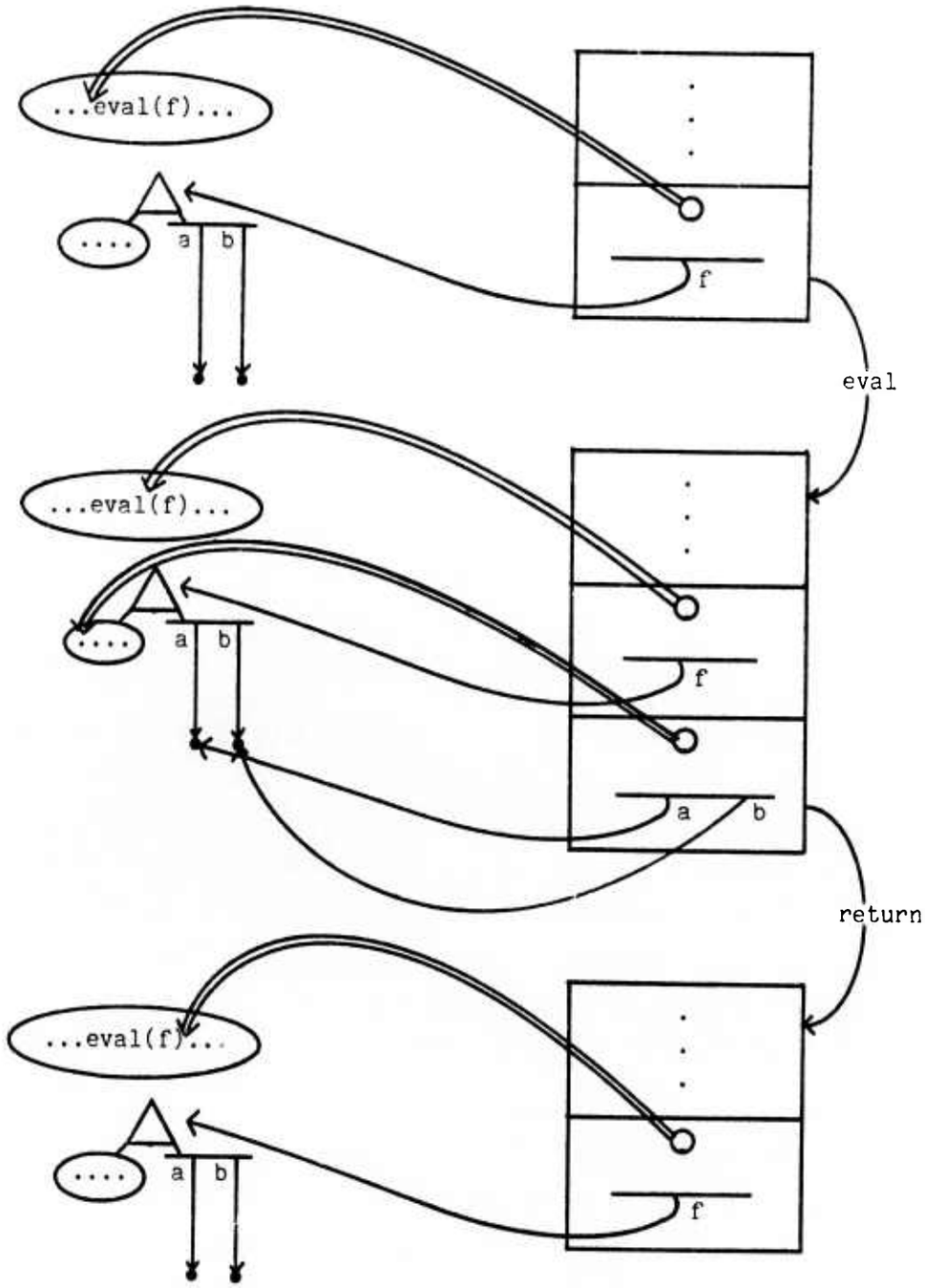


Figure 2.10.1. The effects of eval and return on the process's stack of activations.

2.11. Evaluating a Control Structure.

Manipulation of the items in the graph is done by invoking PO's. The purpose of the control structure of the functional is to specify the sequence of PO's to be invoked during the evaluation of the functional, the operands which are to be used with those invocations, and the results to be returned by the evaluation.

To do these jobs, control structures must have means of specifying PO's, sequencing, and operands. These mechanisms must be powerful enough to arrange for the results of a PO invocation to be used as operands of another. Furthermore, the mechanisms must support sophisticated programming techniques, and yet be at the same time as simple as possible. This section discusses how the control structures of the Binding model are organized, and how their interpretation meets the above requirements.

Firstly, operands of PO's and results of evaluations are specified by using identifiers. Identifiers are represented in control structures representations by symbols. They are indicated to the bind PO also by using symbols. They may thus perhaps best be thought of as an "internal (to control structures and environments) form of the string of characters which define a symbol". During the evaluation of a functional, identifiers appearing in the control structure of that functional are resolved in the environment of the activation created for that evaluation.

Control structures are trees of control directives. There are seven kinds of control directive, one of which (execute/declare) permits PO invocations, one of which (return) specifies termination of the evaluation and the results to be returned, and the others of which (sequence, conditional, iteration, loop, exit) specify sequencing. These control directives are described in detail later in this section.

The "execute/declare" control directive specifies the invocation of a PO, and a list of identifiers which are to be used to name the items which result from the execution of that PO. Thus, invoking a PO which yields one or more results is accompanied by adding bindings for identifiers to the

environment of the activation: one binding for each of the results. Thereafter those identifiers can be used to name the items yielded by the PO. Such adding of bindings for identifiers to the activation's environment is called declaration of those identifiers. Figure 2.11.1 illustrates the action of the execute/declare control directive.

The purpose of declaring identifiers is to route items yielded by PO's to other invocations of PO's where they are used as operands. When all invocations of PO's which use an item as operands have been made, the binding for that item in the environment of the activation can be removed. Contracting the context in this way is desirable for it prevents cluttering the environment with bindings which are no longer necessary.

Removal of unneeded bindings from environments can be achieved by limiting the scope of the declarations made by execute/declare control directives. This is done in the model by giving the execute/declare control directive a body which is a control structure. The evaluation of this body is the scope of the declarations caused by evaluating the control directive. Figure 2.11.2 gives the linear notation and conceptual structure of this control directive. For now, "expression" can be taken to mean the specification of a PO invocation.

The identifiers declared must match in number the results yielded by the PO invocation. Also, they must not already have bindings in the environment when the process encounters the control directive. The identifiers are bound to the items in the environment, and the body is evaluated. When the evaluation of the body is finished, the process returns the environment to the state it was in when the control directive was first encountered. See Figure 2.11.1.

Disallowing the redeclaration of identifiers and requiring the nesting of declarations makes it possible to ascertain the routing of items from PO results to PO operands by observation of the control structure. Also the item bound to an identifier will never change within the scope of that identifier. These facts simplify proving things about the activities specified by control structures, an undertaking often called "proving programs".

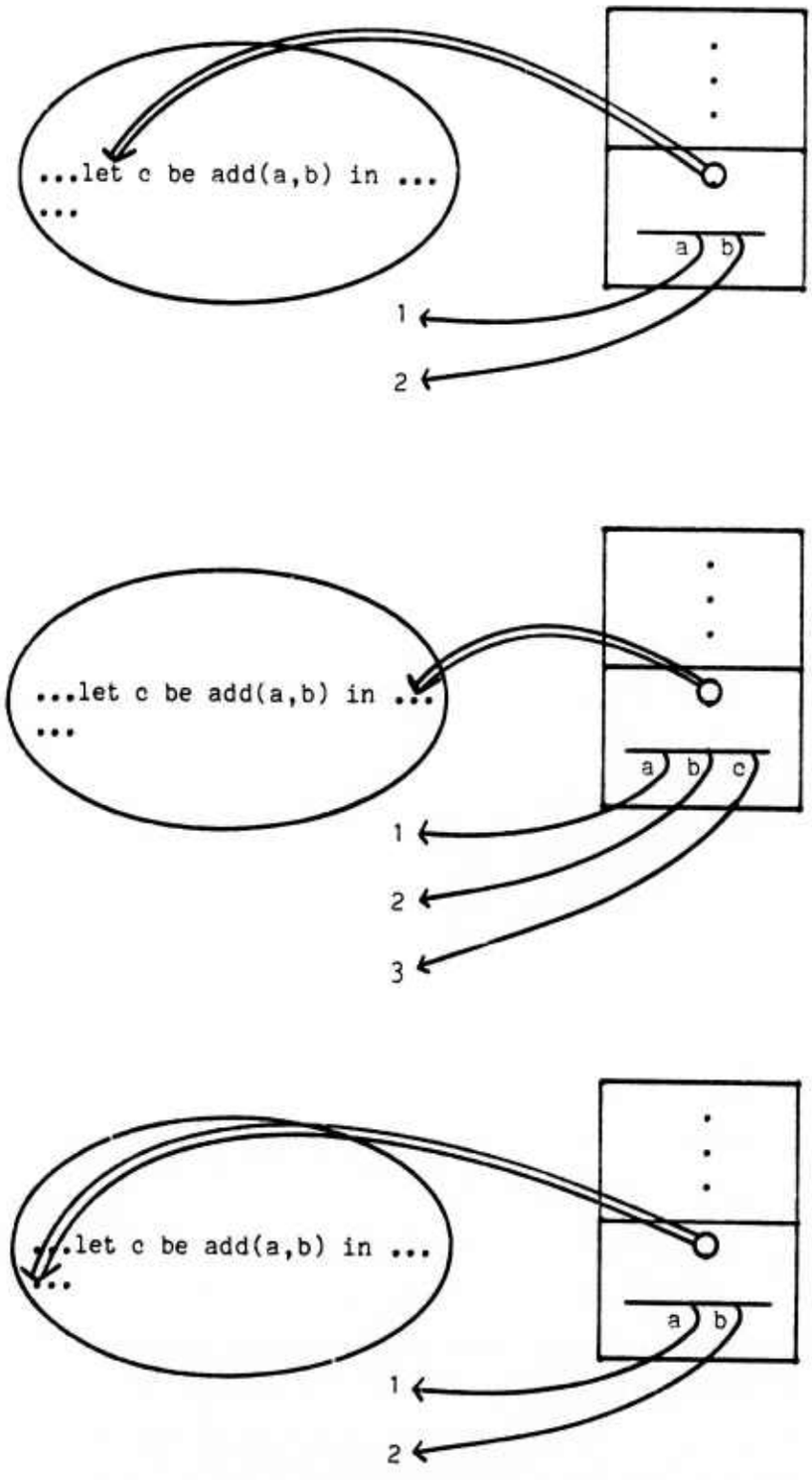


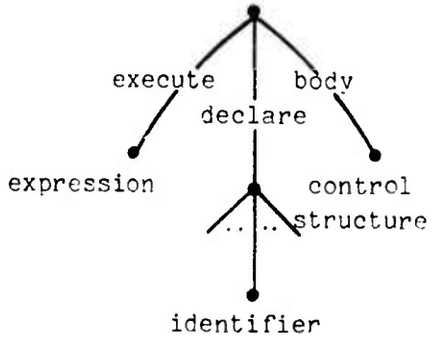
Figure 2.11.1. Action of an execute/declare control directive.

linear notation

conceptual structure

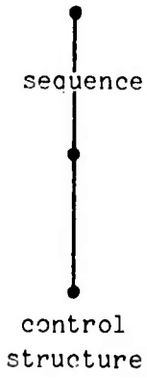
execute/declare

```
let <identifier>,...,<identifier>
  be <expression>
    in <control structure>
```



sequence

```
begin
  <control structure>;
  ....;
  <control structure>
end
```



conditional

```
if <identifier>
  then <control structure>
  else <control structure>
```

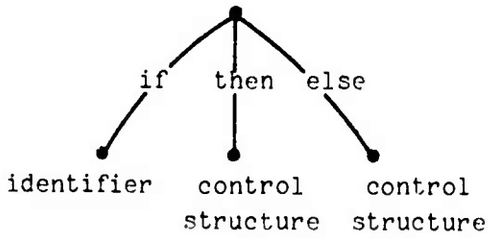


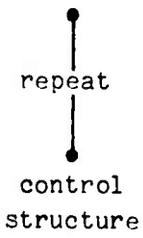
Figure 2.11.2. Structure of control directives.
(This figure is continued on the next page.)

linear notation

conceptual structure

iteration

repeat <control structure>



loop

exit

return

return <identifier>,...,<identifier>

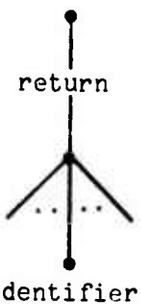


Figure 2.11.2 (continued). Structure of control directives.

These invariances also force the creation of a local variable to be explicit. A new cell is created with new-cell and an identifier is declared to name it. Associations between the identifier and the cell do not change; the contents of the cell may be changed by using update. The fact that local identifiers need not be bound to cells makes immutability the normal case, and mutability the exception to the rule. This contributes to the expressive power of control structures for it permits a programmer to make clear his intentions concerning mutability. The expression of these intentions also can simplify proving programs.

This declaration mechanism provides no means for permanently affecting the environment. Thus some other mechanism must be provided for returning results from the body of the declaration. Therefore in the model the evaluation of a control structure always has a result, called a result structure. Result structures have two parts: a result signal indicating why the evaluation of the control structure terminated, and an optional list of references to items which are to be returned as the items yielded by the evaluation.

There are four signals used in result structures: return indicates that results of the control structure are being returned; loop indicates that the next iteration of the body of the smallest enclosing iteration control directive should be started; exit indicates that the smallest enclosing iteration should be terminated; normal indicates that none of the above three special signals is present. When an iteration is terminated through use of an exit signal, the signal it returns is normal. The elements of a sequence are evaluated in turn until a non-normal signal results. If all elements of a sequence return normal, then the sequence signals normal.

A result structure containing either of the signals exit or loop will terminate the evaluations of control directives and contract the environment accordingly where necessary back to the smallest enclosing iteration control directive. A return result structure will have this same effect all the way back to the top of the activation.

The return control directive (see Figure 2.11.2) creates a result structure with a return signal indicating the termination of a control structure evaluation. Such return result structures exercise the option to include lists of results; no other result structures do. The list is prepared by resolving the identifiers of the control directive in the environment, and using the resulting references to form the list.

The result structure returned by an execute/declare control directive is whatever result structure was returned by the body of that control directive. Consider Figure 2.11.3. Evaluating CD1 will augment the environment as shown by the dotted binding. Then evaluating CD2, the body of CD1, will create a result structure including a reference to the item 3. The dotted binding is then removed from the environment to complete the evaluation of CD1. The result structure yielded by CD1 is that yielded by CD2. It includes a reference to 3 even though the binding for the item 3 is no longer in the environment.

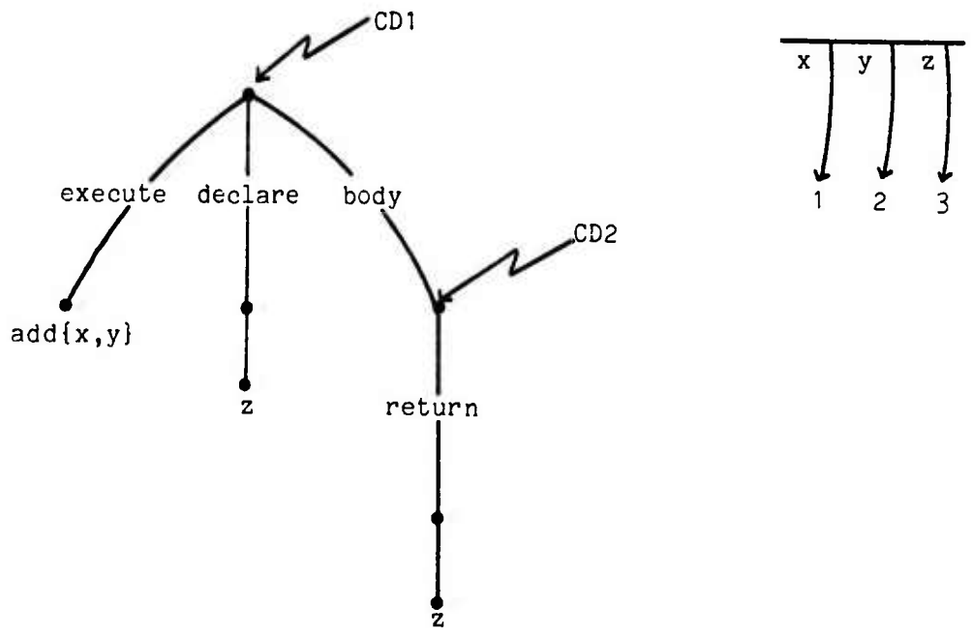


Figure 2.11.3. Returning items from an execute/declare control directive.

The other control directives are for specifying the path of the process among execute/declare and return control directives. The sequence control

directive (see Figure 2.11.2) contains a list of other control structures. If the list is empty, a result structure having a normal signal is yielded by the sequence. Otherwise, each element of the list is evaluated in turn until one of them yields a result structure with some signal other than normal. When this happens, that non-normal result structure is returned as the result of the sequence. If all elements of the list of control structures yield normal result structures, then the sequence also yields a normal result structure. The empty sequence is provided for completeness and as the model's version of a "nop".

The conditional control directive (see Figure 2.11.2) permits the path through the control structure of an activation to depend upon the bindings in the environment of that activation. The identifier is resolved. If the result is not a logical item then it is an error. Otherwise, the "then" control structure is evaluated if the item is true, and the "else" control structure is evaluated if the item is false. The result structure of the conditional is the result structure of whichever control structure is evaluated.

The iteration control directive (see Figure 2.11.2) causes repeated evaluation of the repeat control structure which is its body. Within this control structure, encountering a loop control directive (see Figure 2.11.2) causes the next iteration to be started; encountering an exit control directive (see Figure 2.11.2) causes the iteration to cease and a result structure signalling normal to be returned. This is achieved by having the loop and exit control directives simply generate result structures which signal loop and exit respectively. These result structures return back up the tree of control directives until an iteration control directive is encountered. Loop starts a new iteration; exit causes the iteration to return a result structure signalling normal. If no iteration control directive is encountered in the tree it is considered an error, for then loop or exit have not been used within the body of an iteration.

These rules cause a result structure signalling return to return back through all control directives. Thus return causes the termination of an

activation.

If a result structure signalling normal is produced by the control structure of an activation, the model turns this into a result structure signalling return and having an empty result list. Loop and exit result structures are errors in this situation.

When a result structure for an activation has been produced (it will always signal return), the process discards the activation just completed, and makes the (possibly-empty) list of items in that result structure the result of the invocation of the PO eval which caused the process to create the activation. The process's next job is then a declaration in the calling activation using these results.

The expressions which specify PO invocations are given in Figure 2.11.4. There are four forms, depending upon the number of operands expected. The next section gives an example demonstrating control structures.

A final comment in this section concerns the information imagined to exist in the control point of an activation. When moving back up the tree of control directives, the control point carries with it the result structure for the control structure whose root control directive it has just left. Also it carries some memory of which control directive that was. These two pieces of information, together with the knowledge of the control directive to which the control point has "returned", permit the process to correctly "resume" the evaluation of this previously-visited control directive at which it is now situated.

The evaluation of a control structure ends when the control point tries to "move back up" from the control directive which is the root of the control structure.

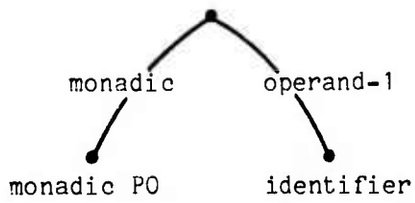
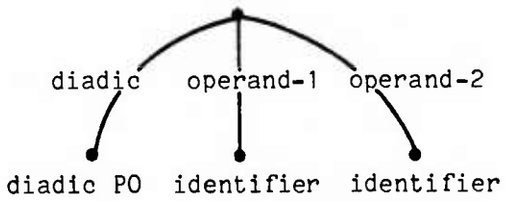
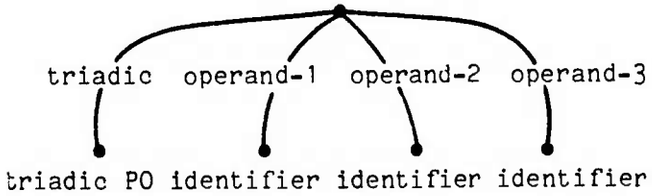
linear notation	conceptual structure
anadic invocation	
<PO-name>{}	
monadic invocation	
<PO-name>{<identifier>}	
diadic invocation	
<PO-name>{<identifier>, <identifier>}	
triadic invocation	
<PO-name>{<identifier>, <identifier>, <identifier>}	

Figure 2.11.4. The PO invocation expressions.

2.12. An Example Control Structure.

Figure 2.12.1 gives the linear notation and conceptual structure of a control structure for the factorial function. This control structure has five "external" identifiers: int-1, int-2, sym-n, f and n. These are identifiers which are used but not declared in the control structure of the functional. It is expected that int-1, int-2 and sym-n would be bound to the items 1, 2 and "n" respectively by some creator preparing a functional which implements factorial. The identifier f must be bound to the final functional itself[4].

To make use of the functional, the identifier n must be bound to a positive integer, and the resulting functional must be evaluated. When this happens, an activation is created with an initial environment having bindings for all five external identifiers. The PO less-than is invoked, and the identifier t1 declared to be the resulting logical item. T1 is then used to control the path of the process: if it resolves to true, then the integer 1 is returned. Otherwise, in successive invocations of the PO's subtract, bind, eval, and multiply with respective declarations of t2, t3, t4, and t5, the integer value "n * f(n-1)" is developed. Notice that the call to f involves both binding to f's external identifier n and evaluating the resulting functional. Finally t5 is used to return the computed integer.

---

4. This requires self-reference; how self-reference can be effected in the model is discussed in the next section.

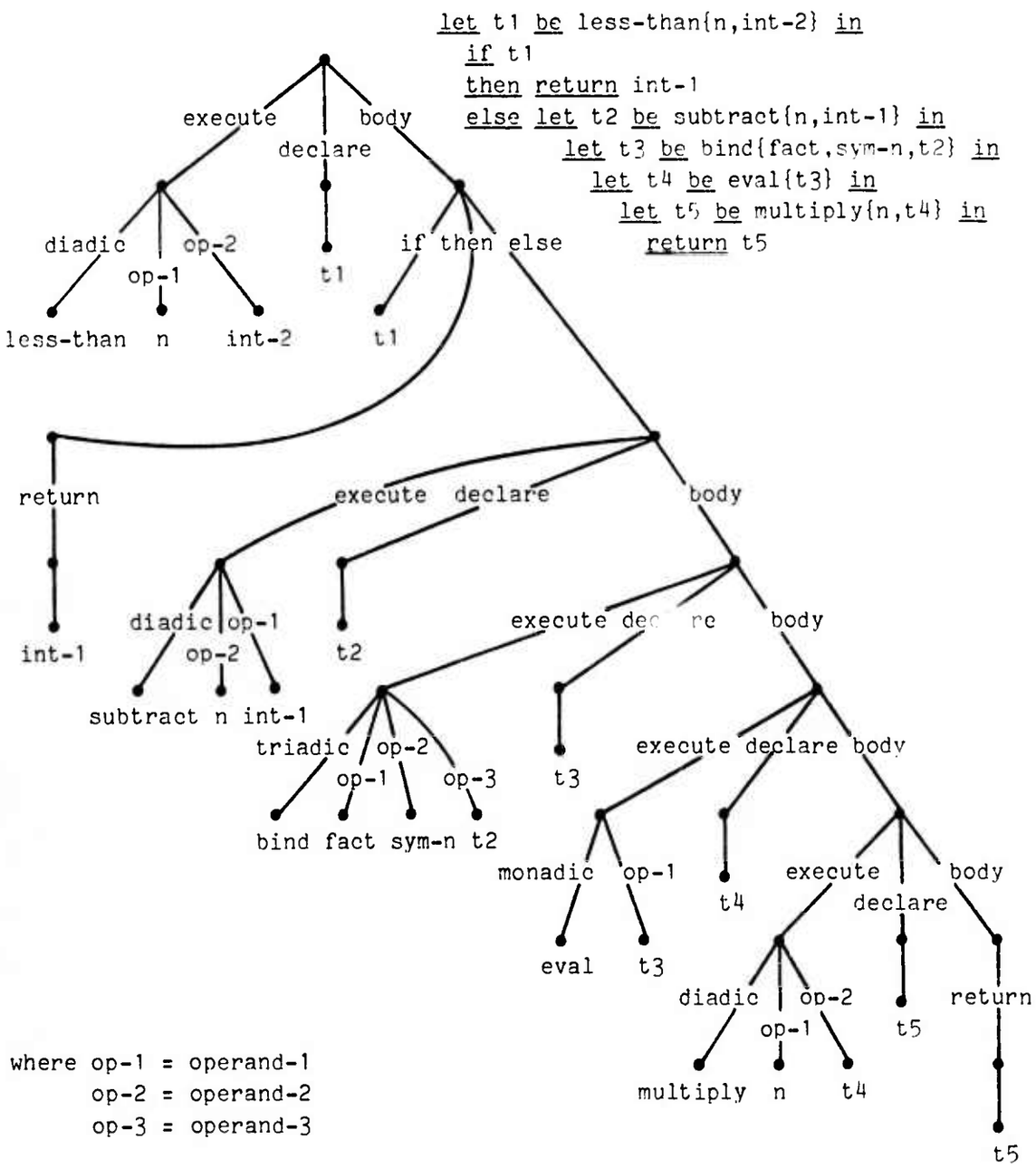


Figure 2.12.1. Schematic of Control Structure for the factorial function.

2.13. Computations and Self-reference.

An expression always yields a list of items. Four forms of expression — the PO invocations — have already been discussed. A fifth form of expression — the compute expression — is included in the model to permit expressions to yield two special sorts of such lists: the results of the evaluation of a complete control structure, and results enclosing self-referential items. By returning results to a position within an activation, all the bindings for "intermediate" items of a computation can be cleaned out of the environment without also relinquishing access to the few items which are the end product of the computation. Further, such "intra-activation returns" permit multi-level exits.

Intra-activation returns are provided by the compute expression in its "simple form" (see Figure 2.13.1). This turns a complete control structure into an expression. The declare part is a list of placeholders. The length of this list indicates how many items are to be returned; it must match the number expected by the execute/declare control directive of which it is a part.

To execute a simple compute expression, the body is evaluated. The result structure is turned into a return result structure in the same manner that eval does: loop and exit are errors, and normal is treated as return with no items. The return result structure is yielded as the result of the simple compute expression.

The second need served by compute expressions is for a way to create cycles in the graph of the items in a state without using cells. An item is called self-referential if it is enclosed by some item which it directly encloses.

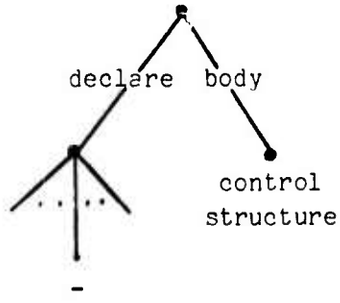
It is simple to create self-referential items by using cells. However in many cases the only purpose for using the cell would be to provide for self-reference. For example, recursive functions are represented in the model by self-referential functionals. Consider the recursive factorial functional discussed in the previous section. Suppose in some environment

linear notation

conceptual structure

simple form

```
compute -,...,-
  in <control structure>
```



full form

```
compute <identifier or ->,
          ...,
          <identifier>
  in <control structure>
```

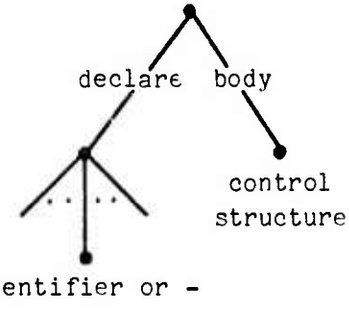


Figure 2.13.1. The compute expression.

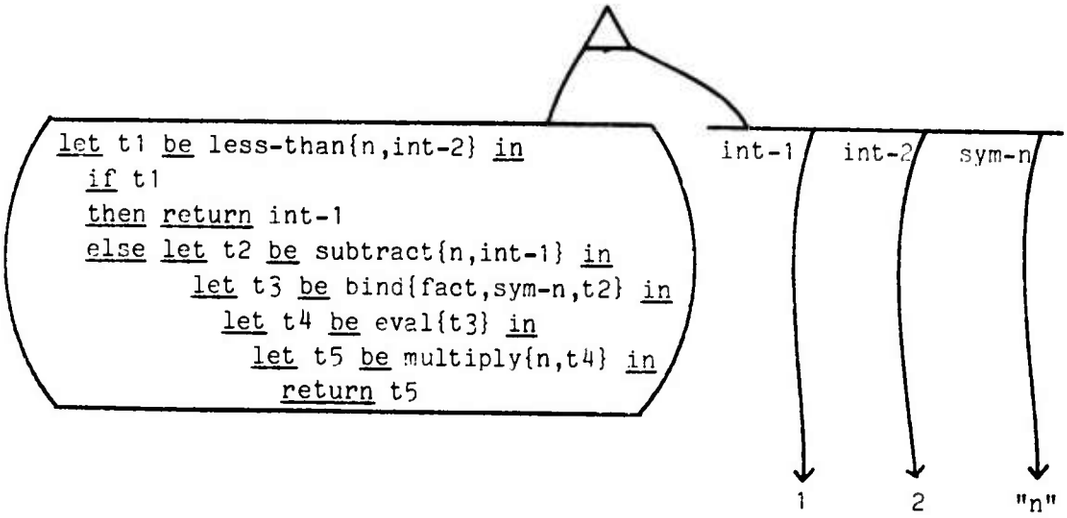


Figure 2.13.2. A partially bound functional representing the recursive factorial function.

the identifier f is bound to a functional matching this form; that is, a CSR was installed, and the literals bound (see Figure 2.13.2).

To save writing, the phrase "item(f)" will be used to mean "the item bound to f", with the environment unspecified where context makes it clear. Identifiers will be underlined in the following discussion.

The identifiers n and f are unbound externals of item(f). In the assumed environment, g is declared to be the result of binding fact to item(f) in item(f). Item(g) will then have fact bound to a functional which in turn has fact unbound (see Figure 2.13.3). When item(g) is evaluated, fact will be unbound and if another recursive call is necessary an error will be detected.

A cell can be used: declare c to be new-cell{0}; declare h to be bind{f,"fact",c}; then update{c,h}. This results in the structure of Figure 2.13.4. Now, the control structure of item(h) will cause an error when it tries to use a cell as the first operand of bind. Not only that, but the cell will never be updated again.

Thus the use of cells for achieving self-reference is not consistent with reserving cells for providing mutation. Also it involves deciding at the time of creating control structure representations that functionals

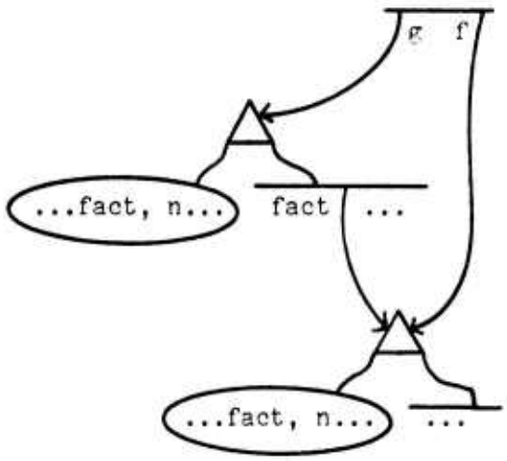


Figure 2.13.3. An incorrect binding for factorial.

Figure 2.13.4. Use of a cell in achieving self-reference.

will be recursive on certain identifiers so that contents may be used before binding is attempted. Above all, using cells does not match the natural intuition of the structuring of recursive functions. Figure 2.13.5 shows the intuitive structure for the self-referential factorial functional.

For these reasons self-reference without cells is necessary. Because mutation is needed to achieve self-reference and because cells are the only mutable item, self-reference without cells cannot be achieved with any of the mechanisms so far discusssed.

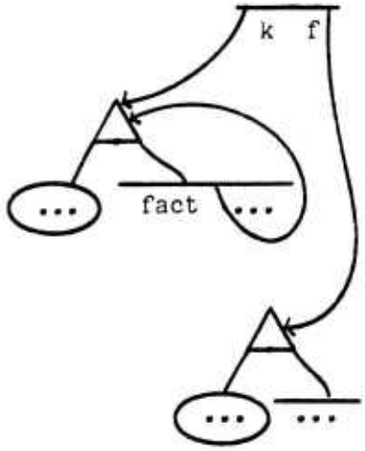


Figure 2.13.5. Intuitive structure of recursive factorial.

The full compute expression is therefore included in the model. It provides a way to get a reference to an item before that item has been created. This is done by creating an entity which stands for some undetermined and probably uncreated item. This object is called a token. A token is not an item, but it may be referred to as if it were one. References to tokens are called potential references. They may be used in creating other items. It is an error to use them to access the item for which they stand, since the identity of that item is not yet known.

Eventually the computation will create or discover the item for which the token has been standing. This item is used to determine the token: All references to the token are replaced with references to the item, and the token is discarded. The item used to determine a token is called that token's determination.

The introduction of tokens into the model somewhat complicates the simple picture of the graph of items, for a token appears as an "incomplete" node of that graph: its information content is unknown. Determining a token removes the incomplete node from the graph by merging it with the item which its the determination of the token. Because graphs with incomplete nodes are conceived of as being a little strange, the presence of such nodes in graphs should be forced to be temporary. Consequently, the model arranges to provide tokens to a process in such a way that the process in acquiring them commits itself to determining them

later.

On beginning the evaluation of a compute expression, a token is created for every element of the declare list which is an identifier. A binding from each identifier to the corresponding token is added to the environment. The body is then evaluated, and a return result structure is created as for eval or simple compute expressions. Items and tokens in this return result structure are paired with elements of the declare list; tokens bound to those identifiers are then determined using the corresponding items in the result list. It is an error to attempt to determine a token using a token; a process can only discharge its commitment to the model by presenting an item with which the model can determine a token. The return result structure is yielded as the result of the full compute.

The simple compute is the special case of the full compute in which no element of the declare list is an identifier.

```
let k be
  compute t1 in
    let t2 be bind{f,"fact",t1} in
      return t2
in ...
```

Figure 2.13.6. Creation of the structure of Figure 2.13.5.

The self-referential functional implementing factorial given in Figure 2.13.5 can now be constructed by performing the actions of Figure 2.13.6.

```
let h1,h2 be
  compute t1,t2 in
    let k1 be bind{f1,"g1",t1} in
      let k2 be bind{k1,"g2",t2} in
        let l1 be bind{f2,"g1",t1} in
          let l2 be bind{l1,"g2",t2} in
            return k2,l2
in ....
```

Figure 2.13.7. Creation of a pair of mutually recursive functionals.

A pair of "mutually recursive" functionals can be created by using a compute expression with two identifiers. Figure 2.13.7 shows how this could be done. The functionals bound to f1 and f2 are both assumed to have the identifiers g1 and g2 unbound. The identifiers h1 and h2 are to be declared in the environment and bound to two new functionals created from f1 and f2 by binding g1 and g2 in each of them to the new functionals themselves. Different pairs of identifiers have been used in this example for clarity. However g1, h1 and t1 could all be replaced by f, and g2, h2 and t2 by g with no change in the final result.

2.14. Control Structure Representations.

The previous four sections have given the details of the structure of control structures in a schematic way. The choice of representation which the model uses internally is irrelevant here. (Some thoughts about that appear in Chapter 7.)

However, some item must be used as an operand of install. The closure of this item is taken by install to represent a control structure. This section explains the forms used in control structure representations (CSR's).

Figure 2.14.1 tabulates the schemes for this representation. All tree-like parts of control structures are represented by sheaves: selectors are symbols which have the same characters as the labels used on the lines in Figures 2.11.1, 2.11.3 and 2.13.1. All lists are represented by rows. All identifiers are represented by symbols with the same characters as the identifers. Loop and exit control directives are represented by symbols "loop" and "exit". The placeholders used in the declare lists of compute expressions are represented by empty sheaves.

Although symbols are used for a number of purposes, context removes ambiguity. Thus "eval" and "exit" could be used as an identifiers in control structures.

The translation of CSR's into control structures is done by the PO install. In effecting these translations, install checks that the CSR's represent well-formed control structures.

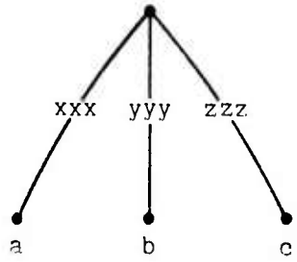
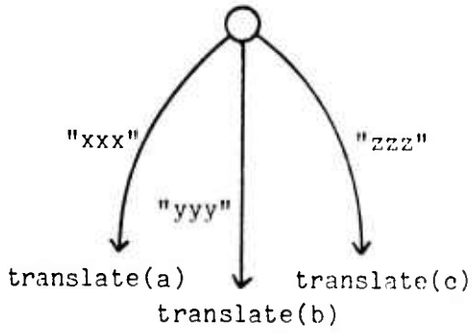
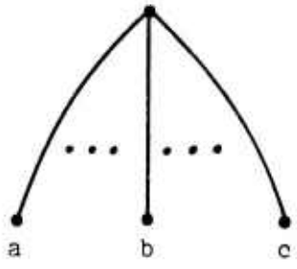
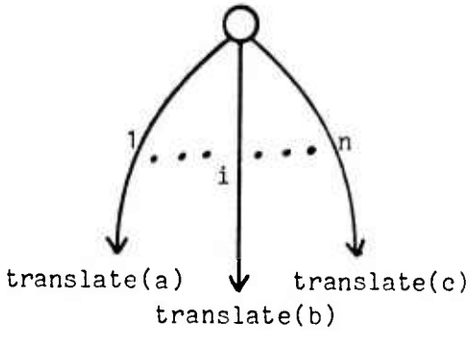
conceptual structure	is represented by	CSR form
record		
row		
symbol	this	"this"
placeholder	-	○

Figure 2.14.1. Schemes of representation for control structures.

Chapter 3. Examples.

This chapter is a collection of examples of the use of the Binding Model. These examples are intended both to increase the reader's familiarity with the model, and to demonstrate how certain constructs of programming language and operating system design are framed in the model.

3.1. A Demonstration Programming Language.

When working in a programming system supported by the Binding Model, a programmer's job is one of creating appropriate control structure representations (CSR's), installing them to create functionals, and binding externals of those functionals. As in existing programming systems, modules of the system can help the programmer with this task. Strings of characters can be interpreted as representing functionals. The scheme of interpretation is called a programming language; the functionals represented are called programs of the language; the strings of characters are called texts of programs which they represent.

To support a language, three things are needed: a means of representing in the model the texts of programs of the language, a method for creating these representations (an editor), and a functional which produces from representations of texts the programs which they represent (a compiler).

The first of these pieces is easy: a text is represented as a (long) row of single-character symbols (think: file of characters).

An editor is an interactive program which translates key strokes on a terminal into a text. To describe an editor, the meaning of "key strokes on a terminal" must be made clear, and a protocol of "editing commands" must be designed. Terminals will be discussed in Section 6.3. Editing is

not discussed in this report because, since many editors already exist, there is no doubt that the design of editing commands can be carried out.

Similarly, since the technology for producing compilers is also well established, there is also no need to discuss it here.

Thus the model is capable of supporting programming languages. The rest of this section gives the design for a simple language called in this report the "Demonstration Language" (DL). The printed representation of texts in this language will be used to represent functionals in the rest of this report.

In the previous chapter, a set of linear notations was introduced to provide a convenient way of describing control structures. In contrast, the forms of DL describe functionals. However, since a functional includes a control structure, DL uses the linear notations of the previous chapter as its backbone.

The following rules define DL. For completeness, they include descriptions of the notations of the previous chapter. Most of the other rules describe acceptable abbreviations of these notations.

In these rules, "CS", "EXP", "ID, and "PO" mean control structure, expression, identifier, and primitive operation respectively. If x's are strings from some class and * is some character, then the form "x*...*x" means a list of x's separated by *'s, with no *'s present in any list with either zero or one x.

1. Reserved words in control directives are underlined.
2. The execute/declare control directive appears in the form

   let ID,...,ID be EXP in CS .

   The short form

   execute EXP in CS

   can be used when the list of identifiers is empty.

3. The sequence control directive appears in the form

   begin CS;...;CS end .

   The short form

   nop

   can be used when the list of control structures is empty.

4. The conditional control directive appears in the form

   if ID then CS else CS .

   No one-armed conditionals are provided.

5. The iteration control directive appears in the forms

   repeat CS .

   The looping and termination control directives which complement it appear in the forms

   loop

   exit .

6. The return control directive appears in the form

   return ID,...,ID .

7. An invocation of a PO appears in the form

   PO{ID,...,ID} .

   The list should have at most three identifiers.

8. The compute expression appears in the form

   compute K,...,K in CS .

   where K is either ID or "-".

9. Identifiers are made up of the alphabetic and numeric charaters and the dash ("-"). They are written without quotes, and are not underlined. Identifiers may not start or end with a dash. (See also rule 19.)

10. The names of PO's are reserved words. They are not underlined.

operator notation	meaning
nil	empty-sheaf{}
.cel	contents{cel}
- int	subtract{0,int}
shf$any	select{shf,any}
bnd!any	extract{bnd,any}
int1*int2	multiply{int1,int2}
int1+int2	add{int1,int2}
int1 - int2	subtract{int1,int2}
int1 ls int2	less-than{int1,int2}
int1 gr int2	less-than{int2,int1}
elm1 eq elm2	equal{elm1,elm2}
log1 and log2	and{log1,log2}
log1 or log2	or{log1,log2}

Figure 3.1.1. DL's operator notation.

form	meaning	precedence
int1 rem int2	compute - let q,r be divide{int1,int2} in return r	like *
int1/int2	compute - let q,r be divide{int1,int2} in return q	like *
elm1 ne elm2	not{equal{elm1,elm2}}	like eq
int1 ge int2	not{less-than{int1,int2}}	like eq
int1 le int2	not{less-than{int2,int1}}	like eq

Figure 3.1.2. More operator notations.

11. Nesting of expressions is permitted. Wherever an ID is permitted (except in lists of identifiers to be declared in execute/declare control directives and compute expressions), an invocation of a PO which yields a single item may be used. The compiler will introduce declarations to provide identifiers bound to the "intermediate results". (For an example, see Figure 3.1.3 at the end of this section.)
12. Operator notation is permitted. Special characters and underlined reserved words are used as infix operators. They are listed in precedence order in Figure 3.1.1.
13. Parentheses are used solely to indicate parsing of phrases in the language.
14. Certain operator notations are provided as shorthands for nested invocations. These are listed in Figure 3.1.2.
15. A special shorthand for update is provided:

    IDa=IDb

    is shorthand for

    execute update{IDa,IDb} in nop .

    This infix operator has the weakest binding power in DL.
16. Creation of sheaves can be abbreviated:

    <IDa:IDb,...,IDm:IDn>

    is shorthand for

    augment{...{augment{nil,IDa,IDb},...},IDm,IDn} .
17. Binding a functional can be abbreviated:

    IDa with IDb:IDc,...,IDm:IDn bound

    is shorthand for

    bind{...{bind{IDa,IDb,IDc},...},IDm,IDn} .
18. Calling a functional can be abbreviated:

    IDa[IDb:IDc,...,IDm:IDn]

    is shorthand for

    eval{IDa with IDb:IDc,...,IDm:IDn bound} .

19. Because of a typographical limitation, the character "-" is used both as a dash and as a prefix and infix subtract operator. To distinguish between these uses, "-" used as an operator must have spaces on either side of it.
20. Indentation is used only as an aid to the reader.
21. Literals for elementary items are permitted. The words true and false are reserved but not underlined; they are the literals for the logical items. Integer literals are the decimal representations of the integer items they represent. Symbol literals are quoted (") strings. In preparing a functional from text, the compiler replaces literals with external identifiers in a CSR, installs the CSR, and binds those externals to the elementary items indicated by those literals.

Figure 3.1.3 gives a number of DL texts for the recursive factorial functional. The programs represented by these texts will have the identifier fact unbound. Part a is the text for a program having the CS given in Figure 2.14.1. Part b is the same text shortened by using nesting. Part c collapses a simple compute into a nested return. Part d includes literals for elementary items and uses the shorthand for calling a functional. Notice that there are many ways to expand part d, only one of which is part a.

It is sometimes useful in the following examples to include tests for exceptional or anomalous conditions. When such conditions are discovered, some sort of error reporting mechanism would usually be invoked. As the model does not have any such mechanism, there is no way to compile pieces of text which make such invocations. Therefore no such pieces of text appear in DL. However DL does include the reserved word error so that the site of such pieces of text can be indicated.

```
a) let t1 be less-than{n.int-2} in
     if t1 then return int-1
           else let t2
                be compute - in
                     let t2 be subtract{n,int-1} in
                       let t3 be bind{fact,sym-n,t2} in
                         let t4 be eval{t3} in
                           let t5 be multiply{n,t4} in
                             return t5
                in return t2
```

```
b) if less-than{n,int-2}
   then return int-1
   else let t2
        be compute - in
             return multiply{n,eval{bind{fact,sym-n,subtract{n,int-1}}}}
        in return t2
```

```
c) if n ls 2
   then return 1
   else return (n*eval{bind{fact,"n",n - 1}})
```

```
d) if n ls 2 then return 1 else return n*fact["n":n - 1]
```

Figure 3.1.3. DL texts for programs implementing factorial.

3.2. LISP Lists.

This is a very simple example of programming in DL. It is included not only because a first example ought to be easy to understand, but also because LISP lists are typical of the data structures used in programming languages.

A LISP list is a graph, each node of which is a mutable doublet of references to other nodes or to unstructured objects called "atoms". An example of the common depiction for LISP lists is given in Figure 3.2.1.

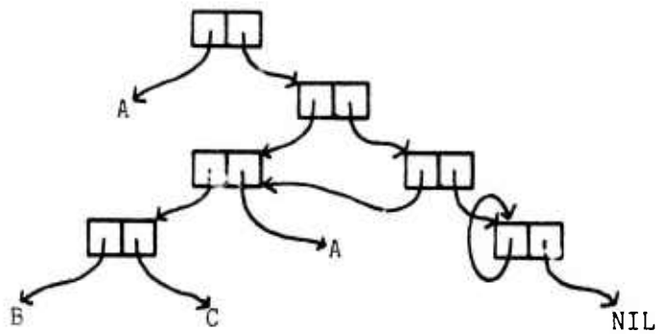


Figure 3.2.1. Common depiction of a LISP list.

There are six operations in LISP for creating and manipulating lists: CONS creates new nodes from pairs of existing ones; CAR and CDR follow references; RPLACA and RPLACD mutate nodes; ATOM tests whether a list is an atom; and EQL[1] tests whether two lists are identical. Because, as a group, these operations totally characterize LISP lists, they are called "defining operations" for lists.

In this example, atoms are represented by symbols, and nodes are represented by sheaves of length two having selectors "car and "cdr" which select cells. Figure 3.2.2 depicts the representation of the LISP list of

---

1. To prevent conflict with the DL reserved word eq, the operation which in LISP is commonly called EQ will here be called EQL.

Figure 3.2.1 using this scheme. Figure 3.2.3 gives texts for six programs which implement the six LISP operations for lists. The intended program name and a list of unbound externals[2] which appear as the first line of each text are purely cosmetic; they are not part of the text. This practice will be continued when giving other texts. For the pairs of operations CAR/CDR and RPLACA/RPLACD, the text for the first of each pair is written out in full, while that for the second is written using the abbreviations of DL.

3.3. LISP's "EQUAL".

A common built-in, but not defining, operation on LISP lists is EQUALL[3]. This operation takes two lists and checks whether they are structurally alike. It is defined recursively: if both arguments are atoms then they must be EQL; otherwise neither argument may be an atom, and their CAR's must be EQUALL, and their CDR's must be EQUALL. EQUALL is usually not defined on self-referential lists; that is, in most implementations of LISP, EQUALL is not guaranteed to terminate if either of it arguments is self-referential.

Figure 3.3.1 gives the text of a program which implements EQUALL provided it is appropriately bound at atom, car, cdr and eql, and is bound self-referentially at equall. This test is written using the defining operations on LISP lists given in the previous section. Therefore, its correct behaviour is not dependent upon the representation used for LISP

---

2. Identifiers used in texts in this report will be composed of lowercase letters. Texts, and by implication the programs produced from them, will be named using uppercase letters. To obviate the need for repeatedly discussing the binding of identifiers in functionals to supporting functionals, the following convention is adopted: An identifier in a text (spelled in lowercase) is intended to be bound to the program named with the uppercase version of that identifier. For example, rplaca appearing in a text is intended to be bound to RPLACA.

3. To prevent conflict with the DL reserved word equal, the operation which in LISP is commonly called EQUAL will here be called EQUALL.

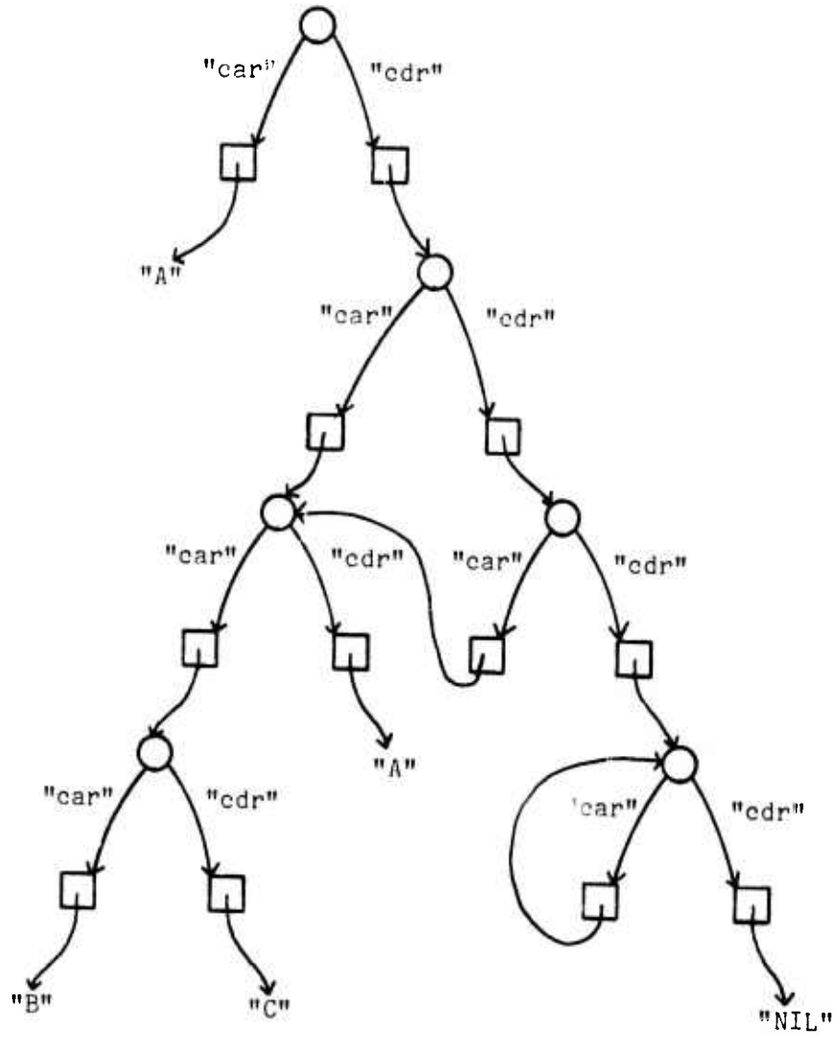


Figure 3.2.2. Representation of the LISP list of Figure 3.2.1.

```
CONS[left,right]:
  return <"car":new-cell{left},
          "cdr":new-cell{right}>

CAR[list]:
  return contents{select{list,"car"}}

CDR[list]:
  return .(list$"cdr")

RPLACA[list,left]:
  let c be select{list,"car"} in
    execute update{c,left} in nop

RPLACD[list,right]:
  (list$"cdr")=right

ATOM[list]:
  return is-symbol{list}

EQL[list1,list2]:
  return identical{list1,list2}
```

Figure 3.2.3. Texts of programs which implement the LISP operations on lists.

```
EQUALL[list1,list2,atom,car,cdr,equall]:
  if atom["list":list1] and atom["list":list2]
  then return eql["list1":list1,"list2":list2]
  else if not{atom["list":list1]} and not{atom["list":list2]}
       then return and{equall["list1":car["list":list1],
                              "list2":car["list":list2]],
                       equall["list1":cdr["list":list1],
                              "list2":cdr["list":list2]]}
       else return false
```

Figure 3.3.1. Text for the derived operation EQUALL.

lists. The representations could be changed, and EQUALL would not need to be re-written; instead it would only need re-binding to the new defining operations. This operation is therefore said to be "derived" from the defining operations for LISP lists.

```
EQUALL[list1,list2,atom,car,cdr,eql]:
  let stack be new-cell{nil} in
  let node1 be new-cell{list1} in
  let node2 be new-cell{list2} in
  repeat
    if atom["list":.node1] and atom["list":.node2]
    then
      if not{eql["list1:.node1,"list2":.node2]
      then return false
      else let stack-element be .stack in
             if length{stack-element} eq 0
             then return true
             else begin
                    stack=stack-element$"next";
                    node1=cdr["list":stack-element$1];
                    node2=cdr["list":stack-element$2];
                    loop
                  end
  else
    if not{atom["list":.node1]} and not{atom["list":.node2]}
    then begin
           stack=<"next":.stack,
                      1:.node1,
                      2:.node2>;
           node1=car["list":.node1];
           node2=car["list":.node2];
           loop
         end
    else return false
```

Figure 3.3.2. Text of an iterative program for EQUALL.

Figure 3.3.2 gives another text for EQUALL. It is written iteratively rather than recursively. It performs a complete walk of the trees defined by the lists until it discovers a structural difference. It keeps its current "positions" in the two trees as contents of the two cells denoted by node1 and node2. It keeps a stack of pairs of nodes to which it must return and check the "cdr" branch. The stack is kept in a cell denoted by stack, and is represented by a sheaf of length 3: two nodes (selected by the integers 1 and 2) and the rest of the stack (selected by the symbol "rest"). The initial contents of item(stack) is an empty sheaf; this is

used to test for a successful completion of the tree walk.

Note that neither of the uses of the loop control directive was necessary, as completion of the sequences would have propogated normal control signals back to the repeat; as normal and loop signals both cause another iteration, the effect of loop is the same as the effect of omitting it.

3.4. Circular, Doubly-linked Lists.

This section gives a sophisticated example of the use of tokens.

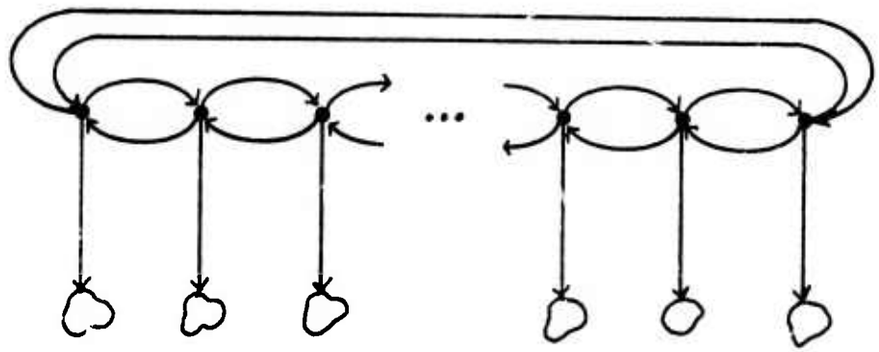


Figure 3.4.1. Schematic of a circular, doubly-linked list.

A circular, doubly-linked list (CDLL) is shown schematically in Figure 3.4.1. From each node in the list it is possible to get to both the preceding and succeeding nodes in the list without passing through intermediate nodes. A CDLL has one node which is considered its "first" node. At the far end of the list is the "last" node. The successor of the last node of the list is the first node of the list and the predecessor of the first node is the last node. Associated with each node is some data.

A program MAKE-LIST which creates CDLL's is to be written. MAKE-LIST will have an external identifier count. The evaluation of MAKE-LIST with count bound to a integer n greater than 0 yields a new CDLL with

item(count)[4] nodes. For simplicity, the data associated with each node will be a new cell with contents initialized to the integer 0.

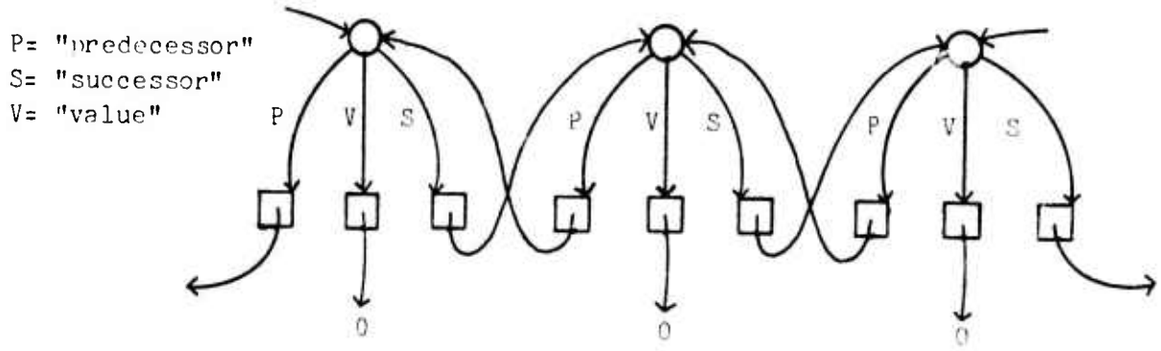


Figure 3.4.2. Three nodes in a CDLL with cells in the links.

Two representations of CDLL's are introduced in this example. Both of them represent a node of a CDLL by a sheaf with three components selected by the symbols "predecessor", "successor", and "value". The "value" component selects the cell which is the data of that node.

The first representation of nodes in CDLL's uses cells as the "predecessor" and "successor" components. These cells have as contents the appropriate nodes. Figure 3.4.2 depicts three nodes of a CDLL which uses this representation.

Figure 3.4.3 gives a DL program for MAKE-LIST which uses this representation. This program constructs a first node, iteratively strings on count - 1 more nodes, and finally links last to first to create the cycle. Thus first denotes a sheaf and last denotes a cell whose contents on each iteration is the most recently created node. There is a new declaration of new-node on each iteration; it denotes a sheaf. Note the use of exit with iteration.

A recursive program to do the same job is given in Figure 3.4.4. In it, MAKE-LIST has a supporting program REST-OF-LIST[count, top, bottom]

---

4. This notation, as introduced in Chapter 2, means "the item to which the identifier count is bound in this environment".

```
MAKE-LIST[count]:
  let first be <"predessor": new-cell{0},
                "successor": new-cell{0},
                    "value": new-cell{0}> in
  let n be new-cell{count - 1} in
  let last be new-cell{first} in
  begin
    repeat
      if .n eq 0
      then exit
      else let new-node be  <"predessor": new-cell{.last},
                             "successor": new-cell{0},
                                 "value": new-cell{0}> in
             begin
               .last$"successor"=new-node;
               last=new-node;
               n=.n - 1
             end;
    .last$"successor"=first;
    first$"predessor"=.last;
    return first
  end
```

Figure 3.4.3. Iterative DL program to create CDLL's with cells in the links.

which creates a (non-circular) doubly-linked list item(count) nodes long, with the successor of the last node being item(bottom), and the predecessor of the first node being item(top). REST-OF-LIST yields two items as results: the first and last nodes of the doubly-linked list created. It calls itself recursively to get a list one shorter than it must produce, to which it adds a node and returns the result. MAKE-LIST links first to last to make the list circular.

The cells on the links of a CDLL are neccessary only if nodes are to be inserted or deleted. Suppose that no such variation is intended. Then the sheaf which represents a node could have as components the sheaves which represnt its predecessor and successor. This leads to immutable (multiply-) self-referential items. Such items can be created in the model only by using tokens. Figure 3.4.5 gives texts for MAKE-LIST and REST-OF-LIST which create such items. The logic of the creations, the

```
MAKE-LIST[count,rest-of-list]:
  let node be <"predecessor": new-cell{0},
               "successor": new-cell{0},
                   "value": new-cell{0}> in
  if count eq 1
  then  begin
          node$"predecessor"=node;
          node$"successor"=node;
          return node
        end
  else let first,last
       be rest-of-list["count":count - 1,"top":node,"bottom":node]
       in begin
            node$"predecessor"=last;
            node$"successor"=first;
            return node
          end

REST-OF-LIST[count,top,bottom,rest-of-list]:
 if count eq 1
 then let node be <"predecessor": new-cell{top},
                   "successor": new-cell{bottom},
                       "value": new-cell{0}>
      in return node,node
 else let node be <"predecessor": new-cell{top},
                   "successor": new-cell{0},
                       "value": new-cell{0}>
      in let first,last
         be rest-of-list["count":count - 1,"top":node,"bottom":bottom]
         in begin
              node$"successor"=first;
              return node,last
            end
```

Figure 3.4.4. Recursive DL program to create CDLL's with cells in the links.

```
MAKE-LIST[count,rest-of-list]:
  let list be
   compute node in
     if count eq 1
     then  return <"predecessor": node,
                   "successor": node,
                       "value": new-cell{0}>
     else let first,last
       be rest-of-list["count":count - 1,"first":node,"last":node]
       in return <"predecessor": last,
                  "successor": first,
                      "value": new-cell{0}>
   in return list

REST-OF-LIST[count,first,last,rest-of-list]:
 if count eq 1
 then let node be <"predecessor": first,
                   "successor": last,
                       "value": new-cell{0}>
      in return node,node
 else return compute node,-
             in let subfirst,sublast
                be rest-of-list["count":count - 1,"first":node,"last":last)
                in return <"predecessor": first,
                           "successor": subfirst,
                               "value": new-cell{0}>,sublast
```

Figure 3.4.5. DL programs to create CDLL's with no cells in the links

calling sequences and items returned are the same as in the programs of Figure 3.4.4. The difference is that the cycles are created by using compute expressions rather than by updating cells. In REST-OF-LIST, notice that the compute expression is used to yield two items while creating and determining only one token.

3.5. Block Structure.

In many languages (for example, ALGOL 60), block structure has been used to indicate the extent of declarations. As in the model, declarations in block structured languages are nested. In contrast to the model, block structured langauges permit an identifier to be declared when it is already declared; the effect is to cause all uses of that identifier within the block containing the new declaration to use the new declaration; the old declaration is reinstated at the end of the block. The old declaration is said to be "occluded" by the new declaration for the duration of the block.

The declaration and use of procedures in some block structured languages (for example, PAL [Wozencraft, 71]) complicate this simple picture of overlayed declarations. In an attempt to achieve a measure of referential transparency, the declaration of a procedure creates a value which is effectively the code of the procedure together with an environment in which those identifiers that are neither locally defined nor parameters have the meaning that they have in the environment as it stands at the time of declaration. This value is referred to as the "closure of the code in the environment of declaration".

In these languages, closures may be used as any other values can. If, at the time of evaluation of a closure, the environment of declaration has been partially occluded, then that evaluation may have to re-establish that occluded environment. Also, if a closure is used as an argument of another closure, then the evaluation of the first closure within the second creates a similar requirement. Finally, if a closure is returned as a result of a computation, the later evaluation of that closure may have to re-establish

```
      begin
1       integer x
2       procedure g
        ..................
3       .........x..........
        ..................
        begin
4         integer y
          ..................
5         ......x....y.........
          ..................
          begin
6           integer x
7           procedure f(a)
8             integer a
              begin
              ..................
9             .......x....y....a...
              ..................
              end
            ..................
10          .....x....y..........
            ..................
11          g=f
          end
          ..................
12        ....x....y...........
          ..................
        end
        ..................
13      ....g(x)....x........
        ..................
      end
```

Figure 3.5.1. A fragment of a block structured language.

source	use
x: declared in line 1	in lines 3, 5, 12 and 13
g: declared in line 2	in lines 11 and 13
y: declared in line 4	in lines 4, 5, 9, 10 and 12
x: declared in line 6	in line 9
f: declared in line 7	in line 11
a: external of text of f	in line 9

Figure 3.5.2. Sources and uses of identifiers in program of Figure 3.5.1.

an environment which has been left completely.

To demonstrate how block structure can be accomodated in the Binding Model, Figure 3.5.1 gives a fragment of text typical of programs in block structured languages. Figure 3.5.2 tabulates sources and uses of identifiers in this fragment. In the innermost block, procedure f will be declared and assigned to procedure variable g. The call in line 13 will cause the environment of line 9 to have x bound to the same value as is the x of line 6, y bound to the same value as is the y of line 4, and a bound to the same value as is the x of line 1.

It is not possible to produce a single DL text which represents a functional correctly representing this fragment. However Figure 3.5.3 gives texts for two functionals which together correctly translate the fragment. MAIN has f as an external. It represents the fragment of Figure 3.5.1 correctly provided it is bound on code-f (external for the code of f) to the functional CODE-F. The occlusions of the fragment have been translated by introducing the "shadow" identifier x1 for the occluding x.

In this translation, shadow identifiers are substituted before procedures are removed. Notice that while the presence of cells is implicit in most languages, it is explicit in DL. Also notice that line 8 of the original fragment has no counterpart in the DL texts, because line 8 is included only to provide type information about the contents of the cell to which a is bound. In DL, no such type information can be expressed.

The translation scheme demonstrated in this example is quite general. It therefore appears that the Binding Model can support block structured

```
    MAIN[]:
1     let x be new-cell{?} in
2     let g be new-cell{?} in
        begin
          ......................;
3         ....x.................;
          ......................;
4         let y be new-cell{?} in
            begin
              ......................;
5             ....x....y............;
              ......................;
6             let x1 be new-cell{?} in
7             let f be new-cell{code-f with "x1":x1,"y":y bound} in
                begin
                  ......................;
10                .....x....y...........;
                  ......................;
11                g=.f
                end;
              ......................;
12            ....x....y............;
              ......................;
          end;
        ......................;
13      ....g["a":x].....x....;
        ......................;
      end

    CODE-F[a,x1,y]:
      begin
        ......................;
9       ......x1...y...a.....;
        ......................;
      end
```

Figure 3.5.3. A translation of the fragment of Figure 3.5.1.

languages.

3.6. External Identifier Binding.

Many languages partition the identifiers of procedures into three groups: those declared during the evaluation of the procedure (the locals), those which are intended to be bound by the caller (the parameters), and others (the free variables).

In most languages the free variables are all bound at the same time, although this time varies from one language system to another. In languages which solve the FUNARG problem (for example, PAL [Wozencraft, 71]), free variables are bound at function definition time. In many LISP systems they are bound just prior to evaluation.

In the model, there is no partitioning of external identifiers into parameters and free variables. Instead, any external identifier may be bound at any time by any activation having a reference to the functional. In particular, the order of binding and the number and identity of the activations and processes involved in the binding of a functional are not restricted. In fact, they may even be determined dynamically (during execution) if it seems useful to do so.

The following examples demonstrate the flexibility of the binding provided by the model. They are "toy" applications intended to illustrate the usefulness, not so much of the functionals, but rather of the flexibility in the binding order.

```
QUADRATIC[x,a,b,c]
  return (a*x+b)*x+c
```

Figure 3.6.1. A generalized quadratic transformation.

Figure 3.6.1 gives a DL program for a generalized quadratic transformation in one variable. By binding a to 0, a generalized linear

transformation is produced. Binding c to 0 in this linear transformation yields a generalized multiplier trnsformation. By binding b to a particular integer item, a specialized multiplier transformation results.

Alternatively, by binding b and c to 0 and a to 1, a "square" function can be created. Again, binding c to 0 produces the generalized quadratic with 0 as a fixed point.

By creating such generalized functionals, binding can be viewed as a means of restricting behaviours. By passing partially bound functionals, the passer can guarantee that the user stay within certain behavioural patterns.

Another example which follows this paradigm is that of a generalized "list whomping" functional (for example, MAPLIST in LISP) which has two external identifiers: one is intended to denote a list; the other is intended to denote some "whomping" functional to be applied to each element of the list. By binding a list to this functional, a whomper for that list is produced: a user may employ it on numerous occasions to manipulate that one list only. Alternatively, by "binding in" a particular whomping functional, a user may be given the means to whomp many lists in only one way.

The kind of restriction capability provided by partial binding is most suited to circumstances in which a number of orthogonal specifications are needed to define an action. The use of binding to determine one specification restricts the behaviour of the resulting functional in only that one dimension. This allows users to ignore this dimension completely; in fact, they need not even be aware that is exists.

3.7. Parameter-less Procedures.

Most programming languages require that the invocation of a procedure be accompanied by binding of a set of arguments. By permitting empty argument lists, it is possible to create procedures which are conceived of as being invoked without parameters. In the model, parameter-less

procedures are simply fully-bound functionals.

```
RNG[seed]:
  let next
  be ...compute the next number in the series from the
                contents of the cell bound to seed...
  in
    begin
      seed=next;
      return next
    end
```

Figure 3.7.1. Schematic program for a random number generator.

The classical example of a parameter-less procedure is the random number generator. Figure 3.7.1 gives a skeleton program for one. By binding a cell to seed, a user can generate a functional which, when evaluated repeatedly, will return a series of random numbers based on the initial contants of the cell. Another series may be obtained concurrently by creating another functional from the same original RNG by binding another cell to seed. As code is "pure", the two random number generators will not interfere with each other.

```
COUNTER[subtotal,total]:
  begin
    subtotal=.subtotal+1;
    total=.total+1;
    return
  end
```

Figure 3.7.2. An event-counting program.

A slightly more sophisticated example arises from the program given in Figure 3.7.2. COUNTER measures the number of times it has been called. Suppose a certain class of events is to be counted. Suppose also that the class is subdivided into a number of different subclasses. COUNTER can be used to keep both subtotals for each subclass and a running total for the

class as a whole. To initialize for the class, a cell with initial contents 0 is bound to total. The resulting functional is made generally available. A user wishing to define a particular kind of event and count it binds a private cell with contents 0 to the external subtotal of this functional. He uses the resulting functional as a parameter-less functional for counting events of that subclass. Each such subclass has an independent counter which is not effected by the counting of events for other subclasses. The whole class has a counter which is incremented by the (possibly inadvertent) co-operation of those counting events of subclasses.

Functionals like COUNTER can be used as unique name generators. The importance of partial, dynamic binding and separated evaluation in such activity is that a great variety of sophisticated naming schemes can be defined and created within the model. Because it is a precise model of procedures interacting on shared data, the model is a good basis for the conceptualization of such schemes.

3.8. Procedural Encapsulation.

In Chapter 2 the notion of a type was introduced as being defined by a collection of operations which create and manipulate instances of the type. One advantage of this notion of types is that usage of instances of types can be made without knowing anything about the representation of the instance.

A recent idea in programming (see [Liskov, 74]) is to provide programmers with the ability to define their own types by defining a collection of operations for the type. Types are then implemented by creating programs which implement those operations. The next few sections discuss aspects of the support for programmer-defined types in the model.

Consider an programmer-defined type "switch": a switch is an object which can be in one of two "positions" called on and off. A switch may be turned on by the operation SET, turned off by the operation RESET, and

tested to determine the position it is in by the operation IS-SET. Also, it is necessary to be able to create switches, and check them for identity. (Frequently the definition of a type will include equal and copy operations. These will be ignored in this discussion.)

Notice that information on how a switch is represented need not be known to users of switches. Some language designers (see [Liskov, 74]) have gone further and made such information unobtainable by those users. As a result, the programmer implementing switches may choose whatever scheme he likes for representing them, provided only that it permit him to implement the operations SET, RESET, and IS-SET. It is necessary that users know how to find those operations when they wish to manipulate a switch. The operations must be consistent with the representations chosen.

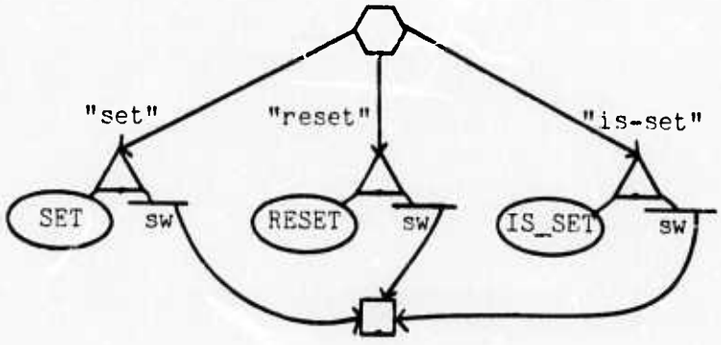


Figure 3.8.1. An instance of a switch.

One way of meeting these requirements in the model is to represent a switch as a bundle with three components, each of which is a functional implementing an operation on that particular switch (see Figure 3.8.1). Figure 3.8.2 gives a program and supporting programs for a functional which creates switches. The operations for a particular switch are all bound to the same cell; they change or check its contents. Switches are created in the off position. Figure 3.8.3 gives a program fragment which creates and manipulates a switch.

The scheme of representing an instance of a programmer-defined type by the collection of its operations is called procedural encapsulation: those items which are used to implement the operations encapsulate all the

```
CREATE-SWITCH[set,reset,is-set]:
  let sw be new-cell{false} in
  let s be set with "sw":sw bound in
  let r be reset with "sw":sw bound in
  let i be is-set with "sw":sw bound in
    return new-bundle{<"set":s,"reset":r,"is-set":i>}

SET[sw]:
  sw:=true

RESET[sw]:
  sw=false

IS-SET[sw]:
  return .sw
```

Figure 3.8. . CREATE-SWITCH.

```
.....;
let sw be eval{create-switch} in
  begin
    .....;
    eval{sw!"set"};
    .....;
    if eval{sw!"is-set"} then ..... else .....;
    .....
  end
.....;
```

Figure 3.8.3. Creation and manipulation of a switch.

information about how the instance is represented. Often, as here, the functionals are structured so that they share a collection of items. If so, that collection is called the rep (short for representation) of the programmer-defined instance. Only the functionals which implement the operations should be able to reference the rep, so as to thoroughly protect it from outsiders.

One feature of procedural encapsulation is that another functional which creates switches can, without conflict, be installed in the system

and used to produce switches having different representations. The fragment in Figure 3.8.2 would work just as well with the new functional bound to create-switch. Better still, a switch can be manipulated with no knowledge of which switch-creating procedure produced it; since switches carry their own manipulating operations, the same operation (eg. eval(sw!"is-set")) will work on any switch in the system[5].

Another example of a programmer-defined type is the LISP list discussed earlier in this chapter. CONS is its creation functional; CAR, CDR, RPLACA, RPLACD, ATOM, and EQL are the defining operations for the type.

Research is currently in progress on the design and implementation of a programming language which supports the creation and use of programmer-defined types [Liskov, 74]. The language, called CLU, has a new construct—the cluster—which can be used to express an implementation of such a type. It appears that the Binding Model can adequately support CLU and the language system which is part of its design.

3.9. Hidden Operations.

This section explains why bundles rather than a sheaves are used for representing instances of programmer-defined types.

Sheaves can be totally analyzed, while bundles cannot. By using keys as extractors on bundles, it is possible to guarantee that certain operations be hidden from users who must have access to the instance but should not be able to use those operations.

An example requiring this behaviour arises in the processing of programming languages. To parse a program written in a language, an instance of the "grammar" of that language is created. This instance "knows" the syntax for the language. It also has a "symbol table" which

---

5. Accomodating diadic operations for user-defined types when instances are represented by procedural encapsulation is difficult. This is discussed further in Section 3.10.

will be used for parsing this program only. The program is the segmented into a sequence of "lexemes", each of which is a string of characters. Lexemes are entered in the symbol table of the grammar by using an operation of the grammar whose job it is to make entries in the symbol table. This operation creates a "token"[6] which is used to stand for that lexeme.

One way of implementing tokens and grammars is as follows: CREATE-GRAMMAR produces an instance which has within its rep a symbol table. When CREATE-TOKEN of that grammar is invoked, an instance of type token results. CREATE-TOKEN may have a "side-effect" of adding an entry to the symbol table of the grammar. The token produced by CREATE-TOKEN carries within its rep an index into the symbol table of the grammar; this index is built into the token by the grammar which created it. Another operation of the grammar is used to test the precedence relation existing between two tokens which that grammar has created. This operation is handed the two tokens as arguments. It obtains from each of them its index into the symbol table and uses that information to determine the desired precedence relation. Thus tokens must have an operation which yields the token's index. Yet this operation ought to be visible to, and usable by, only the grammar which produced the token; that is, it should be "hidden" from others with access to the token.

This can be arranged in the model by having each instance of type grammar use a key which is part of the rep for grammar and not shared by other grammars as the extractor of the index-producing operation of all the tokens it creates. When handed a token, a grammar can insure that the token is one which it created by establishing that the key is one of its extractors. Then using that item as an extractor, the index-yielding operation may be obtained.

---

6. The term "token" has a well-established technical meaning in the field of programming languages. It is used with that sense in this example, and should not be confused with tokens of the Binding Model.

```
CREATE-GRAMMAR[...,create-token,precedence,...]:
  let key be new-key{} in
   ...
    let ct be create-token with ...,"key":key,... bound in
    let prec be precedence with ...,"key":key,... bound in
       ... in
      return new-bundle{<...,"create-token":ct,"precedence":prec,...>}

CREATE-TOKEN[...,key,...,get-index...]:
  let i be ...index into symbol table...   in
    let rep be <...,"index":i,...> in
       ...
      let gi be get-index with "rep":rep bound in
         ... in
        return new-bundle{<...,key:gi,...>}

GET-INDEX[rep]:
  return rep$"index"

PRECEDENCE[token1,token2,...,key,...]:
  if not{is-extractor{token1,key}} then error else
  if not{is-extractor{token2,key}} then error else
    let index1 be eval{token1!key} in
    let index2 be eval{token2!key} in
       ...
```

Figure 3.9.1. Fragments of the text for CREATE-GRAMMAR.

Figure 3.9.1 gives fragments of the text for CREATE-GRAMMAR.

Another use for hidden operations arises if instances have diadic (or triadic, etc.) operations which must be implemented on them. The representation scheme for instances conceives of all operations being associated with, and manipulating the rep of, some single instance. Diadic operations give trouble because they manipulate the reps of two instances.

Consider the programmer-defined type "rational-number". Suppose the rep of a rational number is a pair of reals: the numerator and the denominator. The plus operation is defined for each rational number: it takes a single argument (the "other" rational) and returns a new rational (the sum of the number with which the plus operation is associated and its argument). To do this, the plus operation must gain access to the rep of

its argument. However, users of rationals should not be able to gain such access. A hidden operation is needed.

```
CREATE-RATIONAL[num,den,key,...rep,plus...]:
  let r be rep with "num":num,"den":den bound in
     ...
     let p be plus with "num":num,"den":den bound in
      ... in
        return new-bundle{<key:r,...,"plus":p,...>}

REP[num,den]:
  return num,den

PLUS[num,den,create-rational,gcd,key,other]:
 if not{is-extractor{other,key}} then error else
   let onum,oden be eval{other!key} in
     let nnum be num*oden+den*onum in
     let nden be den*oden in
       let d be gcd["val1":nnum,"val2":nden] in
         return create-rational["num":nnum/d,"den":nden/d]
```

Figure 3.9.2. Fragments of the text for CREATE-RATIONAL with supporting functionals REP and PLUS.

Figure 3.9.2 gives fragments of the code for CREATE-RATIONAL, indicating how the plus operation works. The functional GCD[val1,val2] yields the greatest common divisor of item(int1) and item(int2). It is used to keep rational numbers "in lowest terms". Key denotes the key used as an extractor for the rep-revealing operation. This key is bound into the functional CREATE-RATIONAL. Figure 3.9.3 shows how this could be done. It assumes an environment in which create-rational, rep and plus denote the functionals CREATE-RATIONAL, REP and PLUS respectively. Also gcd is assumed to denote a greatest common divisor routine. Note the mutual recursion on create-rational and plus, and the way in which partial binding makes the construction precise: the non-recursive externals gcd and key each of which denotes a single item for all the functionals involved in the mutual recursion must be bound before the self-reference is created; the non-recursive externals num and den which denote different items for different invocations must be bound after it.

```
let key be new-key{} in
   ...
  let cr1 be create-rational with "key":key bound in
   ...
  let p1 be plus with "gcd":gcd,"key":key bound in
   ... in
    let cr2,...,p2,... be compute cr2,...p2,... in
                          return cr1 with ...,"plus":p2,... bound,
                                   ...,
                                 p1 with "create-rational":cr2 bound,
                                   ...
    in return cr2
```

Figure 3.9.3. Fragments of binding of CREATE-RATIONAL.

3.10. Representation of Instances by reps.

Another approach to representing instances of programmer-defined types dispenses with procedural encapsulation. It lets the rep represent the instance. Bundles are used to protect the rep. The operations for the programmer-defined type are prepared as functionals into each of which is bound a shared key. The operation which creates instances yields bundles having the rep as their sole component; the extractor for removing the rep from the bundle is the shared key. The other operations are able to reference the reps because the key is bound into them. No other functional can extract the rep from the bundles because no other functionals have access to the extractor of those bundles.

A disadvantage of this scheme is that the operations for manipulating instances are not enclosed in those instances; operations and reps travel about the system separately. Thus other protocols must be used to insure that operations are available when instances are to be manipulated. Such protocols are beyond the scope of this report. (For a linguistic approach to providing them, see [Liskov, 74].)

An advantage of this scheme is that diadic operations are symmetrical with respect to taking their arguments and accessing the reps of instances. This is aesthetically pleasing.

Chapter 4. Formal Definition.

The Binding Model was presented informally in Chapter 2. This level of description, while good for developing intuitions about the model, is too imprecise to serve as a basis of formal proofs about these intuitions. Consequently, this chapter presents a formal definition of the model.

The Vienna Definition Language (VDL) is used to define the model. This technique is explained simply and informally in [Neuhold, 71], and is defined formally in [Lucas, 68]. A knowledge of VDL is assumed in what follows.

This chapter describes the representations and programming schemes used in the definition. The definition itself appears in Appendices B (Predicates) and C (Instruction Schemata).

The definition is not intended as a guide to implemention of the Binding Model. Rather, it is written to most directly capture the intuitions about the model which have been presented in Chapter 2. Implementation is discussed separately in Chapter 7.

In Chapter 6, the model will be extended to include facilities for creating and manipulating processes. The VDL definition of the single-process version of the model is designed so that it can be extended into a definition for the multi-process of version of the model. To minimize the changes necessary to effect this extension, the structure of the VDL objects representing states is given the form needed to support the extended definition.

The following sections discuss the elements of the Binding Model, explaining and illustrating the representations of each as VDL objects. Appendix B gives the predicates that formally specify the classes of VDL objects used in the definition. Appendix C gives the instruction schemata

that define the state transition rule for the states of the Binding Model.

4.1. States.

States of the model are represented by VDL objects which are of the form acceptable as states of a VDL definition. These objects will be called VDL states in this report. Such objects are defined to be finite; this enforces the constraint mentioned in Chapter 2 that the graphs of states of the Binding Model must have only a finite number of items. The state transition rule of the model is defined by the instruction schemata of the VDL definition. Each state transition of the model corresponds, in general, to a sequence of state transitions in the definition. These correspondences will be discussed in more detail in the last section of the chapter.

Each VDL state has the model has four components; they are selected by the selectors s-items[1], s-tokens, s-processes, and s-c (see Figure 4.1.1). Together comp(s-items) and comp(s-tokens) represent the graph of items of the state. Comp(s-items) represents all the items of the state; comp(s-tokens) represents all the tokens of the state. Comp(s-processes) and comp(s-c) together represent the process of the state. Comp(s-c) is the VDL control tree. It represents the positions (see Section 2.11) of all activations in the process. Situations appear as trees of instructions yet to be evaluated. Result structures appear as arguments passed to these instructions using VDL's "PASS" mechanism. Comp(s-processes) represents the environments of all activations in the process.

---

1. All selectors in the definition have the form "s-xxx". For brevity, the notation "comp(s-xxx)" will be used to mean "the component with selector s-xxx".

<is-state>

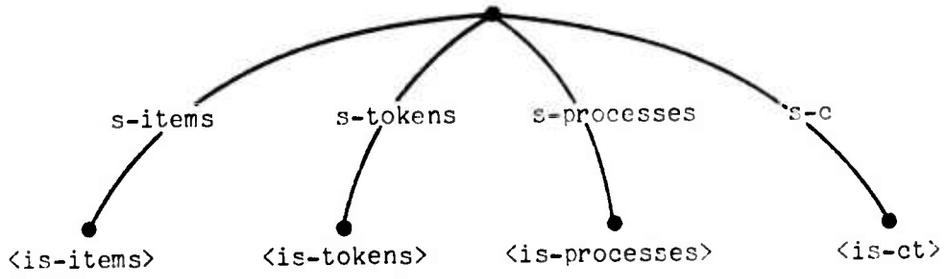


Figure 4.1.1. Top level of a VDL state.

4.2. Items and References.

The items of a state are represented by eight lists one for each primitive type of the model (see Figure 4.2.1). These lists are called type-lists in the rest of this report.

A reference to an item is represented by a type and an integer which is used to find that item in the type-list for that type (see Figure 4.2.2). The primitive types of the model are represented by the elements of the set rtp (reference type). The set rtp is primitive to the definition.

The items are represented as follows:

1. Logicals: The set of logical values is a primitive of the definition. Logical items are represented directly by elements of this set.
2. Integers: The set of integers is also a primitive of the definition. Integer items are represented directly by integers.
3. Symbols: Symbol items are represented by lists of characters. The characters are a primitive set of the definition.
4. Keys: These items have no information associated with them. Each is therefore represented by a place-holder: the VDL empty list.
5. Sheaves: These items are represented by a list of pairs of references. Comp(s-selector) of each pair is a reference to a selector of the sheaf. Comp(s-component) of each pair is a reference to the component corresponding to that selector.
6. Cells: A cell is represented by a reference to the item which is the contents of that cell.

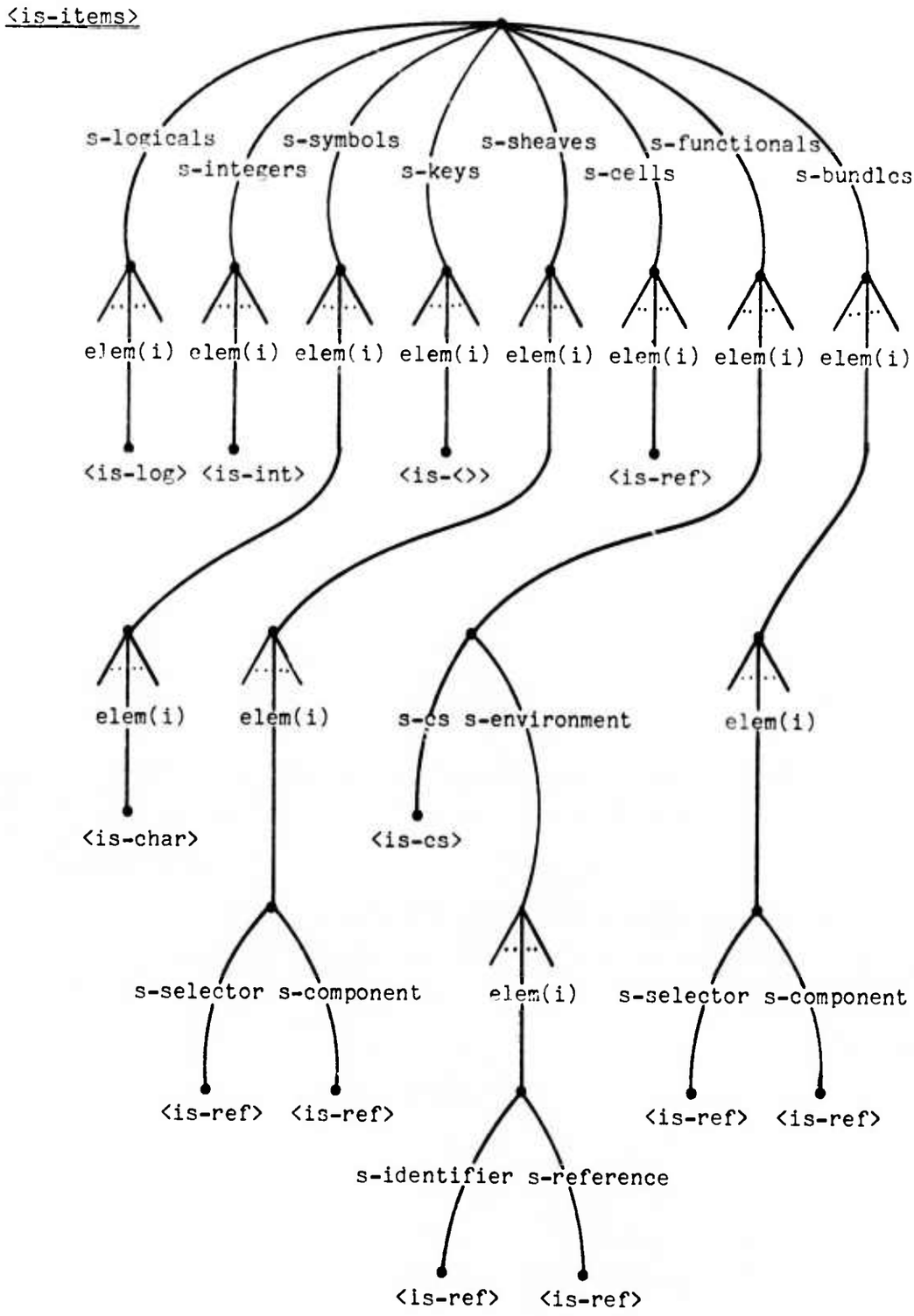


Figure 4.2.1. VDL representation of the items of a state.

<is-ref>

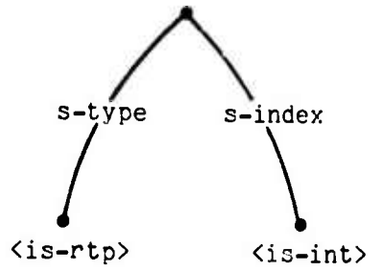


Figure 4.2.2. VDL representation of a reference.

7. Functionals: The representations for these items have two components. Comp(s-cs) and comp(s-environment) are representations of the control structure and environment for the functional respectively.
8. Bundles: These items are represented the same way sheaves are.

4.3. Tokens.

Tokens are represented as a list of placeholders (<>, the VDL empty list) in comp(s-tokens) of the state (see Figure 4.3.1).

Potential references (references to tokens) are represented like regular references: they have a reference type of rtptok, and an index which is interpreted as an offset into the list of tokens.

<is-tokens>

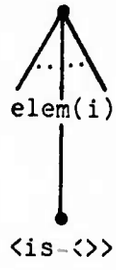


Figure 4.3.1. VDL representation of the tokens of a state.

4.4. Control Structures.

There are seven control directives which recursively define the class of control structures. These are represented in VDL as shown in Figure 4.4.1. The elementary control directives loop and exit are represented in VDL by references to the symbols "loop" and "exit". Identifiers in control directives are represented by references to the corresponding symbols.

There are five expressions. They are represented in VDL as shown in Figure 4.4.2. Four are the invocations of primitive operations (PO's) on 0, 1, 2, and 3 operands. The operands are identifiers, and are represented as just described. The names of the PO's are represented by references to the corresponding symbols. The fifth expression is the compute expression. Its body is a control structure. The placeholders in its declare component are represented by the VDL empty list.

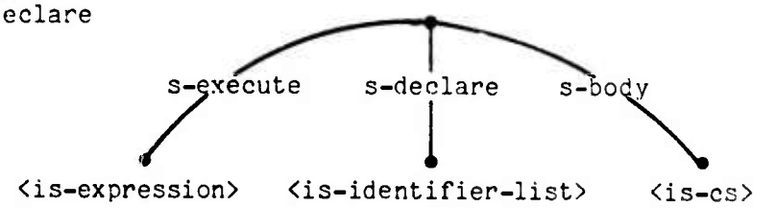
4.5. The Process.

The environments of all the activations are represented by comp(s-processes) of the state (see Figure 4.5.1. In addition, the task of the process is represented by a reference to a functional in the state. Since in this version of the model there is only one process, comp(s-processes) of every state has the form shown.

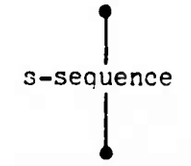
In the initial state of the model, the comp(s-task) of the process is a reference to a functional whose evaluation is the task of the process. Comp(s-activations) of the process is empty. Comp(s-c) of the state is the instruction run-process(1), which is expanded by the VDL state transition rule into instructions that evaluate the functional.

<is-cs>

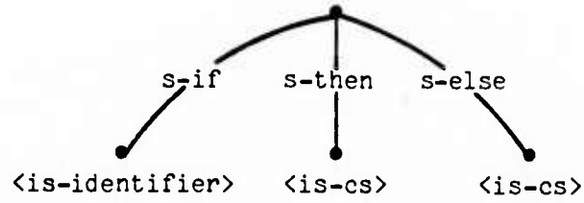
execute/declare



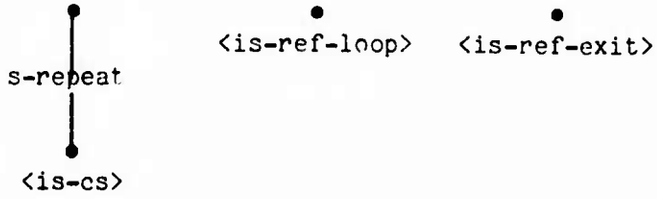
sequence



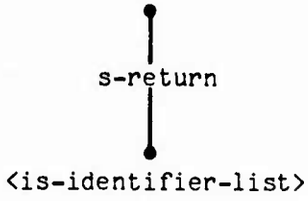
conditional



iteration



return

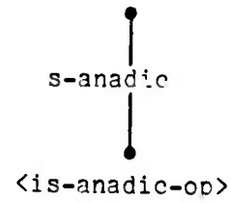


where <is-identifier> is <is-ref> for the corresponding symbol

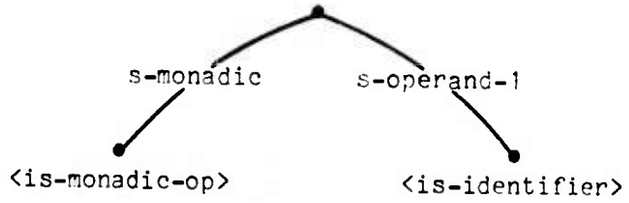
Figure 4.4.1. VDL representations of control directives.

<is-expression>

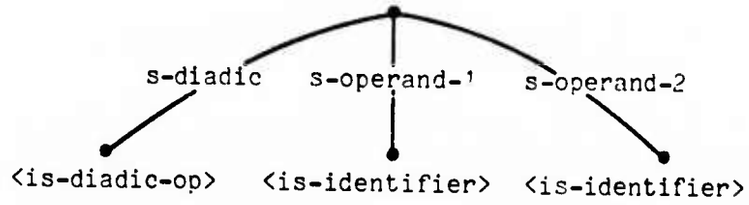
anadic PO



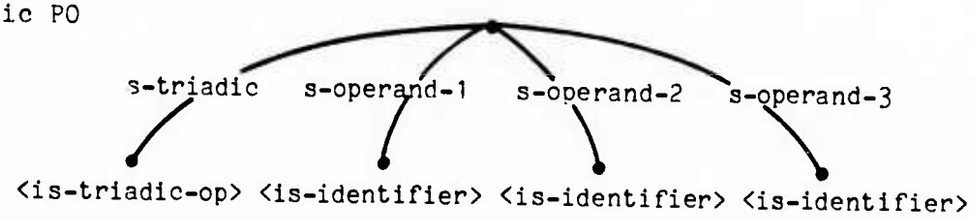
monadic PO



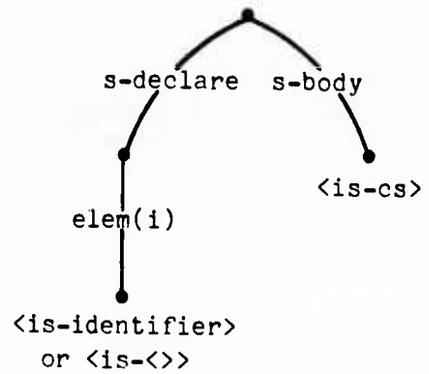
diadic PO



triadic PO



compute



where <is-identifier> is <is-ref> for the corresponding symbol
and <is-xxx-op> is <is-ref> for the corresponding symbol

Figure 4.4.2. VDL representations of expressions.

<is-processes>

<is-process>

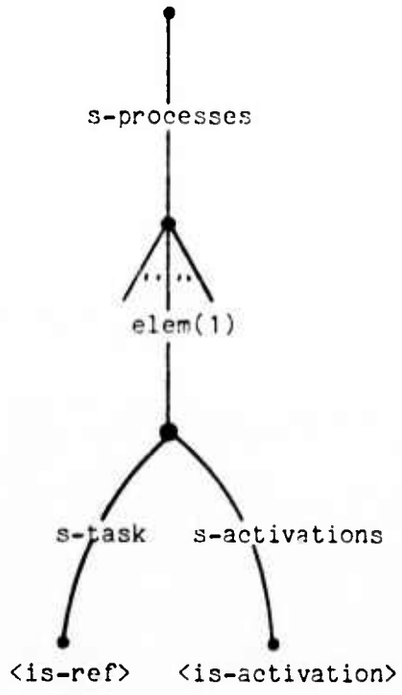


Figure 4.5.1. VDL representation of the process of the state.

4.6. Activations.

Comp(s-activations) of the process is a stack of activations (see Figure 4.6.1). The activation in which the process is actually evaluating is at the top of this stack. Activations have two components. Comp(s-environment) is the environment. It has the same form as the environment of a functional. Comp(s-activation) is another activation which is the return point for the activation. The return point of the oldest activation (the one pushed deepest into the stack) is null.

<is-activation>

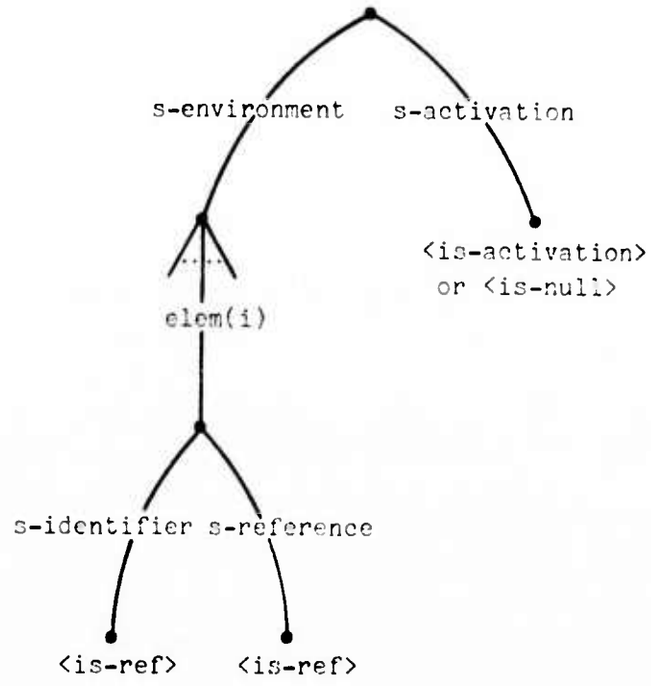


Figure 4.6.1. VDL representation of an activation.

4.7. Result Structures.

<is-rs>

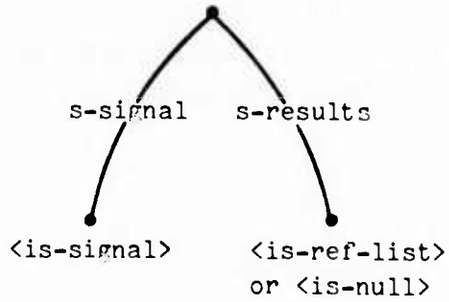


Figure 4.7.1. VDL representation of a result structure.

Result structures are represented as shown in Figure 4.7.1. The signals are represented by elements of the set rtp which is a primitive set of the definition. They appear in comp(s-signal) of result structures. Result lists are represented by lists of references which appear in

comp(s-results) of result structures whose signals are return. On other signals, comp(s-results) is empty.

4.8. Translation.

The correspondence between control structure representations (CSR's) and the VDL representations of control structures is given schematically in Figure 4.8.1. Each evaluation of the PO install includes a translation of a CSR to a corresponding control structure.

The instruction schemata of the definition which effect this translation also perform some checks on the form of the CSR's handed to install. A complete definition would define translation so that all these checks are made. However such a definition would be longer and obscurer than is useful for the purposes of this report. Consequently, the instruction schemata which define translation in the definition given in appendix C are incomplete. To compensate for this, checks which translation could make are in that definition made during evaluations of the control structures resulting from translation. In particular, the appearance of loop or exit control directives other than within the body of an iteration control directive is an error which can be detected during evaluations of install, but is in this definition only detected during evaluations of the resulting control structures. The re-declaration of identifiers is similarly treated.

4.9. Initial state.

The initial state of the model is not uniquely determined. The initial state reflects the programming particular system to be modelled.

Comp(s-processes) of the initial state is assumed to be a single process to be evaluated. This process has a reference to a functional as its s-task component. The process will evaluate this functional without further binding. Therefore this functional defines the system, for it

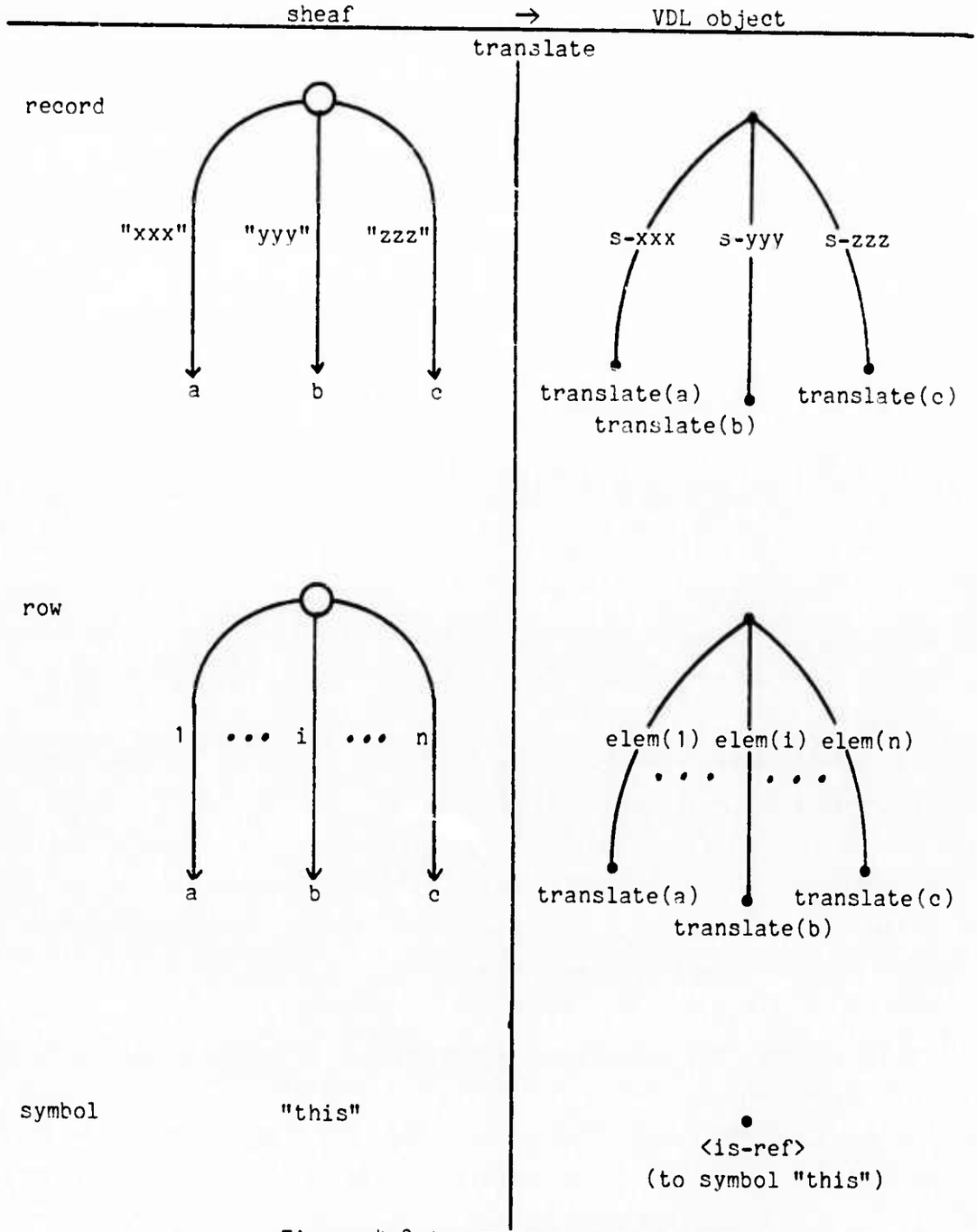


Figure 4.8.1. Scheme of translation.

alone specifies what the interpreter will do.

Comp(s-activations) of the process is null.

Comp(s-items) of the initial state must have those items defined in it which are required to support the task of the process. Therefore it must have at least that functional itself, and usually many other items which that functional references.

Comp(s-tokens) is initially the empty list.

Comp(s-c) of the initial state has the single instruction run-processes(1).

4.10. Predicates.

The definition is based on four primitive sets and a collection of "character-recognizing" predicates (see Section B.12). The sets are the following.

1. The logical values: {true, false}.
2. The integer values: {...,-2,-1,0,1,2,...}.
3. The characters: {a, ..., z, A, ..., Z, -, ., $, ...}. The definition assumes the existence of an ordering relation for this set, which it denotes with "<<" (see lex-sort). The definition also assumes that under this ordering relation, a<<b, b<<c, ..., y<<z.
4. The reference types: {rtplog, rtpint, rtpsym, rtpkey, rtpshf, rtpcel, rtpfnl, rtpbnd, rtptok}. These are used in references to indicate what kind the references are.
5. The signals: {signormal, sigloop, sigexit, sigreturn}. These are used to indicate the nature of result structures.

There are predicates for each of the first two sets in their entirety. There are predicates for each of the elements of the last three sets.

The predicates for characters are needed by the formal definition because the instruction schemata which translate control structure

representations into control structures check for errors in the form of the structures of items handed to install. The forms must have certain symbols as the selectors of sheaves used. Predicates corresponding to these distinguished symbols are derived from those for characters.

4.11. Instructions Schemata.

The instruction schemata of the formal definition of the Binding Model are given in appendix C. The introduction to that appendix explains the notation used there. However, some comments on the style used in the schemata will be helpful.

It is not easy to create understandable VDL instruction schemata. One must read up the page for sequences of activities, and down the page for decisions. Further, it is not possible to include decisions among a sequence of activities; a separate schemata must house each decision. The same is true for activities over sets: they often cannot be included in sequences of activities, but rather must be housed in separate schemata.

Thus it often requires a group of schemata to express what is conceptually a single routine. A naming convention for schemata has been used in the definition to indicate such groupings of schemata into routines. The first schema of the group is given the name of the routine. Subsequent schemata, where they exist, are given a version of the same name formed by adding an integer suffix. Thus, for example, the routine to evaluate an execute/declare control directive is composed of two schemata having names eval-execution and eval-execution-1.

There is not space in this report to give detailed discussions of the instruction schemata which make up the formal definition. However, once a feeling for the style of the definition has been obtained, these schemata should be relatively self-explanatory. Therefore, to help the reader acquire this feeling for style, two routines of the definition—— eval-select and eval-compute—— are now discussed in detail.

the routine eval-select implements the PO select (see Section C.7). It is composed of two VDL instruction schemata. tHe first, eval-select, takes two arguments, both of which are references. It checks that the second reference is a selector (using check-selector) and that the first reference is a sheaf (using get-sheaf). The latter instruction schema returns a VDL object which represents the information content of the sheaf referenced by the first argument. The actions are defined to take place concurrently.

Check-selector appears in Section C.13. It uses the predicate is-sel-ref to check that the reference which it is given is for a selector. If this is tne case, nothing is done. Otherwise, the final state error is made the result of the state transition rule.

Get-sheaf (see Section C.12) first checks that its argument (a reference) is for a sheaf. If it is not, an error is reported. Otherwise, the index of the reference is used to get the representation of the sheaf from the type-list for sheaves. This representation is passed (see Section C.17) as the result of get-sheaf.

The representation of the sheaf referred to by the first argument of eval-select is then searched to find the index of the component which has the second argument to eval-select as its selector.

Search (see Section C.15) takes a VDL list, a VDL selector, and an object. It searches <list> for exactly one element whose <selector> component is <object>. The result of the search is the index of that one element if that element exists, and null otherwise. If two elements exist, search produces an error (VDL's response to the second attempt to supply a local name with a non-null result).

Eval-select then invokes eval-select-1 with a representation of the sheaf, and the index of the selected component. Eval-select-1 checks that the index of the selected component is non-null: a null index indicates that the selector referenced by the second argument to eval-select is not a selector of the sheaf referenced by the first argument to eval-select. If the index is null, an error is reported. If not, the index is used to select from the sheaf the reference to the appropriate component. The

routine return-1-ref (see Section C.16) is used to form a return result structure with the reference to the selected component of the sheaf as the sole element of the return list.

Eval-compute appears in Section C.3. Its routines form all of Subsection C.3.5. The routines are among the most complex in the definition.

The object representing the current environment is obtained using current-environment (see Section C.2.1). It is used to restore this environment after the compute expression has been evaluated.

The declare list is then used to create a list of tokens and placeholders. Get-opprefs (get optional potential references) first checks for the declare list being empty. If it is, the list of optional references is empty. If it is not, get-oppref is used on each element of the declare list. The results are formed into a list and passed back to instructions in eval-compute.

Get-oppref yields a placeholder if the element of the declare list is a placeholdeer. Otherwise (if the element is an identifier), it uses make-token (see Section C.11) to create a new token and provide a reference to it. This reference is returned to the instructions of get-opprefs.

The environment is then altered using declare-set (see Section C.2.1). That routine adds to the environment bindings of any identifiers in the declare list to the corresponding potential references in the list of optional potential references just created.

The body of the compute expression is evaluated in the altered environment using eval-cs (evaluate control structure; see Section C.2). That routine yields a result structure.

To make tests on the nature of this result structure, eval-compute-1 is invoked. The predicates on result structures (see Section B.10) are used. Loop and exit are errors. Normal is treated as a return of zero references. This is effected by forming such a result structure and invoking eval-compute-1 recursively.

The references in the result list of the return result structure are used to determine the tokens referenced in the optional potential reference list. This is done with the routine determine-opprefs which first checks that the lengths of these lists are the same, and then invokes determine-oppref on pairs of corresponding elements of these lists.

Determine-oppref does nothing if the optional potential reference is a placeholder: The corresponding element of the declare list was also a placeholder in that case, so no token was created, and none needs to be determined. If a potential reference is present, then the reference in the result list is checked to insure that it is not a potential reference. If it is not, then a state-wide replacement is made: Wherever in any component of the state the potential reference is found, it is replaced by the regular reference from the return list. This effectively determines the token.

The final acts of eval-compute-1 are to restore the environment using reset-environment (see Section C.2.1), and to pass the result structure from the body of the compute expression to the invoker of eval-compute.

4.12. Some Correspondences.

The discussions in Chapter 5 will be based on the VDL definition, but they will be couched in terms of the constructs and states of the Binding Model. In introducing the VDL definition in this chapter, many correspondences between that definition and the Binding Model have been made clear. However there are a few additional points that need to be clarified in order to properly prepare for Chapter 5.

If in some state, the VDL object representing a reference, either regular or potential, occurs as a subobject of the VDL object representing some part of the model, then that part is said to include the reference. Parts which can include references are items, environments, result structures and instructions (in the VDL control tree).

If an item x includes a reference to an item or a token y, then x is said to directly enclose y.

An item x is said to enclose an item or token y if either x is identical to y, x directly encloses y, or x directly encloses an item which encloses y. That is, the relation "enclose" is the reflexive, transitive closure of the relation "directly enclose". The set of all items and tokens enclosed in an item is called

The detailed discussions of the two routines in the previous section demonstrated that there may be many state transitions of the VDL definition needed to complete an evaluation which is thought of as being indivisible in the Binding Model. It is useful therefore to single out a subset of the VDL states which can be thought of as corresponding to, and defining, the states of the Binding Model.

Comp(s-c) of a VDL state has in many cases just a single instruction which can be evaluated next. The distinguished set of states is defined to be those in which comp(s-c) has this form, and the instruction to be evaluated is from a restricted set of instruction schemata. That restricted set is: all the eval-xxx routines for control structures, control directives, expressions, and PO's, plus the routines declare-set, reset-environment, and determine-opprefs. These routines define the major activities in the Binding Model.

Another observation which must be made in preparation for Chapter 5 is that although each state can be considered to have its own collection of items, it is possible to view the collections of items in a state as being produced by leaving alone, or adding to or changing the information content of items in, the collection of items in the preceding state. Thus items can be regarded as having identity over a number of states in the model.

The VDL definition provides an easy way of formalizing this notion of identity: an item in some state of the model is identical to same item in another state of the model if and only if references to those two items in the two states are equal VDL objects.

The final point concerns mutability. There are only two instruction schemata in the definition which can change items existing in a state: change-contents and determine-oppref. The former is part of the routine eval-update which defines the PO update for cells. Thus cells are mutable.

The latter instruction schema is part of the routine eval-compute. It is used to determine a token. This is done by replacing potential references to the token with regular references to the item which is the token's determination. It might therefore appear that any item which includes a potential reference is mutable.

This is not so, for to show that an item is mutable it must be possible to find some PO which, when invoked twice on that item, yields different results. However, any attempt to access an item through a potential reference is an error. Therefore the only possibility for different results from a PO using an item which includes a potential reference is the difference between its yielding the potential reference itself and, latter, the regular reference which replaced it. But the very act of determining the token which permitted the difference to occur also rendered the difference unobservable, for the (earlier) potential reference was replaced by the same regular reference against which it was later to be compared.

Thus cells are the only mutable items in the model.

Chapter 5. Specially Behaved Functionals.

In the Chapter 3, the wide variety of behaviours which functionals can exhibit was indicated. This chapter discusses some restricted classes of behaviours which are useful in supporting modular programming.

In the model, functionals are used to implement programmer-defined operations. The discussion of this chapter, therefore, centers around the behaviours of functionals. Two special classes of behaviour are of interest: repeatable behaviour and sequence-repeatable behaviour. Intuitively, a functional is repeatable if it does the same thing every time it is invoked. That is, when it is given similar inputs it yields similar outputs. A functional is sequence-repeatable if it is repeatable when considered as a mapping from sequences of inputs to sequences of outputs. That is, two copies of the functional will produce sequences of pairs of similar outputs when given sequences of pairs of similar inputs.

Intuitively, a repeatable functional is sequence-repeatable. The converse is false. Consider, for example, a pseudo-random number generator which stores its seed. Two copies of this functional will generate similar sequences of random numbers. However successive invocations of the same functional with similar arguments (none) produce non-similar outputs.

In what follows, repeatable and sequence-repeatable functionals are referred to collectively as being <u>specially</u> <u>behaved</u>.

The specially behaved functionals are of interest because their behaviours are often somewhat easier to understand than the behaviours of functionals which are not specially behaved. In particular, test invocations of specially behaved functionals can be used to reliably predict the behaviours of those functionals on later invocations. This is helpful in debugging.

To the programmer, functionals represent operations which take representations of instances of primitive and programmer-defined types as inputs, and yield representations of instances of such types as outputs. Thus to be most helpful to the programmer, items should be considered "similar" for the purpose of defining the special behaviours exactly when they represent instances which he considers "equal" instances of programmer-defined types. However there is no way in the model for the programmer to express his conceptions of "equal". Consequently any definition of similar based on the model cannot take these conceptions into account. Means for programmers to express such conceptions are a matter of current research. Adding such mechanisms to the model is therefore beyond the scope of this research and will not be discussed further here.

However it is still possible to be partially responsive to the programmer's needs by finding a definition of "similar" under which similar items are very likely to represent instances which the programmer considers to be equal. If this can be done, the programmer can test for the special behaviours at an abstract level in two steps: he can test for them at the model level, and then he can check them for the (hopefully few) cases in which dissimilar items are used to represent what he thinks of as equal instances. In many situations, the second case will not arise in which case the definition of special behaviours based on the model will satisfy a definition based on user abstractions.

This chapter gives a model-based definition of similar motivated by our model-based intuitions about the special behaviours and by the intuitions about "equal" designed into the model. The notion of "extending similarities" is then introduced. The special behaviours are defined in terms of extending similarities.

The concepts and definitions of the special behaviours add structure to the use of the model. In doing so, they help the programmer in his task of creating functionals in the model. For example, he can characterize his functionals as being or nor being specially behaved; he can set out with special behaviour as a goal.

An additional and more concrete aid to the programmer would be knowledge of a set of conditions sufficient to guarantee that a functional be specially behaved. These conditions would be tests on the structure of, and the existence of references to, a functional and its closure.

Such conditions are developed in this chapter.

5.1. The Intuition of Repeatability.

If elementary items were the only ones permitted in inputs and outputs of functionals, "similar" could be defined to mean identical. Also the definition of a repeatable functional[1] would be easy: when bound to identical inputs on two different occasions, the functional would in both cases either produce identical output items, detect errors, or evaluate forever.

However, the inclusion of non-elementary items, particularly cells and functionals, in inputs and outputs necessitates a more sophisticated definition. Some examples are now given which illustrate those aspects of the model which intuition says must be taken into account when framing definitions of similarity and repeatability. The examples manipulate rows of cells (see Figure 5.1.1).

First, identity must be rejected as a definition of similarity. Consider a functional CREATE which takes an integer and yields a new row having that integer number of cells each with contents 0. Given identical integers it always yields rows which are look alike. However the rows are not identical, for from each invocation new items result.

Intuition says that functionals like this which create non-elementary objects are repeatable. Thus repeatability must not require that outputs

---

1. In this discussion, repeatability alone is considered. This simplifies the discussion without reducing the base of the intuitions in any important way, for the intuitions about sequence-repeatability are just the intuitions about repeatability applied to sequences of invocations.

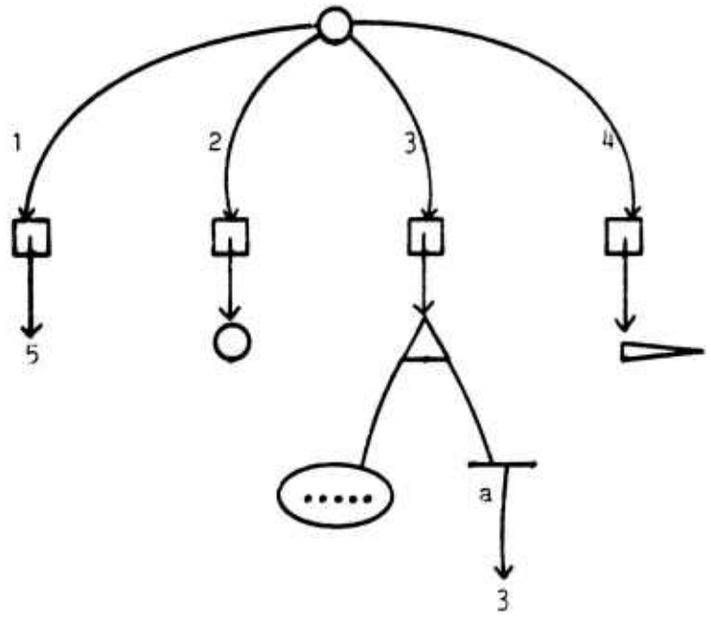


Figure 5.1.1. A row of cells.

be identical, but rather only that they look alike. The term "similar" will be a precise sense of "look alike". It will be defined formally later in the chapter. For now, the intuition is: structurally the same but not necessarily identical.

Another intuition about repeatability is that a functional which simply invokes repeatable functionals in sequence should itself be repeatable. If this "sequence requirement" is to be satisfied, it must be demanded of repeatable functionals that they guarantee in their outputs any conditions which they require of their inputs.

Consider the requirement that functionals produce similar output, for example. If similarity is all that can be guaranteed for outputs of repeatable functionals, then it would be nice if similarity of inputs were all that were needed to guarantee it. For example, consider a functional DIG which yields the contents of the third component of a row. Suppose the functional CREATE yields similar but not identical outputs. If identical inputs were required for repeatability, then no conclusions could be drawn about the similarity of outputs when DIG is applied to the outputs of CREATE.

So repeatability should demand that for all similar inputs, similar outputs are yielded.

Other intuitions concern the presence of cells in inputs. The contents of cells which are enclosed in the inputs to a functional may be changed by that functional during its evaluation. If this happens the functional is said to have side effects.

Consider a functional which takes a row, an integer and an item as inputs, and updates the component of the row selected by the integer with the item. The functional has no output. It is evaluated for its effect. Because those effects on similar inputs are similar, it is intuitively repeatable. To encompass this intuition, the definition of repeatable must take side effects into account.

Side effects can be taken into account in part by demanding that the inputs to invocations of a repeatable functional which are similar to each other before the invocations should still be similar to each other after it. However the side effects of a functional could involve "disconnecting" part of one of its inputs. If this happens, checking that the inputs to two invocations are similar after those invocations does not check the final similarity of the disconnected pieces. Because the invoker may have references to those pieces, it is important that repeatability require that all the items enclosed in the inputs to invocations be left similar after those invocations.

Updating a cell has the potential for affecting operations on any item enclosing that cell. Consequently, patterns of references to cells are an important aspect of the similarity of items. For this reason, repeatable functionals are only required to behave similarly on items whose patterns of references to cells are similar. Because of the importance of patterns of references to cells, the term sharing patterns is introduced for this notion.

Consider, for example, a functional DECREMENT which manipulates a row of cells, all of whose contents are integers. It decreases the contents of each component by 1, and yields true if any of the resulting integers is

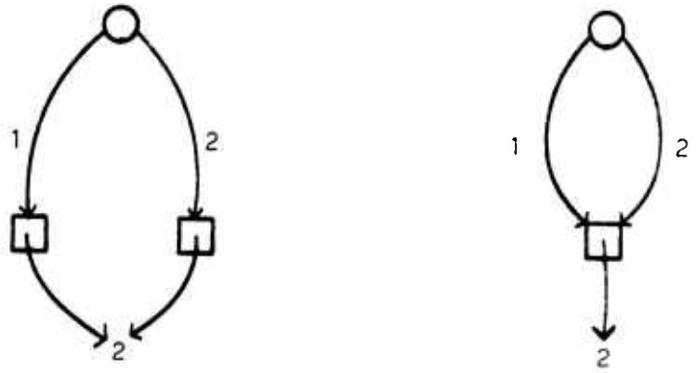


Figure 5.1.2. Different sharing patterns.

zero. Suppose a single cell can act as more than one of the components of the row (two selectors "share" the cell). If sharing patterns are disregarded, the two rows of Figure 5.1.2 are similar. Notice the difference in behaviour of DECREMENT on these two similar(?) inputs. On the first, the altered row has two 1's; on the second, the single cell has contents 0. Consequently DECREMENT will yield different logical items as results when applied to these inputs.

Further, note that when a functional expects more than one item in its input, relationships among those items are also of importance. Even if all pairs of individual items have similar sharing patterns, the patterns of the sets may differ. Thus similar sharing patterns must exist among the entire collections of items in two inputs before a repeatable functional is expected to produce similar outputs when applied to them.

As sharing patterns are of concern in inputs to functionals, by the sequence requirement, sharing patterns in outputs and in the changed inputs resulting from side effects are also important. All these intuitions can be accomodated by making the notion of similarity more sophisticated so as to encompass a demand for similar sharing patterns.

Sharing patterns between inputs and outputs are also important. Consider a functional which searches a row for a component cell whose contents is a particular symbol and yields that cell as its output. Given similar rows each with more than one component having the searched-for symbol as contents, a repeatable functional should yield outputs which

share in the same way with the inputs. Then an employer of the functional can be sure that similar updates to the returned cell will cause similar changes to the row which was used as input.

The input to functionals may also enclose other functionals. If a functional invokes a functional which is enclosed in its input, then the behaviour of the invoking functional becomes dependent upon the behaviour of the input functional.

For example, consider a functional which takes a row and another functional as input, and applies the input functional to the contents of each component of the row in turn. It yields a new row consisting of the results of all the applications of the input functional. (This is LISP's Maplist). Intuitively this functional is repeatable. Yet if the input functional is not repeatable, the output similarities may not be guaranteeable. (Many other examples occur when procedural encapsulation is used to represent instances of programmer-defined types (see Section 3.8)).

Consequently, intuition dictates that a repeatable functional not be held responsible for non-repeatable behaviour introduced by invoking input functionals.

The role of keys in the model is to provide identity which cannot be counterfeited. It is therefore expected that actions of functionals may depend upon the identity of keys in their input.

For example, a functional might check that the contents of the first component of a row is a particular key. The functional's behaviour could be radically different depending on whether the key is there or not.

Consequently, repeatable functionals are not required to behave similarly if the keys enclosed in the inputs are not identical.

Finally, functionals do not exist in vacuums. Rather, there is the possibility that there are "outside" references to items which they enclose. It must be assumed that these "surroundings" are hostile. That is, it is assumed that all outside references will eventually be exploited in any way possible. If something can go wrong, it will.

Intuition therefore dictates that the certification of a functional as being specially-behaved be done with full awareness of, and in spite of, the assumed hostile surroundings in which it exists. Therefore, in comparing the behaviour of functionals, it is only fair that they experience equivalent outside pressures. Similar functionals which are more exposed to their surroundings may not be specially-behaved, and others which are less exposed may be specially-behaved only by virtue of that seclusion.

The next few sections of this chapter give formal definitions for notions in terms of which these intuitions about repeatability can be stated precisely.

5.2. Similarity.

As noted in Chapter 2, the items in a state can be thought of as forming a directed graph. The terminology resulting from that concept was introduced formally at the end of Chapter 4. By way of review, Figure 5.2.1 tabulates direct enclosures by item type.

type	items directly enclosed
logical	none
integer	none
symbol	none
key	none
sheaf	selectors, components
cell	contents
functional	items referenced by environment
bundle	extractors, components

Figure 5.2.1. Direct enclosures, by item type.

Definition: The closure of an item x is the set of all items which x encloses.

Notation: The set of cells in closure(x) is denoted cells(x). The set of

tokens in closure(x) is denoted tokens(x).

The following definitions formalize the notion of "similar" which was introduced informally in the previous section. In essence, they state that two items are similar if they have the same structure.

Definition: Two items, a and b, are said to match if they are of the same type and

1. they are elementary and identical, or
2. they are keys and identical, or
3. they are sheaves or bundles, and
   a. they are the same length, and
   b. they have identical selectors (extractors), or
4. they are functionals, and
   a. their control structures are equal as VDL objects[2], and
   b. they have the same set of bound externals.

Any two tokens match. A token and an item never match.

Definition: Let u and v be matching items, including references r and s respectively. Then r and s are said to correspond if

1. u and v are cells, or
2. u and v are sheaves or bundles, and r and s are associated with identical selectors in u and v, or
3. u and v are functionals, and r and s are affiliated with the same identifiers in the environments of u and v.

Definition: Two items x in state R and y in state S are said to be similar if there exists a relation F in closure(x) X closure(y) such that

1. xFy (that is, <x,y> is in F), and
2. uFv implies
   a. u and v match, and
   b. if r and s are corresponding references in u and v, then either

---

2. The mention of VDL objects may seem at the wrong level for this discussion. However, it will turn out that part 4a of this definition is equivalent to the requirement that the control structure representations used with install to create the control structures were similar.

i. r and s are both potential references, or

ii. r and s are both regular references, in which case if a and b are the items referred to by r and s, then aFb, and

3. F restricted to cells(x) X cells(y) is one-to-one, and

4. F restricted to tokens(x) X tokens(y) is one-to-one.

Any two tokens are similar. A token and an item are never similar.

The relation F is said to establish that x and y are similar. As the contents of cells are state-dependent, similarity is a state-dependent notion. Note that similarity is an equivalence relation. Fig 5.2.2 depicts some pairs of items which are similar to one another.

Definition: Suppose F is a relation between items and tokens of state R and items and tokens of state S. Then F is called a similarity in R and S if, a) for every x in R which is in the domain of F, there is a y in S such that xFy, and F establishes that x and y are similar, and b) (symmetrically) for every y in S which is in the domain of F, there is a x in R such that xFy, and F establishes that x and y are similar.

If a similarity establishes that x and y are similar then F is said to cover x and y.

Definition: A similarity F in states R and S is a subsimilarity of another similarity G in states R and S if the domains of F are subsets of the domains of G and, when considered as sets of ordered pairs, F is a subset of G.

Definition: The intersection of a set of similarities in states R and S is a relation which is the intersection of the members of the set (considered as sets of ordered pairs), and whose domains include only those items and tokens which appear in those ordered pairs.

Lemma: The intersection of a set of similarities in states R and S is the largest similarity which is a subsimilarity of each member of the set.

The proof of this lemma is an elementary exercise in set theory and is therefore omitted.

Definition: If two items or tokens are similar in states R and S, then the

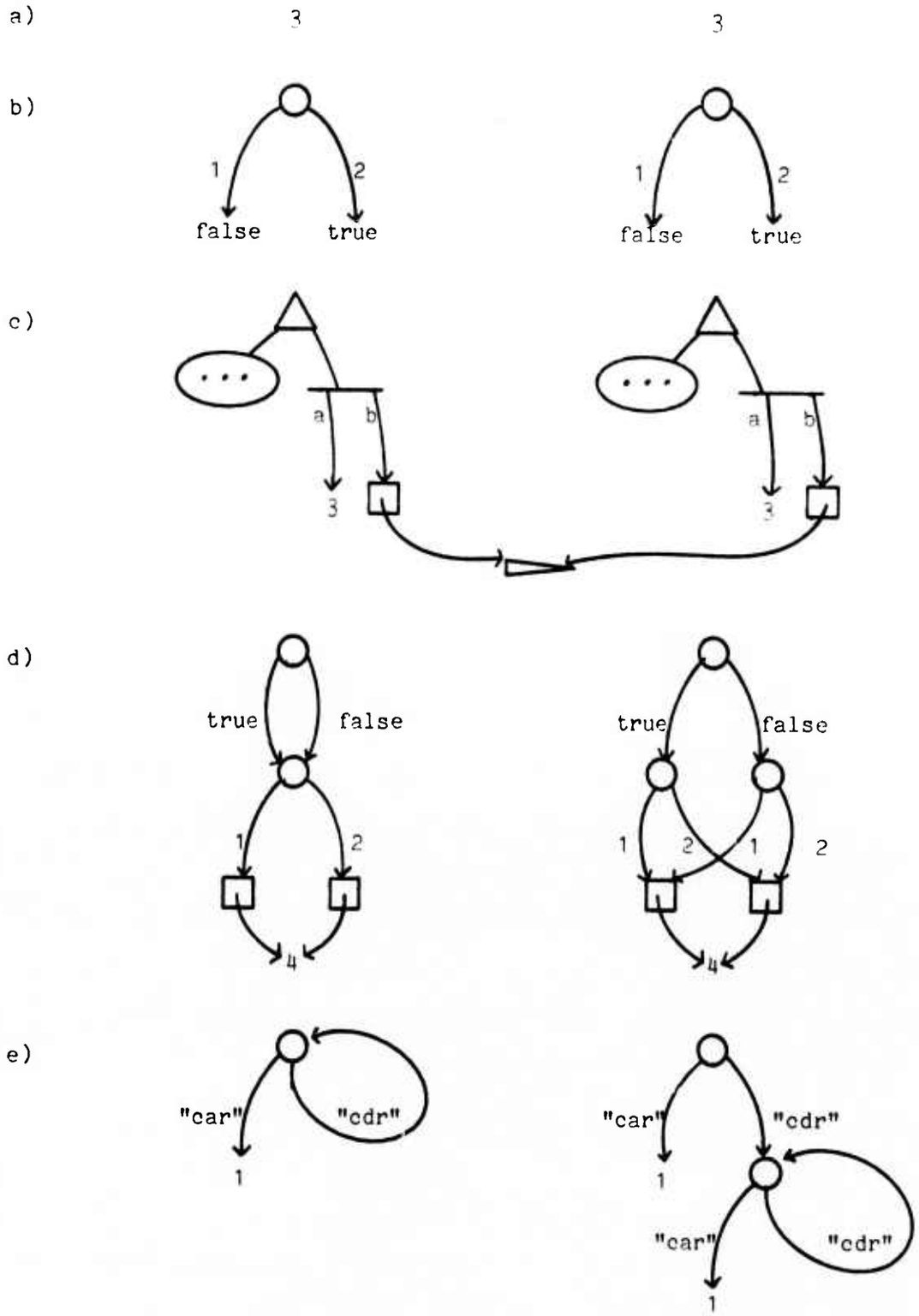


Figure 5.2.2. Pairs of similar items.

intersection of all similarities which establish the similarity of x and y is called the minimal similarity covering x and y.

In the previous section, the "inputs" and "outputs" of a functional were discussed informally. The notion of input is based on the idea (presented in Section 2.8) of "calling a functional with certain items as input". The output is the collection of items or tokens returned by such a call. In what follows, extensive use will be made of these notions. They must therefore be introduced formally. To do this, named collections of items and tokens are introduced. They will be called "packages"; the names used in their formation will be called "tags".

Definition: A tag is either a symbol or a distinguished entity denoted by *.

Definition: A package is a partial function from the tags to the items and tokens in some state of the model. An item or token in the image of a package is called a constituent of that package.

The intended usage of the distinguished tag * is to label the activity specification being invoked. It is special so that there can be no conflict with symbols used as tags.

Now the notions of closure, matching, corresponding and similarity can be extended to packages.

Definition: The closure of a package is the union of the closures of all constituents of the package.

Definition: Two packages match if they are defined for the same set of tags.

Definition: If p and q are two packages and r and s are constituents of p and q respectively, then r and s are said to correspond if they are the images of the same tag.

Definition: If p is a package in state R and q is a package in state S, then p and q are similar if p and q match and there exists a relation F

on closure(p) X closure(q) such that, for each constitutent x of p and corresponding constituent y of q, F establishes the similarity of x and y.

The relation F is said to <u>establish</u> that p and q are similar. Also if F is a similarity in R and S which establishes that p and q are similar, F is said to <u>cover</u> p and q.

Defintion: If p and q are similar packages, then the intersection of all similarities which establish that p and q are similar is called the <u>minimal similarity covering p and q</u>.

To complete this section, the notion of similarity is extended to environments by defining a package based on an environment.

Definition: The <u>derived</u> package of an environment is the mapping of symbols to items or tokens defined by letting each affiliation in the environment define a pair in the mapping.

Definition: Two environments are said to be <u>similar</u> if their derived packages are similar.

5.3. Extending Similarities.

The previous section gave formal meaning to the notions of similar items and similar groups of items. It would therefore now be possible to formalize repeatability as "similar inputs guarantees similar outputs". Such a definition would be inadequate because it would not capture all the intuitions discussed in section 5.3. In particular, it would not express the intuitions about side effects and patterns of sharing between inputs and outputs. It is the purpose of this section to introduce the idea of "extending" a similarity in terms of which these intuitions can be expressed.

Including items in a package permits discussion of sharing patterns among them. Therefore one is tempted to use the output of a functional to expand the input package, and then to demand of a repeatable functional

that it produce similar expanded packages from similar input packages. Because inputs are included in the expanded package, the possibly changed inputs are required to be similar.

This definition of repeatability is also inadequate, for it does not express the intuition about repeatability of functionals whose side effects involve disconnections of parts of inputs. If the disconnected parts are altered dissimilarly, intuition says that the functionals are not repeatable while this definition says they are.

The intuitions can be captured properly, however, by talking about extending, not the input packages, but rather the similarity which covers those packages. This extension is effected by adding items, including those in the closures of the output packages, to the domains of the similarity, and adding ordered pairs of items to the relation itself. It is then demanded of a repeatable functional that when invoked on similar inputs it should behave in such a way that an extension of the input similarity can be found which is a similarity and covers the output packages of the invocations. Notice that requiring the extensions to be similar says that all side effects leave similar those items which were similar under the input similarity.

The idea of extending similarities is actually more broadly applicable than just to functionals. In particular, the discussion of this chapter is concerned with the circumstances under which primitive operations (PO's) and control structures can be shown to extend similarities on their inputs. These results are then used to find conditions under which functionals extend similarities.

The term activity specification will be used to refer collectively to PO's , control structures and functionals. This term reflects the idea that the process of the model regards these entities as specifications for activities. The notion of an "activity" is a sequence of states of the model which record the evaluation of an activity specification.

Definition: An interval is a set of adjacent states in the interpretation of the model.

Definition: An activity is an interval in which the an activity specification is evaluated.

Definition: Let p in R and q in S be two similar packages, and let F be a similarity in R and S covering p and q. Let f and g be two activity specifications for which p and q are acceptable as inputs. Let p′ in S′ and q′ in R′ be the output packages produced by the activities of evaluating f on p and g on q. If there exists a similarity F′ in R′ and S′ which is a relational extension of F and which covers p′ and q′, then it is said that f and g extend F on p and q.

Notice that this definition concerns a pair of activity specifications. This generality is introduced because the results of this chapter usually guarantee the production of similar output from similar inputs. Thus it is natural to think in terms of pairs of similar functionals. For example, pairs of functionals in similar inputs of a pair of applications may be invoked in the understanding that, as a pair, they will extend the similarity covering the environments of the activations associated with those applications.

Definition: If p and q are similar input packages for activity specifications f and g, and if F is a similarity covering p and q , and if f and g extend F, then the intersection of all similarities which establish this extension is called the extension of F by f and g on p and q.

Definition: If p and q are similar input packages for activity specifications f and g, and if F is a similarity covering p and q, and if F′ is the extension of F by f and g on p and q, then f and g are said to creatively extend F on p and q if all items added to the domains of F to arrive at the domains of F′ were created by the activities of f on p and g on q.

There is no mention here of creating tokens. The following lemma shows why.

Lemma 5.3.1: The set of tokens created during the activity of an activity specification is equal to the set of tokens determined during that

activity.

Proof: PO's do not affect tokens.

In a control structure, tokens are created and determined on entry to and exit from compute expressions. The semantics of compute expressions guarantees that all the token created are determined.

The activity of a functional is the evaluation of a control structure together with invocations of other functionals. By the argument just given, control structure evaluations determine exactly the tokens they create. So recursively, all functionals do.

The following result plays a critical part later in establishing that if PO's extend similarities then control structures also do. It is included here to demonstrate the usefulness of the idea of creative extension.

Theorem 5.3.1: Let p and q in R be similar packages covered by a similarity F. Let G be subsimilarity of F which also covers p and q. Let f and g be activity specifications which creatively extend G on p and q. Then f and g creatively extend F on p and q.

Proof: F can be extended by separating it into two relations with possibly overlapping domains. One part is G. The other part which will be called H is the difference of F and G (all three considered as sets of pairs). The domains of H are the minimal ones dictated by its pairs. Clearly F is the union of G and H.

Let G' be the extension of G by f and g on p and q. Let F' be the union of G' and H (all three considered as sets of pairs). F' will now be shown to be a similarity which extends F. Thus it will have been shown that f and g extend F on p and q.

Since G' extends G, F' extends F. It muust be shown, however, that F' is a similarity. Suppose xF'y. Then either xG'y or xHy. In the former case, the fact that G' is a similarity establishes that x and y are similar under F'.

The latter case (xHy) takes more thought. Let R' and S' be the final states of the activities of f on p and g on q. If x is a token, then so is y. By lemma 5.3.1, the activities of f and g cannot have

determined x or y, so they remain similar. Now consider items. Firstly, suppose x is not a cell. Then y is not a cell (because they were covered by similarity F). Then x and y are immutable. So if v is directly enclosed in x in R′, and w is correspondingly directly enclosed by y in S′, then that was also the case in R and S. But because F covered x and y, it follows that vFw (by the definition of similarity). Because F′ extends F as relations, vF′w.

Now suppose x is a cell. Then y is also a cell. It can be concluded that neither x nor y can have been in a domain of G by the following argument: Without loss of generality, suppose that x is in a domain of G. Because xHy, then xFy. Because F is a similarity, it is one-to-one on cells. But G is a subsimilarity of F with a domain containing x. Therefore xGy. Therefore it cannot be that xHy (by construction of H). But this is a contradiction.

As neither of the cells x or y is in a domain of G, neither of them could have been updated by the actions of f and g on p and q. Thus their structure in R′ and S′ is as it was in R and S. Therefore the argument just given for non-cell items covers these cells. To show that F′ on cells is one-to-one, note that the previous argument shows that no cell is in a domain of both G and H. So G and H are both one-to-one on cells. Because G′ is a creative extension of G, no cell is in a domain of both G′ and H. Therefore as both G′ and H are one-to-one, so is F′. Every item in a domain of F′ is either in a domain of G′ or in a domain of H (and therefore F). Consequently, F′ covers all items in its domains. So F′ is a similarity.

Since H is not changed between R/S and R′/S′, and since G′ is a creative extension of G, then F′ is a creative extension of F.

The rest of this section discusses those activity specifications which extend similarities on all input packages. It is shown that most PO′s extend similarities covering packages of operands, and that all control structures which restrict themselves to these PO′s extend similarities covering packages derived from environments.

However, functionals do not in general extend similarities. It is the work of the rest of this chapter to define the special behaviours by

specifying the cases in which functionals exhibiting those behaviours are required to extend similarities on pairs of inputs, and then to find some structurally-defined classes of functionals which satisfy these definitions.

The first order of business is to decide which PO's of the model extend similar packages of operands. To do this, the packages manipulated and the activities specified by PO's must be defined.

Definition: PO's take from zero to three operands. The input package of a PO is a mapping from the appropriate subset of the set of tags {"operand-1", "operand-2", "operand-3"} to items or tokens. The activity specified by an invocation of a PO with a particular package as input is the evaluation of that PO using as its first operand the item or token tagged in the package with the symbol "operand-1", etc. PO's yield from zero to two items or tokens. The output package for a PO is the package which maps the appropriate subset of the symbols "result-1", "result-2", etc. into those items or tokens.

Thus for example, the PO divide has an input package defined for "operand-1" and "operand-2", and has an output package defined for "result-1" and "result-2".

Theorem 5.3.2: All PO's except eval, identical and new-key creatively extend the minimal similarities covering each pair of similar inputs .

Proof: By case analysis on the PO's.

1. not, and, or, add, subtract, multiply, divide, less-than, equal: These PO's take elementary inputs and yield elementary outputs. As similar means identical for elementary items, the outputs for similar inputs will be identical. If the output item(s) is (are) not already in the domains of the input similarity, then create an extended relation by extending each domain to include the item , and augment the relation by defining the item to be related only to itself. If the output item(s) is (are) already in the domain of the relation, the input relation will serve as the output relation.
2. is-logical, ..., is-bundle: These are the type-testing PO's. If a

pair of inputs is similar, then they have the same type. Consequently the logical items which are the outputs of the PO will be identical. The output relation can be created as described in 1.

3. decompose, empty-sheaf, augment, new-cell, install, bind, new-bundle: These PO's create new items. The input similarities are extended by adding the new items to the domains and relating them to each other. To show that the resulting relation is a similarity, the items directly enclosed in the resulting items must be shown to be appropriately related.

3a. decompose: As the input is elementary, similar means identical. The resulting items are sheaves with equal lengths, identical integer selectors and identical symbol components.

3b. empty-sheaf: The results have no directly enclosed items.

3c. augment: If the inputs are covered by a single similarity, then the outputs will be covered by the same similarity provided that the new component is "inserted" in the ordering at the same place in both cases. Similar for selectors means identical, and the sorting for identical items always achieves the same ordering. If the new selector is already a selector of the input sheaves, then augment will detect an error in both cases.

3d. new-cell: The contents of the pair of new cells are the only items which those cells directly enclosed. Those contents are in the inputs to new-cell. As inputs are similar, so are the contents of the resulting cells. Because a single related pair of cells has been added to the input similarity, the output similarity is one-to-one on cells.

3e. install: The nature of the translation process is to take similar control structure representations (sheaves or symbols) into equal VDL objects. For similar inputs, install yields functionals with equal VDL objects as control structures. The functionals resulting from invocations of install have no directly enclosed items.

3f. bind: If the functionals in the inputs are similar, then their control structures are equal VDL objects. Hence the control structures of the outputs will be equal VDL objects. If the symbol to be bound is already bound, then errors will occur in both cases. Otherwise the yielded functionals will be similar under the similarity of the input.

3g. new-bundle: The outputs have the same structure as the inputs. Consequently they are similar under the input similarity.

4. is-selector, is-extractor: These PO's return logical items. For sheaves and bundles to be similar, their selectors and extractors must be similar. For selectors to be similar, they must be identical. Therefore for similar inputs, these tests will yield identical, and therefore similar, logical items. The input similarity is conditionally extended as in part 1 of this proof.

5. select, selector, contents, extract: These return items which are enclosed in their inputs. Consequently, the input similarity establishes the similarity of the outputs.

6. length: Similar sheaves have the same structure, and consequently the same length. So on similar inputs, the outputs will be identical (and therefore similar) integers. The input similarity can be extended as in part 1 of this proof.

7. compose: Similar inputs are rows of symbols. So selectors and components are elementary; to be similar they must therefore be identical. Thus from similar inputs, identical, and therefore similar, symbols result. The input similarity can be extended as in part 1 of this proof.

8. equal, same: These are the identity tests for restricted domains. Equal is for elementary items and keys. For these types similar means identical, so identical (and therefore similar) logical items result. Same is for cells. The similarity on these items must be one-to-one. Thus identical (and therefore similar) logical items result. The input similarity can be extended as in part 1 of this proof.

9. update: This PO has no output, but it does have a side-effect on its input. Consequently the input similarity will serve as an output relation, but it must be established that it still establishes similarity. The side effect is in changing the contents of the cells which are input. But the new contents were also in the input, and therefore were and are covered by the input similarity. Consequently the outputs are similar under the same relation.

In this case analysis, the only additions made to the domains of the minimal similarity on the input packages are items created by the

action of the PO. Consequently the extension of the minimal input similarity is creative.

The next step is to examine control structures. Again, the details of input packages, activities specified, and extension to output packages must be defined.

Definition: A control structure causes side effects on environments and yields result structures. The input package is the package derived from the environment existing when evaluation of the control structure begins. The activity specified by a control structure with a derived package as input is the evaluation of that control structure in the environment from which the package was derived. If the result structure signals anything but return, the output package is the empty mapping. In the case of a return result structure, the output package maps some of the tags "result-1", "result-2", etc. into the members of the return list.

Definition: A simple control structure is one having no occurrences of the PO's identical, eval or new-key.

Theorem 5.3.3: For every pair of similar input packages, a simple control structure extends the minimal similarity on those packages and yields two signals which are the same.

Proof: Control structures used in constructing simple control structures must be smaller than those constructed, and must themselves be simple. So an induction on the size of control structures can be used.

A. Induction statement: As in the statement of the lemma.

B. Induction base:

B1. exit: This control directive (CD) computes no new items, so the input similarity will serve as the output similarity. The control signal returned is always sigexit.

B2. loop: Like exit, except that the control signal returned is always sigloop.

B3. return: This CD also does not affect the environment. The references returned have affiliations in the environment, so the input similarity

covers them already. The control signal returned is always sigreturn.

C. Induction step:

C1. sequence: Three cases must be explored.

C1a. If the length of the sequence is zero, then the environment is unaffected, and the input similarity will serve for the output. The control signal is always signormal.

C1b. If the sequence has only one element, then the induction hypothesis for that element suffices to prove the case for the sequence.

C1c. If the sequence has more than one element, a supporting induction on the length of the sequence is needed.

C1c1. Sub-induction statement: Assuming the main induction statement for elements of a sequence, then after the pair of evaluations for the n-th element of the sequence, the signals from those evaluations are the same, and the similarity covering the inputs to the sequence can be extended to cover the packages derived from the then-existing environments and the return lists (if present).

C1c2. Sub-induction base: The main induction hypothesis proves that after 1 element has been evaluated the sub-induction statement holds.

C1c3. Sub-induction step: Assume that the first n-1 elements of the sequence have been evaluated. By the sub-induction hypothesis, the evaluations of these n-1 elements of the sequence satisfies the main induction statement. Therefore the control signals returned are equal. If the control signals resulting are not signormal, then the evaluation of the sequence is finished with equal signals returned. On the other hand, if the control signals are signormal, the n-th element is evaluated. By the sub-induction hypothesis, the evaluation of the first n-1 elements of the sequence has provided an extended similarity covering the environments as they exist after the execution of n-1 elements. By the main induction hypothesis, the execution of the n-th element will extend that similarity, and will result in the same control signals. This extension is (by transitivity) an extension of the similarity which covered the inputs to the sequence. By this induction, there exists a similarity which extends the initial similarity and covers

the environments at the time the evaluation of the sequence is finished. Furthermore it covers the return list (if any). Also the signals are the same.

C2. iteration: This CD also requires a supporting sub-induction. The statement and argument are similar to that given for sequences, and therefore will not be included here. The only difference isin the stopping conditions: another iteration is started if the signal returned by the evaluatin of the body is either signormal or sigloop. Sigexit or sigreturn terminate the iteration. The signal returned by the iteration is signormal in the case of stopping because of a sigexit, and is the result of the body when stopping because of sigreturn.

C3. conditional: The evaluation of the identifier will yield similar items. If they are not logical items, then errors will be detected in both cases. Otherwise they will be identical, and consequently the same component control structure will be executed in both cases. The effect of the conditional is the effect of that component chosen, which by the hypothesis satisfies the main induction statement.

C4. execute/declare: This is proved in two steps: it is shown that the execution of the expression and the declaration extend the environments similarly, and then that the evaluation of the body in the extended environments extends the extension to give the desired result.

C4a. When the expression of an execute/declare expression is a PO invocation, two earlier theorems must be used. It has been shown (thoerem 5.3.2) that the PO's found in simple control structures creatively extend the minimal similarities covering pairs of inputs. Also these similarities are subsimilarities of that establishing the similarity of the environments. Thus (by theorem 5.3.1) the executions of one of the PO's of simple control structures in similar environments creatively extends the similarity of those environments so as to cover the items yielded by those executions.

C4b. When the expression of the CD is a compute expression, the environments are augmented with affiliations for (possibly zero) tokens. As the initial environments were similar, these

declarations will succeed in both cases, or in both cases will detect an error in attempting to declare symbols which are already declared. If the declarations succeed, the same number of tokens are added to both environments. Consequently the environments following these declarations are covered by the input similarity augmented with ties for the pairs of tokens just created. As pairs of tokens are added, the one-to-one requirement for tokens is satisfied in the augmented similarity. Then by the induction hypothesis the body of the compute expression will extend this similarity. The same signals will be returned. Thus either errors will be detected or the lists of items returned will be covered by this similarity.

The action of determining the tokens substitutes regular references for the potential references to the tokens wherever they occur. Since the extended similarity is still one-to-one on taokens, and as tokens are determined in pairs, no ties between tokens will become ties between tokens and items. Also, since the pair of items used to determine any related pairs of tokens are similar, all items directly enclosing the tokens will still be similar after the replacements. Also because the tokens were related one-to-one, determining tokens with cells does not affect the one-to-one requirement on cells. Thus the extended similarity produced by the body of the expression covers both the environments existing after the tokens have been determined and the lists of items being returned. The undeclaring of the identifiers added to the environment by the compute expression leaves environments still covered by this similarity.

This establishes that both PO invocations and compute expressions extend input similarities to cover both the resulting environments and the lists of items produced.

Now by the induction, the evaluation of the body of this CD can be shown to extend this similarity. Undeclaring symbols of similar environments yields environments which are similar under the same similarity. The control signal returned by this CD is that returned by the body. The list of items returned by this CD is that returned by

the body, and therefore has all its items covered by the similarity which exists following the evaluation of the body.

This completes the lemma.

Now consider functionals. A functional has unbound externals. These are affiliated with references by using the PO bind. The result is evaluated using the PO eval. A list of references to items results.

Definition: The acts of binding externals of a functional and evaluating the result are together called an application of the functional. The primary evaluation of an application is the execution of the PO eval on the bound functional. The primary activation of an application is the activation created by the primary evaluation. The duration of an application of a functional is the interval whose states record the primary evaluation of that application.

The following defines the input packages, the activities specified, and the extension to output packages for the application of functionals.

Definition: The input package for an application of a functional maps the unbound externals of that functional into the items and tokens which those externals are to be bound to in the functional. The activity is all state transitions composing the primary evaluation of that application. The output package is the package which maps the appropriate subset of the symbols "result-1", ""result-2", etc. into the list of items and tokens included in the return result structure of the primary evaluation of that application.

In general, functionals do not extend similarities. In fact, even for the case where functionals have simple control structures this is not true. The problem is that a functional may enclose cells. In this case, it is possible for those cells to be updated during the application of the functional. Presented with similar inputs, the initial environments cannot in general be established as similar, for the items to which the externals of the functional are bound need not be similar. A very simple case is given in Figure 5.3.1.

```
let x be eval{g};
     ...x denotes 1
let y be eval{g};
     ...y denotes 2
```

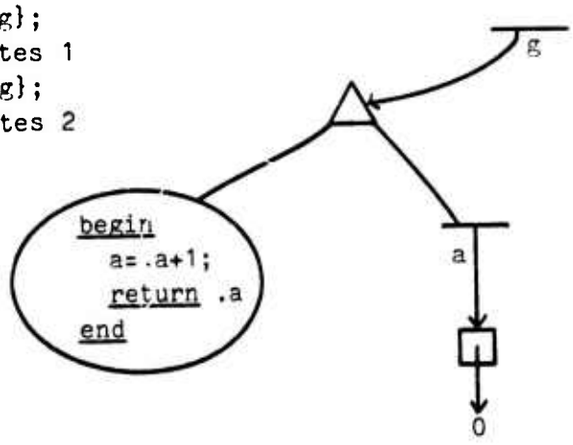


Figure 5.3.1. A simple functional g which does not extend similarities.

Even when the environments of a functional on two different occasions are similar, and the inputs are similar, the sharing patterns between the environment of the functional and its inputs may produce initial environments which are not similar. An example is given in Figure 5.3.2. Although the example is contrived, it highlights the nature of the problems produced by dissimilar sharing patterns.

```
let x be g["a":d];
       ...x denotes 0
let y be g["a":d];
       ...y denotes 1
```

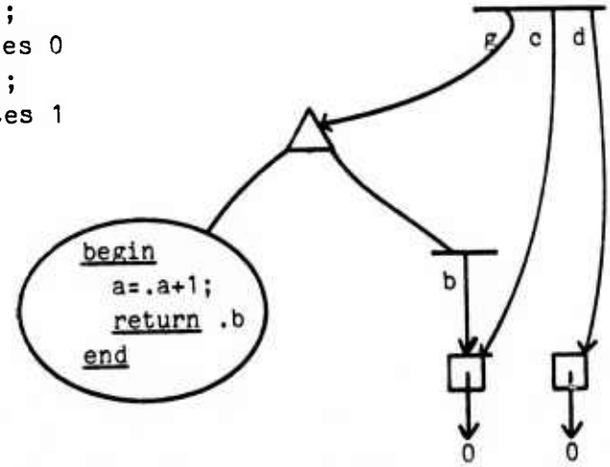


Figure 5.3.2. Applications of similar (in fact, identical) functionals on similar inputs resulting in dissimilar initial environments.

The preceding lemmas and observations suggest that extending similarities is a good measure of well-behavedness. The rest of this chapter follows this suggestion, defining the special behaviours by giving conditions under which functionas exhibiting those behaviours must extend similarities.

5.4. Proper Functionals.

Section 5.1 argued that a repeatable functional should not be held responsible for the bad behaviour of functionals which are passed in as part of the input and which are evaluated during the course of the evaluations of the repeatable functionals. This section explores the notions of bad and good behaviour in functionals which are enclosed in the input to another functional.

The goal is to choose a subclass of functionals which will be permitted in the inputs to other functionals when testing those functionals for special behaviours. For ease of reference, that subclass of functionals will be called the admissible set.

Consider the applications of a functional, f, to two similar input packages. Suppose that the initial environments of the primary activations of those applications extend the similarity of the inputs and that the activities of the functional in the two activations further extend that similarity. Now suppose a pair of functionals, g and g′, which were in those inputs and were related under the input similarity, are applied. Their behaviour will be considered good if they also extend the similarity of the environments, and bad otherwise. Notice that when the applications of g and g′ are about to begin, the similarity of environments of the activations of f covers not only the inputs to g and g′ but also g and g′ themselves. This is a strong condition. Functionals which do not extend similarities even under this strong condition are considered badly behaved.

A more subtle form of bad behaviour is producing a badly behaved functional. This is like handing someone a time bomb. Consequently, all

functionals introduced by a well-behaved functional into the environments of its appliers (through side-effect or return list) should themselves be well-behaved.

These ideas concerning well-behaved functionals can now be formalized.

Definition: If p is an input package for a functional f, and q is formed by adding to p a mapping of the distinguished tag * into f, then q is called the combined package for f on p.

The integer 0 is chosen as a tag for the functional so that the combined package may be extended to enclose the outputs of its application on the input without conflict in the use of tags.

Definition: Let S be a set of functionals. Let f and g be in S, and p and q be input packages to f and g. Let k and l be the combined packages of f on p and g on q respectively. Then S is called well-behaved if, whenever

1. k and l are similar, and
2. all functionals enclosed by p and q are in S,

then it can be shown that f and g extend the minimal similarity on p and q in such a way that all functionals enclosed in the domains of the minimal extended similarity are in S.

It is desirable to find a large well-behaved set of functionals for use as an admissible set. The larger the set, the more demanding are the definitions of the special behaviours, for then more pairs of similar inputs will have similarities which will have to be shown to be extended by specially-behaved functionals.

An obvious choice for the admissible set is the largest set of a well-behaved functionals which exists.

Definition: The largest well-behaved set of functionals in the model is called the maximal well-behaved set.

It must be established that this set is well-defined. To illustrate the problems with doing that, consider the functionals y and z of figure 5.4.1. First, observe that if a functional is in the maximal well-behaved

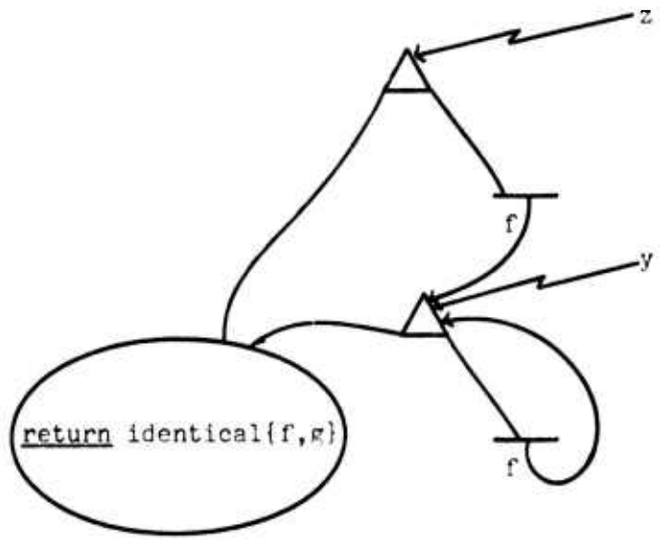


Figure 5.4.1: Picture of functional y for which "well-behaved" is not well-defined.

set, then any functional similar to it is also in the set. This follows from the symmetry of the definition of well-behaved. It is easy to see that y and z are similar but not identical. Both have "g" as the sole unbound external. Applying y to itself yields true. Applying z to itself yields false. Consequently, y and z should not be in the maximal well-behaved set. On the other hand, if they are left out then inputs enclosing them need not be considered in determining whether functionals are in the set. But they are enclosed in themselves. So they need not be considered in determining their own right to be in the maximal well-behaved set. But for all other inputs to y and z, these functionals yield false. Consequently, y and z should be in the set.

Thus, if it is assumed that y is in the set, it can be shown that y should not be in the set; and if y is assumed not to be in the set, it can be shown that it should be in the set. So the well-defined-ness of the maximal well-behaved set is questionable. This is left as an open question.

The source of the trouble with the definition of the maximal well-behaved set appears to be the PO identical. As the earlier lemma showed, only eval, identical and new-key can fail to extend similarities.

If identical and new-key were disallowed, then by induction eval could be shown to extend similar environments. So if the model were without identical and new-key, the set of all the functionals would be well-behaved.

It is undesirable, however, to leave identical and new-key out of the model. Firstly, with self-referencial sheaves it is impossible to write exhaustive search functionals without identical. Secondly, identical is the only way of establishing the commonly used "weak" equality on the functions implemented by functionals. Thirdly, identical is indispensible in creating the equality operations for programmer-defined types. And finally keys are essential for protection, and new-key is essential for keys.

Nor is the removal of identical from the model justified by its being a primary source of non-well-behavedness. When the model is later extended to include facilities for achieving concurrent execution through multiple processes, other sources are introduced (see chapter 6).

Although the PO's identical and new-key are useful in the model as a whole, there is no reason that they can't be forbidden in the functionals of the admissible set. A good candidate for the admissible set is the set of all the functionals which have no occurrences of the PO's identical and new-key. The difficulty with this set of functionals lies in the behaviour of the PO install, for install can create functionals which use identical and new-key. Thus for a functional to be in the admissible set, and to include install it must always insure that the functionals which it installs and returns are also in the admissible set. That is, they must include no occurrences of identical and new-key, and they must use install correctly. Such restricted use of install is artificial indeed; one would not expect to find many creators of functionals that use install who wish to so limit the functionals that they create. Also, it may be very difficult to tell whether a functional satisfies that limitation or not.

For both these reasons, the admissible set chosen will be functionals which have no occurrences of identical, new-key and install.

Definition: A functional is said to be limited if its control structure contains no invocations of the PO'S identical, new-key and install.

Definition: A function is said to be proper if it and all the functionals which it encloses are limited. An item is said to be proper if all the functionals which it encloses are proper.

The admissible set to be used in the rest of this chapter is the set of proper functionals. It must therefore be shown that this set is well-behaved.

Theorem 5.4.1: The set of proper functionals is well-behaved.

Proof: The initial environments of both activations enclose only proper items, since both the inputs and the bound items were proper. As f and g are proper they cannot install any functionals, no improper functionals can get into the environments that way. Binding a proper functional to a proper item produces a proper functional. Consequently, all functionals applied must be proper. Hence, recursively, the results will be proper items. Hence items returned will be proper. Also inputs will remain proper.

The initial environments are similar. By an earlier lemma, all the PO's in proper functionals except eval extend similarities. But, again recursively, eval applied to similar proper functionals extends similarities. Consequently, environments remain similar; therefore the evaluations extend the similarity.

The set of proper functionals is quite large. The only things that proper functionals cannot do is test for identity on those items where similarity intentionally obscures identity, install new functionals, and create new keys.

The inability to install may appear to be a significant restriction on the largeness of the set. However, considering their role as test functionals, it is seen to be less serious. For consider some functional which, on the basis of some input, computes and installs a control structure producing a proper functional. Then there exists another functional which takes the same inputs, carries out the same activities but

instead of installing the computed control structure, it simply supplies an already-installed (proper) functional. As the installed functional is proper, the new (substitute) test functional is also proper. This argument is easily extended to repeated invocations. Thus for any test input enclosing a functional which installs proper functionals when that input is used to test a particular functional, there exists a substitute proper functional which has the same behaviour when used in that test input with the functional under test. So if a functional meets the test (for whichever special behaviour) for all inputs containing proper functionals, it will also meet the test for inputs containing functionals which install proper functionals.

With the admissible set in hand, the definitions of the special behaviours can be given, and some structurally-defined classes of functionals which satisfy these definitions presented.

5.5. Repeatability.

The simple-minded notion of a functional's being repeatable is that it should extend the similarities of pairs of similar inputs. As has been seen, to adequately capture all the intuitions about repeatability, it is necessary to restrict inputs to being proper.

Also, for wider applicability, repeatability should somehow be applicable to pairs of similar functionals. The approach taken is to define a functional to be repeatable if certain properties hold for a restricted subset of the functionals which are similar to it. The restriction is introduced to normalize the effect of the surroundings of the functionals.

Therefore, before repeatability can be defined, the notion of similar contexts must be formalized.

Definition: Let y be an item enclosed in, but is not identical to, an item x. Then an item or environment of an activation z is said to be a handle at y in x if

1. z is not enclosed in x, and
2. z directly encloses y.

An environment of an activation is said to be a "handle at y in x" if that environment includes a reference to y.

This definition is depicted in figure 5.5.1.

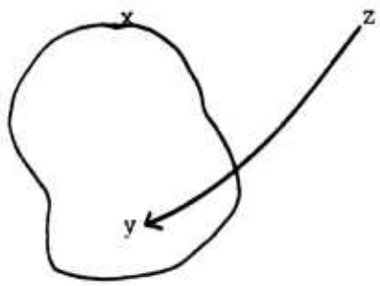


Figure 5.5.1: z is a handle on y in x

Definition: A functional f is said to be vulnerable at an item y which it encloses if there exists an item which is a handle at y in f.

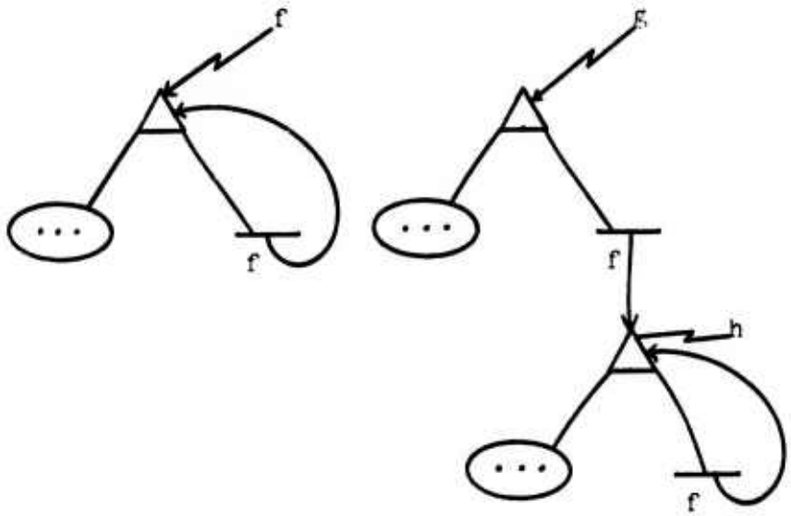


Figure 5.5.2: Similar functionals

Intuitively, a pair of similar items will be considered to be "similarly-vulnerable" if there are handles for them at items which are related under the minimal similarity covering the pair. There is one exception to this rule. In figure 5.5.2, the functionals f and g are similar. So are f and h. Suppose g is vulnerable at h. Then h could be applied. But an application of h would correspond to an application of f which in turn would correspond to an application of g. Then either all applications of h must be regarded as applications of g, or g must be required to be invulnerable in h. The former solution is unsatisfactory, for a user of g must be aware of h and all its applications. This is non-modular. The latter solution is therefore chosen.

Definition: Two functionals f and g are said to be <u>similarly-vulnerable</u> if

1. f and g are similar, and
2. if F is the minimal similarity covering f and g, then for all x enclosed in f and y enclosed in g,
   a. if y is not identical to g, and fFy, then g is not vulnerable at y, and
   b. if x is not identical to f, and xFg, then f is not vulnerable at x, and
   c. xFy implies that f is vulnerable at x if and only if g is vulnerable at y.

These definitions display a lack of concern over the nature of the "outside" items which are handles on items in the functions. This reflects the assumption that any reference into a functional will sooner or later be exploited to its limit.

Notice also that vulnerability is state-dependent. The vulnerability of a functional cannot increase unless a functional includes in its output references to enclosed items. Vulnerability can be reduced by discarding references to items. While building a functional this will be common. Once built, however, it must be assumed that references will not be discarded.

The notion of repeatability therefore involves the idea of a state in which to measure the vulnerability of a functional: Before that, the

functional was being built. After that, it is ready for use.

Now repeatability can be formalized.

Definition: A functional f in state R is said to be repeatable if for any state S and any functional g in S such that either

1. g in S is similarly-vulnerable to f in R, or
2. g is identical to f, and S follows R,

and for any pair of similar proper inputs p in R and q in S, then f and g extend the minimal similarity in R and S of p and q.

Notice that there is no requirement that f in R and f in some subsequent state S be similar. Also, in case 1, the functionals and the inputs are similar separately. Thus in neither case 1 nor case 2 is there is any guarantee that the initial environments of the activations created will be similar.

Theorem 5.5.1: If f in R is repeatable, and g in S is similarly vulnerable to f in R, then g in S is repeatable.

Proof: Suppose h in T is similarly vulnerable to g in S. Then h in T is similarly vulnerable to f in R. So f and h will extend similarities on pairs of similar proper inputs, as will f and g. Therefore so will g and h by transitivity of similarity.

Suppose state T follows state S, and g in S and g in T do not extend the similarity on some pair of similar proper inputs. Consider the sequence of packages K upon which g has been invoked between S and T. Because, g in S is similarly-vulnerable to f in R, applying f to the elements of K will produce a state T′ in which f will be similar to g in T. Consequently, whatever caused g in S and g in T to fail to extend similarities will cause f in R and f in T′ also to fail. But this cannot happen. So g in S and g in T must extend similarities on pairs of similar proper inputs.

Corollary: If f in R is repeatable, and S follows R, then f in S is repeatable.

Proof: This follows by the same argument used in the previous proof.

One goal of this research was to create a model which would support a definition of repeatability under which there would be a class of repeatable functionals which was both "large" and "structurally defined". The following lemma shows that this goal is satisfied by the definition of repeatability just given.

Definition: An item is called cell-free if it encloses no cells.

Theorem 5.5.2: A proper cell-free functional is repeatable.

Proof: If a functional encloses no cells, its structure cannot change. Consequently case 2 of the definition of repeatability is implied by case 1. Case 1 of the definition is just a restatement of theorem 5.4.1.

Outlawing cells may seem like a high price to pay for repeatability. Notice, however, that cells are only forbidden in the closures of these functionals. This still permits cells to be enclosed in inputs, created and used during applications, and returned in outputs.

The converse of this lemma is not at all true. The functional f depicted in Figure 5.5.3 shows that proper functionals enclosing cells can be repeatable. Note that the cell denoted by c which counts the number of times the functional has been applied can even be vulnerable.

An interesting open question is whether outlawing cells enclosed in functionals is any hardship at all. That is:

Open questions: Is it true that for every repeatable functional, there is a repeatable cell-free functional which has the same behaviour? Is it true in the restricted space of proper functionals?

Not only is repeatability not dependent upon excluding cells, as the functional f of figure 5.5.4 demonstrates, it is also not dependent upon only enclosing and applying repeatable functionals. The functional f uses the non-repeatable functional g in its construction. The output of g alternates between 1 and 2; each application of f causes two applications of g. f implements the constant function 3, and is therefore repeatable.

a, b invulnerable
c possibly vulnerable

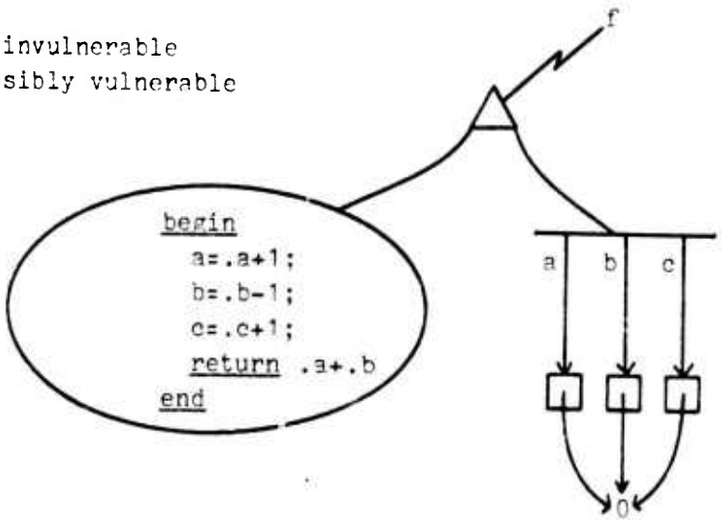


Figure 5.5.3: A repeatable functional enclosing cells

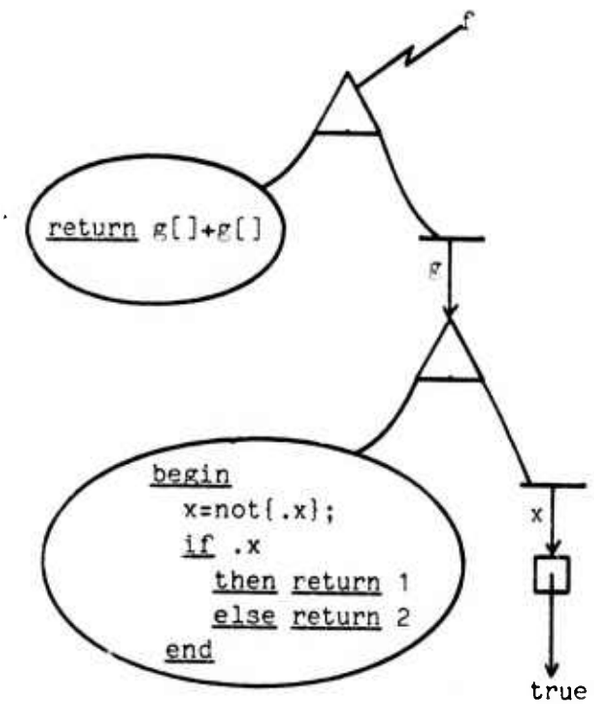


Figure 5.5.4: Repeatable f built using non-repeatable g

Again, questions arise:

Open Questions: Is it true that for every repeatable functional f there

exists a repeatable functional g which has the same input-output mapping as f and all of whose enclosed functionals are repeatable? Is it true in the space of proper functionals?

The proper cell-free functionals do not exhaust the space of repeatable functionals for there are repeatable functionals which check identity on sheaves, which install functionals, and which create new keys. However the proper cell-free functionals are a large structurally-defined class of functionals which are repeatable. Thus the goal mentioned above has been met.

5.6. Sequence-repeatability.

Intuitively, a functional is sequence-repeatable if it is repeatable over sequences of inputs. As with repeatability, sequence-repeatability is formalized in terms of a pair of similar functionals in a pair of states. In turn, these functionals are applied to pairs of similar inputs. Each pair of applications must extend the similarity of its inputs. Note that this is a weaker requirement than repeatability, for similar inputs to the same functional need not produce similar outputs. In fact, sequence-repeatable functionals are often thought of as keeping a history of their previous applications and using that together with their inputs to compute outputs. So similar inputs may be expected to produce dissimilar outputs on repeated applications.

The output of a functional can be handed back to it as input to later applications. The sharing patterns between input and output sequences must be taken into account when defining sequence-repeatability. This is done by requiring that applications should extend the similarity of the inputs, and that the extended similarity be further extendable to cover the inputs to the following pair of applications.

Between applications, changes may be made to the inputs and outputs of the preceding applications. In testing for sequence-repeatability, it is assumed that these changes are similar in both sequences. That is, the

similarity covering inputs and outputs up to the end of a corresponding pair of applications must still be a similarity when the following applications begin. Also, it is assumed that any new items added to the similarity are proper. That is, all items handed to the functionals, whether directly in the form of inputs or indirectly through side effects, must be proper.

If this cycle of extensions by the applications and the users can be continued arbitrarily, the functionals are said to have been sequence-repeatable in their original states.

As with repeatability, vulnerability must be similar in the initial states of the functionals.

The following definitions make precise the idea of a sequence of applications of a functional. Notice that if a functional invokes itself recursively, then that recursive call is considered part of the encompassing application and is therefore ignored. This saves trying to order an application with respect to an encompassing application.

Definition: An interval is said to <u>separate</u> two applications of f if

1. its first state is the last state of an application of f,
2. its last state is the first state of another application of f, and
3. it contains no applications of f.

The two applications separated by the interval are said to "precede" and "follow" one another.

Definition: A functional f in state R is said to be <u>sequence-repeatable</u> if for any state S of the model and any functional g in S such that

1. S is not included in the duration of an application of g, and
2. g in S is similarly-vulnerable to f in R,

then the following construction can be repeated arbitrarily often:

In this construction, the initial input/output (I/O) similarity is the empty relation with empty domains.

a. Choose any pair of inputs p and q such that
   1. the I/O similarity can be extended to cover p and q, and
   2. p and q are proper.

b. Let interval A be the duration of an application of f to p with duration starting at state R and let interval B be the duration of an application of g to q with duration starting at state S.

c. (Requirement) f and g must extend the I/O similarity on p and q.

d. Let C be an interval separating A from the following activation of f, and let D be an interval separating B from the following activation of g, such that the I/O similarity can be extended so that all previous inputs and outputs are still similar under it in the final states of C and D and all items added to the domains of the similarities are proper.

Functionals which are repeatable on single inputs certainly should be repeatable on sequences of inputs. The following lemma states that this is indeed the case.

Lemma: If f in R is a repeatable functional and R is a state which is not a member of one of its applications, then f in R is sequence-repeatable.

Proof: (Names of clauses refer to the definition of sequence-repeatability.) Let g in state S be any functional similarly-vulnerable to f in R. By an earlier lemma, g in S is also repeatable. Let p and q be any similar proper inputs. (If the requirements can be met for all inputs, they can be met for the set restricted as in clause a.) Let A and B be as in clause b. As f is repeatable, clause c must be satisfied.

Let C and D be any intervals separating A and B from the following applications of f and g. (If the requirements can be met for all such separating intervals, they can be met for the intervals as restricted in clause d.)

Now notice that f and g in the final states of C and D respectively may not be similarly vulnerable. But since f in R and g in S were repeatable, and the final states of C and D follow R and S respectively, f in the final state of C and f in R will extend similarities, as will g in the final state of D and g in S. But f in R and g in S were similarly vulnerable, and therefore extended

similarities. By transitivity, f and g in the final states of C and D extend similarities. Therefore the cycle can be repeated.

Similarly, it can be shown that the cycle can be repeated after the next application, and so on. Thus the construction can be repeated indefinitely. So f in R is sequence-repeatable.

The next lemma emphasizes the fact that cells are the difference between repeatability and sequence-repeatability.

Lemma: Let f be sequence-repeatable in R. If f is cell-free, then f is repeatable in R.

Proof: Consider f and any similarly-vulnerable functional g on their first cycles. Since the I/O similariity relation is empty, the selection of inputs is unrestricted. Consequently, for any inputs, sequence-repeatability guarantees that f and g extend the input similarity. This satisfies case a of repeatability.

Since f is cell-free, nothing can change its structure. Therefore at the beginning of its second application it will be similar to f in state R. However, it may not be similarly-vulnerable to the way it was then, for its output could have released handles on items enclosed in f. As nothing can change f, dissimilar vulnerability is of concern only because some item enclosed in f may be used in an input to f. If identity of an input matters so as to make f not extend similarities then it would have done so for f as a sequence-repeatable functional. But that is impossible. Hence f on its second cycle may play the role in the definition of sequence-repeatability of "the other functional" to its first cycle. Thus the choice of inputs for the second cycle need not be restricted by the I/O similarity relation resulting from the first cycle. By repeating this reasoning, it can be seen that f in any state following R will extend similarities on all pairs of similar proper inputs. This satisfies the second case of the definition of repeatability. Hence f is repeatable.

Again, one goal of the research is to find a "large", structurally-defined set of sequence-repeatable functionals. The following lemma establishes such a set.

Definition: A functional f in some state S is said to be invulnerable if f encloses no items at which it is vulnerable.

Lemma: Let f in state R be proper and invulnerable. Then f is sequence-repeatable.

Proof: (Names of clauses refer to the definition of sequence-repeatability). Let g in S be similarly-vulnerable to f in R. Therefore g is also invulnerable. Let p and q be as in clause a. Because f and g are invulnerable, p and q are disjoint from f and g. Consequently the union of the similarity for f and g with that for p and q is a (single) similarity covering the combined packages for f on p and g on q. Let A and B be as in clause b. Because f, g, p and q are proper, and enclosed in a single similarity, clause c must be satisfied (by the lemma about proper functionals). Also (by the same lemma) f and g remain proper and are left similar to one another.

Some items in f and g may have been enclosed in the output of A and B. Hence f and g may no longer be invulnerable. But these same items are now included in the I/O similarity.

Let C and D be as in clause d. To show that in the final states of C and D a single similarity can be found which extends the I/O similarity and covers f and g, the single similarity existing for the first states of C and D is split into two parts: those items in the I/O similarity and the rest (the still invulnerable items in f and g). C and D extend the first part (by the assumption of clause d). The second part is unchanged (by invulnernability). Also (by invulnerability), the two extended parts do not overlap. Putting the parts together again results in a new single similarity in the final states of C and D. It extends the similarity for the initial states of C and D, and covers f and g. Hence f and g are similar in the final states of C and D.

On the next cycle, choosing inputs as in clause a, the I/O similarity may have to be extended. Again dividing the single similarity into two parts, the invulnernable items of f and g cannot conflict with the new items being included in the inputs. Thus a new single similarity can be established enclosing f, g, the new inputs,

and all previous inputs and outputs. All these items are proper. On the second cycle the argument can be repeated, meeting the requirements, possibly reducing the invulnerability of the functionals, and re-establishing the conditions for a further cycle. So the cycle can be repeated indefinitely.

As to the "largeness" of this set, it can be observed that invulnerability places no limitations on the behaviours of proper functionals, or indeed on the behaviour of any sequence-repeatable functional. For if it did, the dependence would have to be either because, through handles, the states of functionals could be manipulated between applications, or because handles are an essential part of each input. The latter reason can be dispensed with because for any input there is a similar input which shares only on elementary items and keys, so the identity of the item which is vulnerable cannot be important. However, as handles are unimportant for preparing inputs, applications of a functional can immediately follow one another. Thus in general, no time is available between applications. Therefore the former reason can be dispensed with. So vulnerability is unimportant to the behaviour of sequence-repeatable functionals except for limiting the inroads of hostile contexts.

Thus the full power of the proper functionals is available in the form of sequence-repeatable functionals. This is a large structurally-defined class of functionals.

Pursuing a question raised in the previous section, the functional f of Figure 5.6.1 demonstrates that repeatable (and therefore sequence-repeatable) functionals may be constructed from functionals which are not sequence-repeatable. This could be done by making judicious use of powerful handles. Alternatively, use could be made of only a subset of all the inputs which a functional could reasonable accept. Over this subset, the functional might be specially-behaved.

As in the previous section, it is clear that while it is possible to implement the constant function 3 in this way, there is no need to do so. Again, this raises questions.

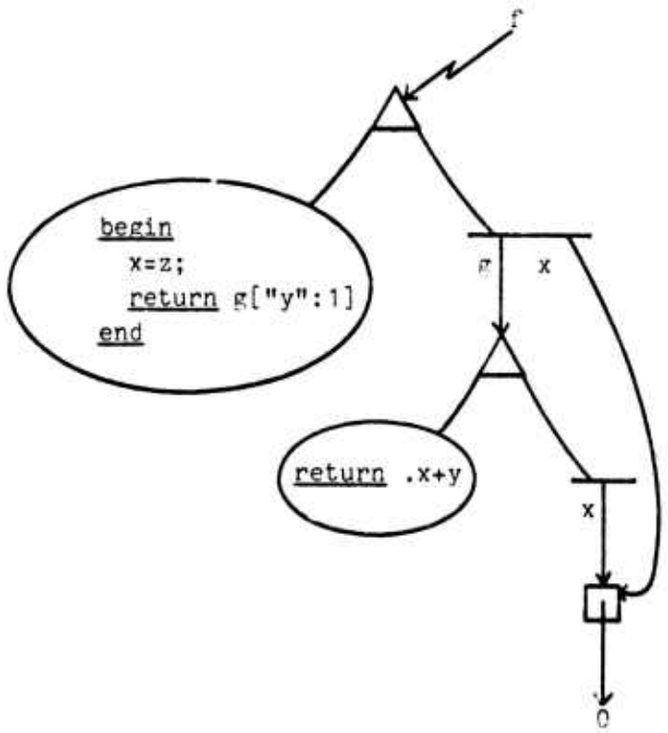


Figure 5.6.1. Repeatable f using non-sequence-repeatable g.

Open Questions: Is it true that for every repeatable functional f there exists a functional g which has the same input-output mapping as f and all of whose enclosed functionals are sequence-repeatable? Is it true for all sequence-repeatable functionals? Is either of these true in the restricted space of proper functionals?

Three lemmas concerning the construction of functionals are now given.

Lemma: If f in R is repeatable and x is an item which is proper and cell-free, and if g is the result of binding x to an unbound external s of f, then g is repeatable.

Proof: Let h in S be similarly-vulnerable to f in R. Suppose p and q are similar inputs to g and h. The action of g on p is the same as that of f on a package created from p by adding a mapping from s to x. As g is bound at s, and h is similar to g, then h is bound at s. Therefore there is a functional h′ and a proper cell-free item y such that binding h′ to y at s yields h. The action of h on q is the same as that

of h' on the package created by extending g to map s to y. But f and h' are similarly-vulnerable, and the extended packages are similar and proper. But f was repeatable; so since f and h' would have extended similarities, so will g and h.

The argument for the second case of the definition is similar.

Lemma: Binding an invulnerable proper cell to a repeatable functional yields a sequence-repeatable functional.

Proof: (This proof is similar to the proof for the preceding lemma.) Because the cell can change contents during applications of the functional, the extended input packages are possibly different on each pair of applications. Because the cell was in the input of the repeatable functional the output similarity covered it. Because the cell is invulnerable, the output similarity guarantees the input similarity on the next cycle. So the repeatability of the initial functionals can be used to show extension of similarities in each case.

Lemma: Binding any invulnerable proper item to a sequence-repeatable functional yields a sequence-repeatable functional.

Proof: This proof is similar to the proof for the preceding lemma.

The next section discusses anotner restructuring question concerning sequence-repeatable functions.

5.7. Conditions.

The intuition behind sequence-repeatability is that a functional exhibiting this behaviour has an "initial internal condition", and that applications compute outputs and change the internal conditions. In these terms, sequence-repeatability would require extending I/O similarities of sequences, and leaving "like" conditions after each pair of applications. (The internal condition is sometimes called the "internal state" of the functional. The term "condition" will be used in the following for to prevent confusion with states of the model.)

It is tempting to search for a reflection of this intuition in the model by looking for items which represent conditions of sequence-repeatable functionals. Then like conditions might be represented by similar items. Suppose functional f in R is sequence-repeatable. Find a repeatable functional g with one more input and one more output than f. Find some item c in R to represent the condition of f in R. Now g applied to c and any inputs should return whatever f would have returned if it had been applied to those inputs. Also g should return (as the additional output) an item representing the condition of f following the application. This pattern could be repeated, recycling the condition of f with each application of g. The functional g will be called a "proxy" for f and c is called the "initial condition" of f for g.

Definition: If a functional f in R is sequence-repeatable, then a functional g in R is said to be a <u>proxy</u> for f, and an item c in R is said to be an <u>initial condition</u> of f for g if

1. g has one more element in each input and each output than f. On the first application of g, c is the additional element; on subsequent applications of g, the addition element of the output of the preceding application is the additional element of the input.
2. considering only the remaining elements of the input and output, g has the same input/output mapping as f.
3. considering all the elements of the input and output, g is repeatable.

The problem with this definition is that it does not capture the intuition completely. Consider the following lemma.

Lemma: Every sequence-repeatable functional has a proxy.

Proof: Suppose f in R is sequence-repeatable. Let f be its own condition. Then consider the functional g which takes f's inputs and f as condition, applies f to its inputs, and returns f's outputs and (the presumably now changed) f. This functional clearly has the same input/output mapping as f. It is proper and encloses no cells, so it is repeatable. Thus g is a proxy for f.

The definition fails to capture the intuition of proxy simply because it puts no restrictions on the nature of the conditions which are permissible. This leaves the following question.

Question for Thought: To appropriately capture the intuition of proxy, what restrictions should be placed upon the conditions which proxies may use?

The following definition indicates one possibility.

Definition: A proxy is said to be <u>cell-free</u> if its initial condition and all the conditions which it produces are cell-free.

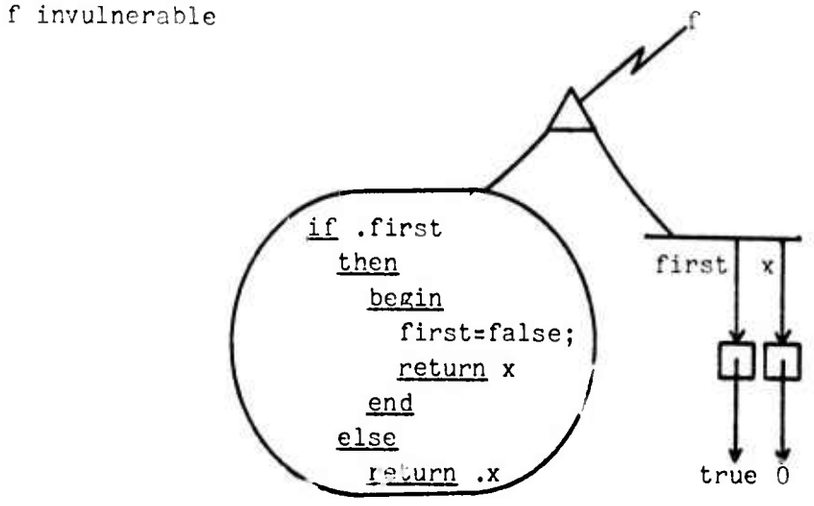


Figure 5.7.1. Sequence-repeatable f with no cell-free proxy.

The functional f of Figure 5.7.1 demonstrates that there is a functional which has no cell-free proxy. It is sequence-repeatable. It behaves like a cell which can be read. Notice that the first application makes the cell vulnerable. However, because it is an output of the functional, its updating is the responsibility of the users.

The first application of the proxy of f must return a cell (plus a condition). Any update by a user of that cell must affect the condition. This can only be done with a cell in the condition. Hence f has no cell-free proxy.

The trouble here is that the definition of sequence-repeatable allows functionals to materially increase their vulnerability. Of course, there is a large class of sequence-repeatable functionals which will not do this. So again there are questions.

Question for Thought: How does one characterize the subset of the sequence-repeatable functions which "do not materially increase their vulnerability"? What is the relationship between the set of sequence-repeatable funetionals which possess cell-free proxies and those which materially increase their vulnerability?

The notions and questions about proxies are interesting mathematically. However the use of proxies conflicts with modularity because functionals requiring proxies are incomplete.

5.8. Instances of Programmer-defined Types.

In Section 3.8, the technique of procedural encapsulation was advanced as a means of supporting programmer-defined types. This section extends the results of this chapter to include instances of programmer-defined types represented in this way.

An instance is represented by a bundle of functionals. The idea is to associate such a bundle of functionals with a single functional whose behaviour is then examined.

Definition: If b is a bundle of functionals, then the functional f is said to be the "eonglomerate" of b if f takes as inputs an extraetor for b and inputs for the funetional extracted by that extractor, does the extraction, applies the funetional, and returns the results.

Definition: A bundle of functionals b is said to have a eertain property if its eonglomerate has that property. The properties of interest are proper, repeatable, sequence-repeatable, cell-free.

Notice that the act of forming a conglomerate does not need to use install, identical or new-key so the notion of proper is as one would

intuitively expect. The notions of similar, similarly-vulnerable and invulnerable are already defined for bundles of functionals.

All the results of this chapter extend to bundles of functionals. As a consequence, the real importance of sequence-repeatable becomes apparent: instances of certain mutable programmer-defined types such as switches and stacks, while not repeatable, are certainly sequence-repeatable. Knowing this, new (and similar) instances can be created as items "local" to each application of some other functional. If these applications have similar environments on similar inputs, then the I/O similarities for the instances will be subsimilarities of the similarities on those environments. Therefore use of these mutable instances will be guaranteed to extend similarities within the applications. This is helpful in constructing specially-behaved functionals.

5.9. On the Balancing of Definitions.

This chapter has established that the model satisfies the requirement that it adequately support definitions of the special behaviours. The tests for adequacy have been two: that intuitions concerning the special behaviours be captured by the definitions, and that large, structurally-defined classes of functionals should exist which satisfy those definitions.

Thus the requirement that the model support definitions of the special behaviours is expressed using two loosely-defined notions: the intuitions about special behaviours, and the largeness of classes of functionals. Therefore there are undoubtedly many ways to satisfy the requirement. Classes of functionals of various sizes can be argued to be "large", and intuitions vary. Furthermore, the requirement is met by balancing three different definitions against one another: the definition of the model, the definition of special behaviour, and the definition of the classes of functionals. Undoubtedly, these definitions can be made to balance in many different ways.

An example of this balancing may be of interest here. An earlier version of the model did not have the primitive type key. Items of all kinds were permitted as selectors of sheaves and extractors of bundles.

```
IDENTICAL[item1,item2]:
  let sheaf be <item1: 0> in
    return is-selector{sheaf,item2}
```

Figure 5.9.1. A functional implementing identical in an older version of the model.

In that version, finding a large class of functionals was difficult, for identical can be derived from the PO select on sheaves, as shown in Figure 5.9.1. Therefore to avoid identical, the use of is-selector and is-extractor must be forbidden; this severly reduced the class of functionals which could be shown to meet the definitions of the special behaviours.

An obvious alternative was to allow only elementary items as selectors and extractors. However, hidden operations then became impossible.

These considerations led to the redesign of the model to include keys. By introducing a new primitive type, the desirable cases of identity testing on non-elementary items was made structurally distinguishable from the non-desirable cases.

The definitions given in this chapter have demonstrated that the model thus-defined achieves the required balance.

Chapter 6. Processes.

The goal of this research has been to design an abstract machine which will support modular programming. Chapter 1 made it clear that this involved supporting a community of programmers. Of special interest is the support of on-line systems where programmers interact with their programs, particularly those where the interactions with a number of programmers can take place concurrently.

To support concurrent activity, the model is extended to permit the creation and co-ordination of processes. This chapter discusses this extension, first informally, then in examples, and then formally in VDL. How the results of the previous chapter are affected by the extension is then discussed. Finally, a design for implementing a multi-programmer interactive programming system is given.

6.1. Informal Description.

A state of the extended model comprises a graph of items and a set of processes. A state transition is defined by chosing one process from the set and carrying out a transition as if that process were the only one in the model. The choices of states are assumed to be "fair", in the sense that no process gets neglected forever.

A new primitive operation (PO)

fork{fnl}

is added to the model. This PO creates a new process whose task is the functional which is its operand. Fork (like update) yields an empty list of items as its result. Once created, a process evaluates its task just as did the single process in the unextended model.

The final states of the model are those in which all processes have completed their tasks and the distinguished state error which indicates that some process has detected an error during the evaluation of its task.

Notice that processes are not items in the model. Thus it is not possible to reference a process. In keeping with this, there is no concept of an "inactive" process for, even if there were appropriate PO's, no "active" process could ever refer to an inactive one to reactivate it.

Thus processes are totally independent of one another, except insofar as the functionals which are their tasks enclose shared items. Shared cells can be used for communication. For example, one process "writes" into the cell (using update), and another "reads" it (using contents). Such communication requires co-ordination however, for the reading process must wait until the writing process has in fact written.

To provide for co-ordination, semaphores (see Dijkstra, 65b]) are included in the extended model. They are non-elementary items. A semaphore may be thought of as having a count and a queue of processes. One is depicted as in Figure 6.1.1.

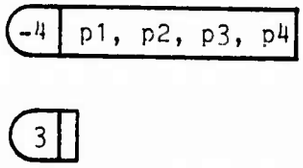


Figure 6.1.1. Depiction of two semaphores.

The primitive type semaphore is supported by four PO's:

new-semaphore{int}
p{sem}
v{sem}
is-semaphore{any} .

The PO new-semaphore yields a new semaphore whose initial count is the operand of the PO. This count must be non-negative; otherwise, an error results. The PO p decrements the count of its operand by one; if the

resulting count is negative, the process places itself at the end of the queue, and is not permitted to continue evaluation. When the process is removed from the queue (through an evaluation of the PO v by some other process), the execution of p is finished. The PO v increments the count of its operand. If the original count was negative, it dequeues the process at the head of the queue in the semaphore. The PO's p and v yield no items as results; they are executed purely for their effect in co-ordinating the executing process with others in the model. The PO is-semaphore yields a logical item indicating whether or not its operand is a semaphore.

In the unextended model, cells were the only mutable items. Semaphores are mutable in a different dimension: the PO's for semaphores always yield identical items (none) when they terminate; but their execution on identical items may terminate on one occasion and not on the next. So in the extended model, cells are still the only items which are mutable with respect to the results of PO's, while semaphores are the only items mutable with respect to the termination of PO's.

Control structure representations are extended to provide for the additional PO's. Like the other PO's, fork, new-semaphore, p, v and is-semaphore are represented by corresponding symbols.

6.2. Examples.

The demonstration language DL is extended so that programs using processes and semaphores can be written: fork, new-semaphore, p, v and is-semaphore are used to indicate the additional PO's. Fork, p and v which yield zero results are also supported by abbreviated invocation forms, like that for update. For example,

```
fork{f}
```

means

```
execute fork{f} in nop .
```

Also, like the call form for eval, the combination of binding and forking is supported with an abbreviation:

f⦃ID1:ID2, ..., IDm:IDn⦄

is a shorthand for

fork{f with ID1:ID2, ..., IDm:IDn bound} .

The invocation of new-semaphore on the integer 1 yields a semaphore which can be used for mutual exclusion. A mutual exclusion semaphore is useful for locking a shared item. If each access to the item is preceded by an invocation of p on the semaphore, and succeeded by an invocation of v on the semaphore, then only one process at a time will access the item.

One way to insure that all accesses to an item are surrounded by such co-ordinating invocations is to encapsulate the item and the semaphore in functionals which manipulate the item. Consider, for example, the programmer-defined type "queue", with operations ENQUEUE, and DEQUEUE. Instances of this type implement a "first in, first out" communication discipline. Figure 6.2.1 depicts a bundle representing a queue. Figure 6.2.2 gives texts for the operations. Notice that DEQUEUE yields two results: the first is a logical item indicating whether or not the queue had an item which could be dequeued; the second is the item if there was one, or false (meaning "irrelevant") otherwise.

Processes can only transmit items to each other through the use of cells. If a functional creates a process to help in carrying out the work of the functional, the task of that process must enclose a cell through which to communicate its result to its creator. Also co-ordination on the use of that cell must be provided.

Figure 6.2.3 depicts a bundle representing a newly-created instance of a programmer-defined type called "result-cell". This type has the two operations WRITE and READ. Attempts to read and write must alternate; the first operation permitted is a WRITE; semaphores are used to enforce this discipline. Figure 6.2.4 gives texts for WRITE and READ.

As an example of the use of processes to achieve concurrent computation, Figure 6.2.5 gives the text of a program REVERSE which

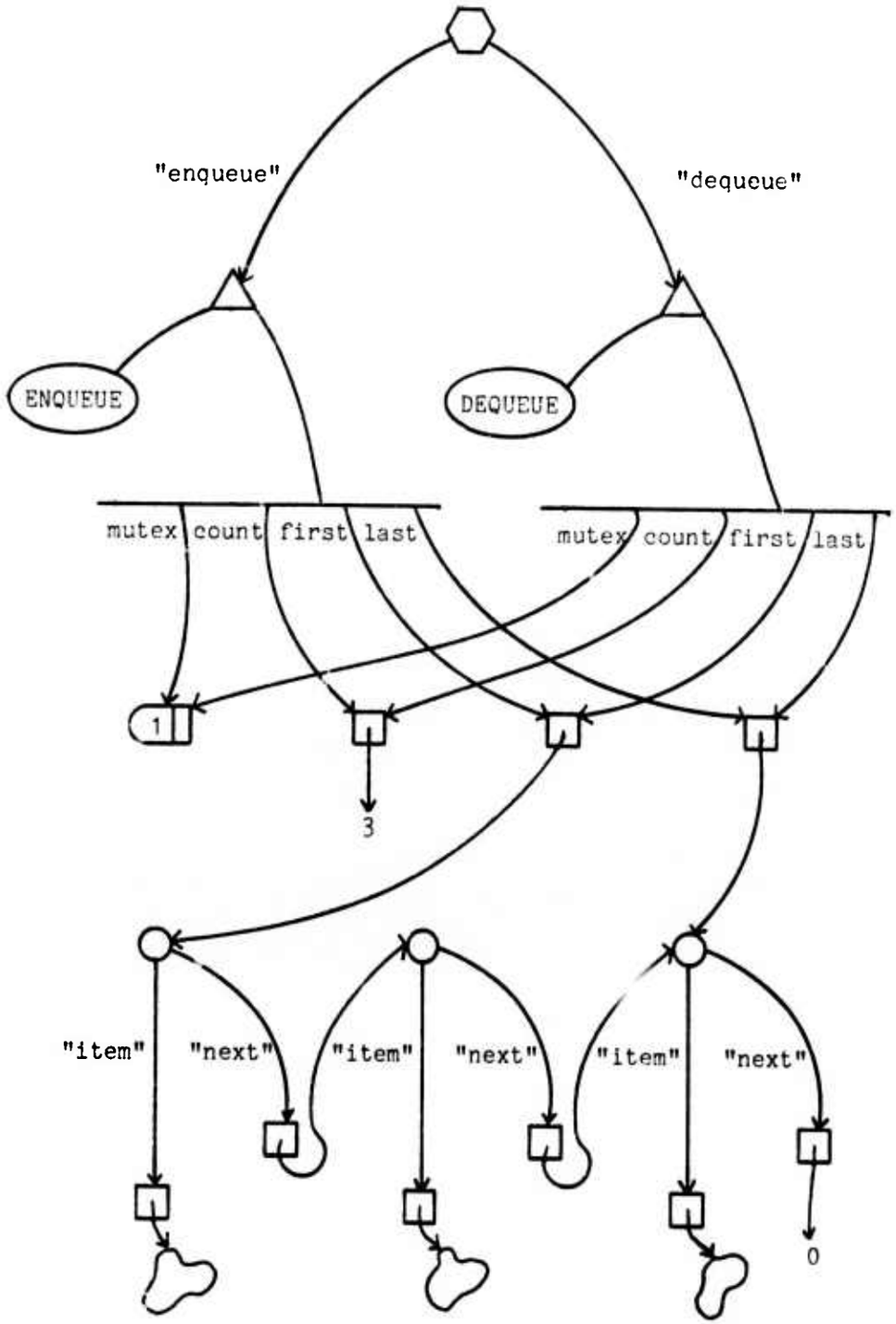


Figure 6.2.1. A bundle representing a queue having 3 elements.

```
ENQUEUE[item,mutex,count,first,last]=
  begin
    p{mutex};
    let element be <"item":new-cell{item},
                    "next":new-cell{0}> in
      if count eq 0
      then begin
             first=element;
             last=element
           end
      else begin
             (.last)$"next"=element;
             last=element
           end;
    count=.count+1;
    v{mutex}
  end


DEQUEUE[mutex,count,first,last]=
  begin
    p{mutex};
    if .count eq 0 then return false,false else nop;
    let item be .((.first)$"item") in
      begin
        if .count eq 1
          then begin
                 first=0;
                 last=0
               end
          else first=.((.first)$"next");
        count=.count - 1
        v{mutex};
        return true,item
      end
  end
```

Figure 6.2.2. Texts of operations on queues.

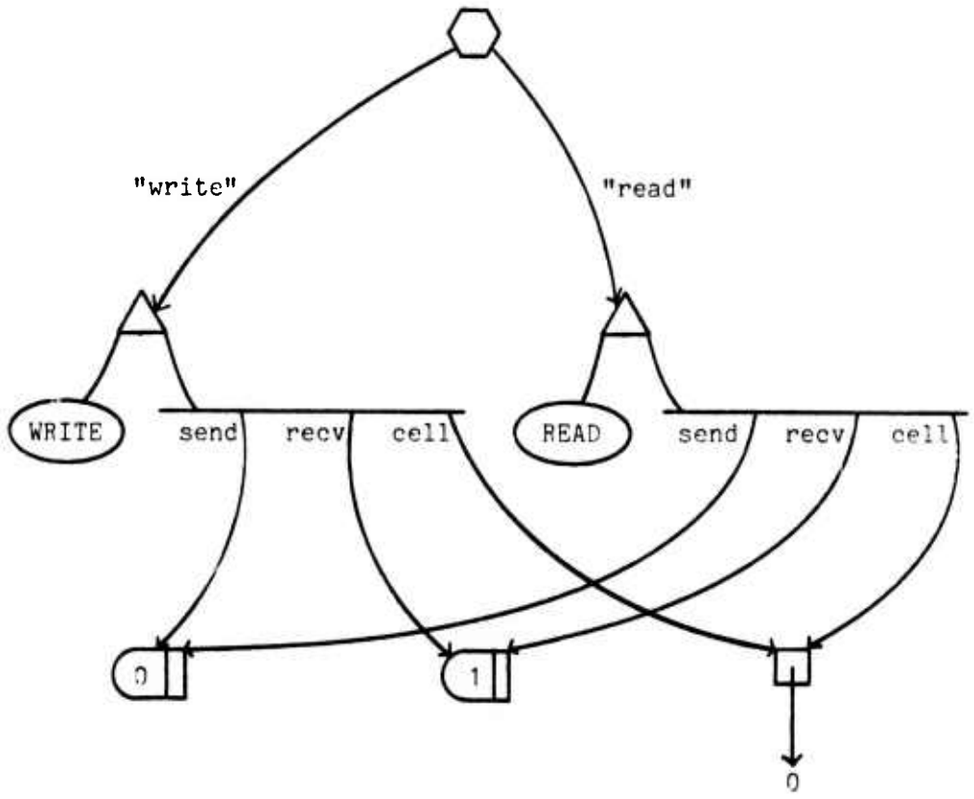


Figure 6.2.3. An instance of a newly-created result-cell.

```
WRITE[send,recv,cell,item]=
  begin
    p{recv};
    cell=item;
    v{send}
  end
```

```
READ[send,recv,cell]=
  begin
    p{send};
    let item be .cell in
      begin
        v{recv};
        return item
      end
  end
```

Figure 6.2.4. Text for operations on result-cells.

```
REVERSE[list,rev,make-result-cell]=
  let rc be make-result-cell[] in
    let fun be rev with "list":list,"rc":rc bound in
      begin
        fork{fun};
        return (rc!"read")[]
      end

REV[list,rc,rev,make-result-cell,atom,car,cdr,cons]=
  if atom["list":list]
    then (rc!"write")["item":list]
    else
      let rc-car be make-result-cell[] in
      let rc-cdr be make-result-cell[] in
        begin
          rev{"list":car["list":list],"rc":rc-car};
          rev{"list":cdr["list":list],"rc":rc-cdr};
          let rev-car be (rc-car!"read")[] in
          let rev-car be (rc-cdr!"read")[] in
            let ans be cons["left":rev-cdr,"right":rev-car] in
              (rc!"write")["item":ans]
        end
```

Figure 6.2.5. Text of REVERSE which uses concurrent processes to reverse a LISP list.

operates on LISP lists. The reverse of an atom is itself. The reverse of a node is another node whose car is the reverse of the original node's cdr, and whose cdr is the reverse of the original node's car. REVERSE uses an auxiliary program REV which WRITES its result into the result-cell denoted by rc, rather than returning it. The identifiers atom, car, cdr and cons denote the defining operations for LISP lists; make-result-cell denotes the program which creates instances of type result-cell as just discussed; rev denotes REV (that is, REV is self-referential at rev). Notice the use of both DL abbbreviations for invocations of the PO fork.

6.3. A Demonstration System.

This section briefly outlines how a programming system can be constructed in the extended model to support a community of programmers who can create and share modules as discussed in Chapter 1. This programming system would take the form of a functional, undoubtedly having a rather extensive closure of supporting functionals and data. It will be called the Demonstration System (DS) in what follows. This functional could be used as the task of a process. For simplicity, it will be assumed that this process is alone in the initial state of the model.

If the model with DS as the process's task is to be interactive, some means of representing keyboard/typewriter terminals is necessary. The best way to do this would be to add a primitive type to the model. Items of this type would have the properties of a terminal. The definition of such a type is beyond the scope of this work. For the purposes of specifying DS, therefore, it is assumed that in the initial state of the model there exists a distinguished set of six items for each terminal to be represented. Three are for input, and three are for output. Each of these groups of three items comprises a cell and two semaphores; they are manipulated using the discipline of result-cells (see Section 6.2). On the input set, the "terminal" will be assumed to attempt WRITE operations when keys are depressed. On the output set the "terminal" will be assumed to be continuously attempting READ's and typing the results. The items passed are assumed to be single-character symbols. Thus each terminal can be represented as a bundle with two components which are functionals for reading and writing. The initial state is assumed to include a sheaf of these.

DS also uses directories. A directory is a mutable list of associations between symbols and items. It has operations to add and delete associations, and to search for the item associated with some symbol. Its representation could be a cell whose contents is a sheaf with the symbols as selectors and the items as components.

DS has one directory of "programmers"; it is called the "system" directory. The symbols in the associations of the directory are the programmers' names; the items are sheaves with two components: a password (some symbol) and a programmer's directory. A programmer's directory lists all the items in the system which the programmer can reference. Some of these items may be very complex: directories of directories, or functionals with extensive closures. Some may be very simple yet very important: keys which provide access to otherwise unobtainable services.

Initially, DS uses fork to create a process for each terminal, using as operand a functional which encloses both the bundle representing that terminal, and the system directory. This process is called the "logger" for that terminal. The evaluation of DS is then finished; the process whose task that evaluation is is therefore complete.

The logger for the terminal writes a herald message on the terminal and waits for a person to type something; the wait is achieved by attempting to READ from the terminal. When characters can be read, the logger echoes them and assumes they are a request to "log in". The logger parses this request and locates the programmer's name and a password. The name is used as a selector in the system directory. The password component of the resulting sheaf is compared with the password created from the log in request. If they match, the logger calls a "command language interpreter" functional with the programmer's directory and the terminal as arguments.

The "command language interpreter" (CLI) repeatedly writes a prompt character on the terminal, reads and echoes characters from the terminal, parses the resulting string as a command, and takes the action requested by that command. The iteration continues until a command is invoked the semantics of which is a return from CLI. The logger then writes a closing message and again gives the herald message and waits for another human to login. Each iteration of the logger is called a "session".

The specification of the system and semantics of the commands understood by CLI is called a "command language". For simplicity, suppose one is designed to look as much like DL as possible. Thus CLI would start by writing "begin". Then each command would be a DL representation of a

control structure. It could take any number of lines and be ended with ";". The "end;" command would be the end of the sequence; "end" is echoed and CLI returns. "Return;" has the same effect as "end;". The programmer's directory would be used as an environment for resolving denotations of all identifiers in the commands. Certain identifiers which are not reserved in DL would be reserved in CL for special purposes: terminal could denote this terminal, directory could denote the programmer's directory.

Declarations could be handled in two ways: Following DL, the command "let....be....in;" could cause echoing of all but the semi-colon, cause augmenting of the programmer's directory as indicated, and cause a new "begin" to be written introducing the body of the declare control directive; the body would then be a sequence created interactively by more typed commands. CLI could even force reasonable indentation for the new sequence. "End" would finish the sequence and undeclare the identifiers approrpriately. "Return" would cause all closing "ends" to be written, appropriate identifiers to be undeclared, and CLI to be finished. To effect permanent changes in the programmer's directory, another command would have to be used.

The other scheme for doing declarations does not follows DL; it is more conventional. "Let....be....;" would augment the directory. "Forget....;" would delete associations from the directory. Declarations are not nested in this scheme; there is only one sequence being interactively created; "end" and "return" have the same effect.

In either scheme, CLI could, and probably should, have special commands for listing the current environment so the programmer can ascertain where he is when he loses his place.

Invocation of eval is the vehicle for "running" existing functionals. A special case of this is applying the functional CLI with the terminal and some other directory as arguments. This permits work to be carried out in the context of another directory.

There is nothing to prevent two sessions on two different terminals from being simultaneously associated with the same programmer. The

operations on directories must therefore be written using semaphores to co-ordinate concurrent use of directories.

A simple way of passing items among programmers is to have a shared directory which is associated with the symbol "root" in each programmer's directory. By establishing protocols for adding to and deleting from this shared directory, a programmer can arrange to share items. Suppose, for example, that two programmers wish to set up an item which only they can reference. One programmer creates it. If he puts it in the root directory, any programmer can get it. So he creates a new key, and a functional which yields that key on its first invocation, and the logical item false thereafter. (This functional must use a mutual exclusion semaphore.) This functional is placed in the system directory. The intended sharer invokes it. If the item returned is the item false, then an intruder stole the key. In this case the whole procedure is repeated with another key and another functional. Eventually, a key will get to the intended sharer. Now the item to be shared can be made the sole component of a bundle; the shared key is used as its extractor. This bundle is placed in the system directory. Only the intended sharer will be able to extract the component. If the item shared is a directory, then this somewhat involved procedure need only be followed once, for further sharing can be done through this directory to which only the two sharers have references.

The preceding discussion assumes that humans do not make mistakes. While CLI can complain about syntax errors in commands, it cannot prevent the invocation of a functional which loops forever. Some means of interrupting the evaluation of a command must be provided. This requires adding interrupt and error-handling mechanisms to the model. Such additions are beyond the scope of this work, and will not be discussed further here.

6.4. Additions to the Formal Definition.

This section describes the extensions which must be made to the VDL definition of Chapter 4 in order to formally define the model with multiple processes and semaphores. These extensions appear in brackets in appendices B and C.

To the s-items component of each state a new type-list is added for semaphores (see Section B.2). A semaphore has three components: s-lock which is a logical, and s-count, and s-next-process both of which are integers. The first of these is used to implement a mutual exclusion lock so that at most one process at any time is manipulating the representation of the semaphore. The second is the semaphore's count. The third is the index in the list of processes of the process which is at the head of the queue for the semaphore; if the queue of processes is empty (as for example just after the semaphore's creation), comp(s-next-process)[1] is 0.

An additional element rtpsem is added to the primitive set of reference types. A reference to a semaphore has this element as its s-type component and an index into the list of semaphores as its s-index component.

The s-processes component of each state has an additional component: s-done (see Section B.3). It is an integer, which indicates the number of processes which have finished their tasks.

Comp(s-processes) of the state, which in the definition for the unextended model had a single component representing the single process in the model, becomes a list of processes. A process is referred to within the definition by its index in this list.

Each process has an additional component in the extended definition; comp(s-next-process) is an integer. If a process is not enqueued on some semaphore, this component is -1. If the process is enqueued on a semaphore, this component is the index of the process which follows it in

---

1. As in Chapter 4, this notation is used to mean "the s-next-process component".

the queue; 0 indicates that this process is at the tail of the queue. Thus the queue of a semaphore is represented by the s-next-process components of the semaphore and of the processes in the queue, each giving the index of the following process, or 0 to indicate the end of the queue. On initialization of a process, s-next-process is set to -1.

Only one new primitive predicate is required: is-rtpsem (see Section B.11). From this, the extended primitive sets of types can be derived. From exisiting primitive predicates, the predicates for the additional references to symbols used in control structure representations can be derived.

The concurrent evaluation of processes is represented in VDL by arranging that the VDL object which is the s-c component of the state has a branch for each process (see Section C.1). VDL's state transition rule then effects the choosing between processes which was discussed earlier in this chapter. The fairness assumption of this transition function (that a leaf instruction will be interpreted after at most a finite delay) is intended to impose on this definition of the extended model the fairness requirement discussed earlier. As Chapter 4 pointed out, the instruction schemata for a single process involve no essential concurrent evaluation. Thus all the control trees resulting from run-process(i) have only a single branch.

In the extended definition, a new top-level schema is added. Run-processes invokes run-process for every process created in the model. These invocations are arranged as independent branches of a control tree, so that all the trees resulting from the various invocations of run-process evaluate concurrently.

Run-processes(i) arranges for the concurrent evauation of the i-th and all following processes. Run-processes(i) first checks to see whether all processes have finished. As part of completing its task in the extended model, a process evaluates the shema finish which adds 1 to the s-done^s-processes component of the state. This component is 0 in the initial state. Thus if the s-done^s-processes component of the state is an integer whose value is the length of the list of processes, all processes

have finsihed. In this case run-processes is finished. Otherwise, if the i-th process exists, run-process is invoked on it, and concurrently an invocation of run-processes(i+1) is made to handle the concurrent evaluation of the rest of the processes. If the i-th processes does not exist, run-processes invokes itself again, thus effectively waiting for either completion of all existing i-1 processes or the creation of the i-th process. The initial state has as s-c component the instruction run-processes(1).

Initializing a process is extended to set the s-next-process component to -1 to indicate that the process is not enqueued on a semaphore.

All five of the PO's for semaphores are monadic, so five additional clauses are added to eval-monadic (see Section C.3).

Evaluation routines for the new PO's are added (see Section C.6). Evaluation of fork requires checking that the operand is a functional, and lengthening the list of processes by 1, making comp(s-task) of the new process a reference to the functional.

A new semaphore is initialized unlocked (s-lock is true), with a count equal to its operand, and an empty queue (s-next-process is 0). The PO is-semaphore is similar to the type-testing PO's for the other primitive types.

The PO p must lock a semaphore before any manipulation of it is done. (The routines lock and unlock are in Section C.13). Once locked, the semaphore has its count decremented. Release-and-wait just unlocks the semaphore if the resulting decremented count is not negative; in this case no wait is necessary. When a wait is required, the process enqueues itself at the end of the semaphore's queue, unlocks the semaphore and waits. Unlocking must precede waiting, for otherwise no other process could ever alter the semaphore with the PO v so as to terminate this process's wait. To place itself at the end of the semaphore's queue, a process traverses the queue looking for 0 in the s-next-process component of the semaphore and each enqueued process; then it changes the 0 component to its own index, and changes its own s-next-process component to 0. Waiting is

achieved by repeatedly polling this component; when it is found to be negative, the wait is complete.

The PO v also locks the semaphore. It increments the count. If the result is positive, then the semaphore is unlocked, and 0 items are returned. If the incremented count is negative or zero, the process which is at the head of the semaphore's queue is dequeued before unlocking the semaphore. This involves changing the s-next-process component of the semaphore to the next process in the queue (or to 0) and setting the s-next-process component of the dequeued process to -1.

Extending the type-list to make a new semaphore is done in the same way as are the extensions for the other non-elementary types (see Section C.10).

Because a semaphore is mutable, the "get" routine for semaphores returns the index of the semaphore rather than the VDL object which represents the semaphore (see Section C.11). Thus, as with cells, changes in the semaphore can be made.

Attempt-to-lock takes a VDL compound selector as argument (see Section C.13). In a single VDL state transition, it accesses the component of the state selected by that selector and changes that component to the logical value false. Using this swapping routine, lock waits until the component it returns on some attempt is true. Thus lock waits until a selected component is true and then sets it to false and returns. Lock is an implementation of the basic test-and-set primitive commonly used for achieving co-ordination of processes. The implementation relies on VDL's ability to make a state transition and pass a component of the old state. Unlock simply changes the selected component to true.

As in Chapter 4, it is necessary to single out a subset of the VDL states which will define the states of the Binding Model. Notice that now processes are evaluating concurrently.

In Chapter 4, distinguished states were chosen by requiring that the configuration of comp(s-c) of the state have as its sole "leaf" node an instruction produced from some member of a distinguished set of instruction

schemata. In the extended definition, there is a branch of comp(s-c) of the state for each process in the model. When, in some state, the branch for a process is in one of the distinguished configurations of Chapter 4, that process will be said to be "stable" in that state.

Also, between each pair of adjacent states, evaluation takes place on exactly one process. This process will be called the "producer" of the second state of the pair.

The set of distinguished states comprises those with the following property: the producer of the state is stable in that state. This set of states is made up of just those in which a process has become stable.

6.5. Special Behaviours.

This section discusses some of the effects which the extension has on the definitions and results presented in Chapter 5 concerning the special behaviours. The presentation is informal.

The possibility of concurrent evaluation creates no problems if cells and semaphores are not used. All other items are immutable, so the evaluation of a functional in one process cannot be effected by the existence of other processes.

However, it has been noted that without cells no communication between processes can take place. Also, communication between programmers cannot happen in the Binding Model without cells, and communication between programmers is necessary for sharing between programmers. So cells must be permitted. Semaphores must also be included to co-ordinate processes when they use shared cells.

The special behaviours are defined in the extended model as they were in the single process model using the idea of extending input similarities. However, some additional assumptions and extensions are necessary.

First, the definition of similarity must be extended to cover semaphores. Two semaphores match when they have the same count. This

means that the same number of p's can be done to each before the process is enqueued. Because semaphores are mutable, an additional requirement on similarity is added: as in the case of cells, the restriction of a similarity to the semaphores in its domains must be one-to-one. This insures that activity specifications do the same number of p's (and v's) on each pair of semaphores which are related under the input similarity.

Another extension needed is to the definition of applications of functionals: fork must be added to eval as a way of bringing about the primary evaluation of an application. Thus an application of a functional can take the form of either binding and evaling, or binding and forking, that functional.

Another problem concerns the creation of new processes as part of the activity of an application of a functional. A process is said to be spawned by an application of a functional if either it is a process forked during that application, or it is a process forked at any time by a process which was spawned by that application. That is, a spawned process would not have been created had the application not taken place.

The problem arises in defining the duration of an application. In the single process model, an application finished with the functional's return. However, in the extended model processes spawned by an application can mutate items and tokens in the input or output of that application after its primary activation has returned.

An easy way out is to say that any such behaviour constitutes a violation of the special behaviours. That is, items and tokens enclosed in the inputs and outputs of an application of a specially-behaved functional shall not be mutated by processes spawned by that application after that application has returned. If they are, the functional is not specially-behaved.

This solution is adopted primarily because no simple intuitions exist concerning the special behaviours when spawned processes can have effects after the application. This lack of intuition probably reflects a lack of experience with concurrent evaluation. Few programming languages even

permit programmers to experiment with mutation following return. PL/I for instance, kills all processes spawned by an application when the primary evaluation of that application returns.

Another problem concerns applications of the same functional which have overlapping durations. For sequence-repeatability, this causes difficulties in defining when two applications of a functional follow one another. In the single-process model, this could only happen by a recursive application, in which case one duration was a subset of the other. As such, they were presumed to be properly co-ordinated. Therefore, the subordinate applications were ignored entirely.

In the extended model, however, overlapping durations, and interlaced evaluation can occur. The problem is to define, for a group of applications with overlapping durations, which applications "follow" which. Firstly, if an application of a functional causes another evaluation of itself, either in the same or a spawned process, then that other evaluation is considered to be part of the causing activation. It is therefore ignored. Secondly, all possible permutations of the remaining applications are used to define the ordering of the set of applications in the total sequence of applications of the functional. Each of these total sequences is called a history of the functional. The model may define a large collection of histories for a functional. Using this notion, sequence-repeatability can be redefined: If a functional in some state R is sequence-repeatable, then for any similarly-vulnerable functional in some state S, there must exist a pair of histories (one for each functional) which extend similarities as described for sequence-repeatability in the single process model.

In effect, this requires of a sequence-repeatable functional that if it is applied with overlapping durations it arrange that its behaviour be the same as it would have been for one of the possible orderings of those applications made with non-overlapping durations.

Another problem is mutable entities (cells, semaphores, and tokens) in the inputs to functionals. If, during an application of a functional, an unrelated concurrent process updates cells in the input, then the

functional cannot be held responsible for not extending the now-destroyed input similarity. If during the application, semaphores in the input are subjected to uses of the PO's p and v by unrelated processes, failures to finish, or uncoordinate uses of cells, may result. If tokens are determined asynchronously, failures to finish may result non-repeatably. A process is said to be unrelated to an application of a functional if it is neither the process carrying out that application nor a process spawned by that application.

To save writing, the PO's update, p and v will be said to mutate cells and semaphores, and determining a token will be said to mutate that token.

So, specially-behaved functionals are required to extend input similarities only when unrelated process do not mutate mutable items or tokens in the similar inputs to those functionals during their applications.

As an example, the functional REVERSE of Section 6.2 is repeatable under the definition extended with these assumptions and requirements.

Do the classes of functionals which satisfied the unextended definitions of the special behaviours satisfy these extended definitions? To answer this, it is first necessary to say what the classes are in the extended model. Limited is extended to exclude fork as well as identical, install, and new-key. Then proper is defined as before: functionals all of whose enclosed functionals are limited.

For repeatability, the answer is yes: the proper functionals have no uses of the PO fork, so no spawned processes result. Consequently, the requirement concerning delayed mutation is trivially met. The cell-free proper functionals extend similarities provided that input cells, semaphores, and tokens are not mutated by unrelated processes during applications of the functionals.

For sequence-repeatability, the answer is no. The functional f of Figure 6.5.1 which is invulnerable and proper and therefore sequence-repeatable in the unextended model is not sequence-repeatable in the extended model. The trouble is that concurrent applications of the

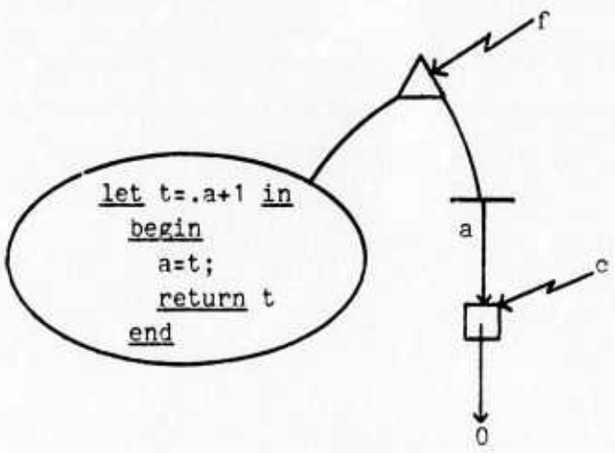


Figure 6.5.1. An invulnerable, proper functional which is not sequence-repeatable under the extended definitions.

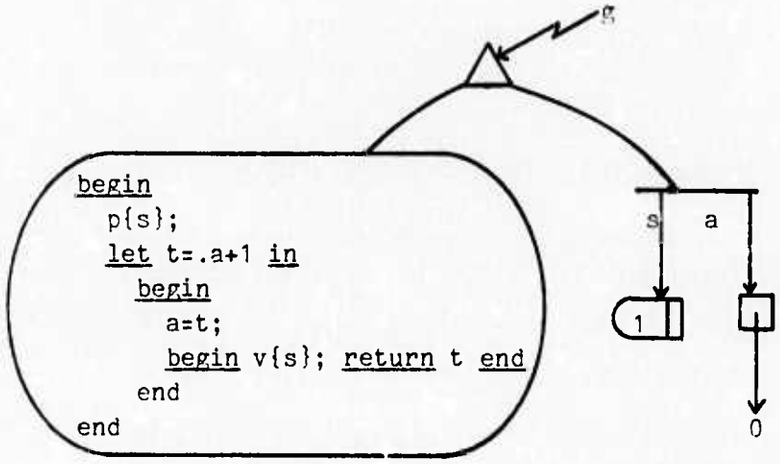


Figure 6.5.2. Co-ordinated version of the functional of Figure 6.5.1.

functional do not co-ordinate their uses of the cell c enclosed in the functional. Thus contents can be taken twice before updating is done. For example, if the first two applications were concurrent, it is possible that t in both activations would denote 1. A later third application would declare t denoting 2. However, the first three applications of some similar functional might not overlap. In that case, t would be declared respectively as 1, 2, and 3 in these activations.

The functional g of Figure 6.5.2 shows an obvious solution to this problem. By using a mutual-exclusion semaphore, all applications of g are effectively ordered. Thus g is sequence-repeatable under the extended definitions.

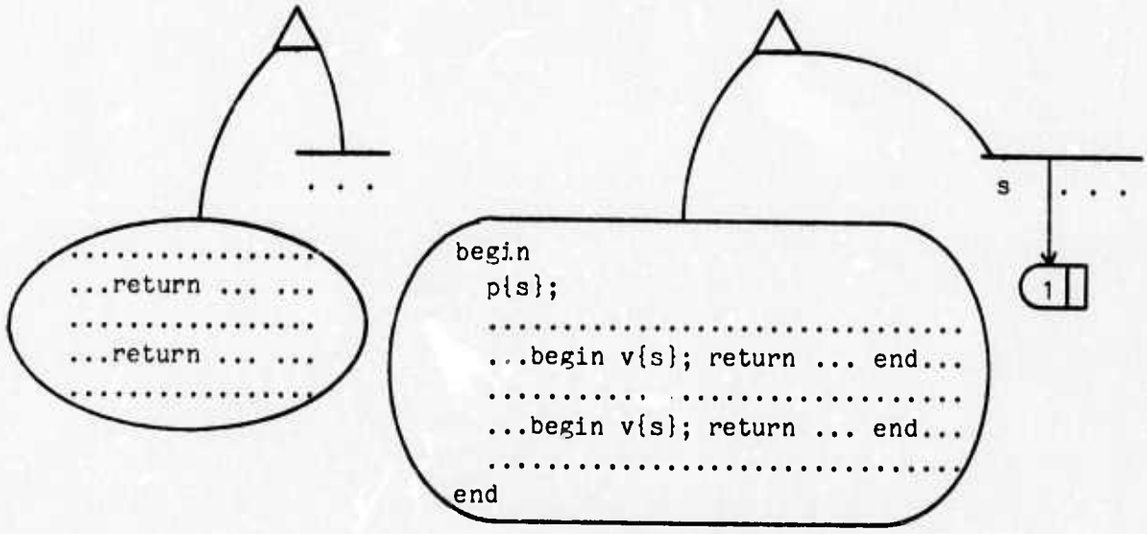


Figure 6.5.3. Co-ordinating a functional.

Generalizing, consider the set of all invulnerable proper functionals modified in the same way that f of Figure 6.5.1 was modified to get g. This modification is demonstrated schematically in Figure 6.5.3.

Care must be taken with functionals which are recursive. The modified functional which is made available to users must be bound to the unmodified functional for its own use. Otherwise in applying itself a functional will end up waiting for itself, enqueued on a semaphore that will never be v'd.

The version of a functional resulting from this modification will be called the "co-ordinated" version of that functional. The set of co-ordinated invulnerable proper functionals is a large set of structurally-defined functionals which are sequence-repeatable under the extended definition.

The other results of Chapter 5 fare in the extension as follows: A repeatable functional is still sequence-repeatable. A cell-free sequence-repeatable functional is repeatable. Binding a cell-free proper

item to a repeatable functional does not necessarily yield a repeatable functional. This lemma must be generalized to replace "cell" with "cell, semaphore, or token"; that is, building a "mutation-free" proper item to a repeatable functional yields a repeatable functional. Binding an invulnerable proper mutable item or token to a repeatable functional does not necessarily yield a sequence-repeatable functional. However, the co-ordinated version of that functional is sequence-repeatable. binding an invulnerable proper item to a sequence-repeatable functional yields a sequence-repeatable functional provided that item is immutable; the co-ordinated version of the functional is sequence repeatable if the item is mutable.

All the open questions of Chapter 5 may also be asked for the extended model.

Chapter 7. Thoughts on Implementation.

Included in the goals for the research which were set out in Chapter 1 was the requirement that the model be implementable using reasonable extensions of existing hardware and software. This chapter discusses programming techniques which lend credance to the statement that this goal can be met.

An obvious point of departure is to use the formal VDL definition as a guideline to implementation. However, that definition was not created for that purpose. Whenever a choice between ease of implementation and clarity arose, the latter criterion was favoured. Consequently the formal definition suggests some techniques which if followed in an implementation would lead to grave inefficiencies.

Often in the design of the formal definition, however, there was no conflict in between implementation and clarity. In those places the formal definition can be followed.

The following sections discuss the implementation of the various constructs in the Binding Model. Where following the formal definition is reasonable, representations for the VDL objects of that definition are suggested. Where the mechanisms of the formal definition are impractical, alternative mechanisms are proposed.

7.1. References.

The references of the model constitute a single global namespace through which all naming of items is effected. They are copied and manipulated extensively. Thus their representations should be small.

One way to represent references on word-oriented hardware is as single words. A short field (4 bits) would be used for type; the rest would be ndex.

An alternative scheme is to use variable length representations of references. The index portion of the reference could vary in length to suit needs. The following sections discuss where this could be an advantage.

7.2. Elementary Items.

The identity of elementary items which are alike (see Section 2.2) is guaranteed in the formal definition by searches over the elementary type-lists when "creating" "new" elementary items. This technique is an impractical method of achieving this guarantee. Because elementary items are immutable, they may be copied, so long as some means of implementing the PO equal is provided.

Further, it is possible to make a copy of an item for each reference to that item, and then to include that copy in the reference. Thus, for example, a reference to an integer would have, in place of an index into the type-list for integers, the integer itself.

If the length of a reference is 64 bits, the index part of the reference is 60 bits. Only 1 bit is needed to represent a logical. 60 bits provide an adequate integer range. But if characters are represented with 8 bits, 60 bits provides only 7-character symbols. This is inadequate.

One solution is to go to 128 bit references. The resulting 15-character limit on symbols is possibly adequate. Alternatively, short (less than 8-character) symbols could be stored in the reference, and long symbols could be stored in an abbreviated type-list.

A third way is to use variable length representations of references. Logicals could be very short; integers and symbols could be as long as required. If this is done, integers can have unbounded range. The cost of

copying references to integers and symbols would vary with length; programmers who used long items would pay for them.

7.3. Keys.

No use is ever made of the information content (the VDL empty list, <>) which appears for an individual key in the type-list for keys. Instead, the identity of keys is captured in the indices of references to keys. Consequently, the type-list for keys need not be implemented.

However, to generate the indices of new keys, the definition uses the length of the type-list for keys. Therefore, if the type-list for keys is not implemented, it must be replaced by an indication of the highest numbered index assigned to a key. This indicator is incremented whenever a key is created.

7.4. Non-selector Items.

It has been noted that type-lists for elementary items and keys need not be implemented because their information content can be included in references to them. This technique cannot be used with non-selector items (sheaves, cells, functionals, bundles, and semaphores): cells and semaphores are mutable, so they may not be copied, and references to self-referential items would have unbounded size because they would have to include themselves. Also the information content of these items can be arbitrarily large; repeated copying of references would be expensive if they were to include all that information.

An implementation of the model would therefore represent a non-selector item and references to it as separate entities. This technique generates the need for a mechanism to locate an item from a reference to it. That is, the reference context (see Section 2.3) must be explicitly implemented.

The VDL definition "implements" the reference context with type-lists. This technique makes every item equally easy to access. However such uniformity is not generally necessary: over any reasonably short interval of the evaluation, the items accessed by each process in a large programming system implemented in the model are only a very small subset of all the items in the states of that interval.

It would therefore be beneficial to establish a graduated scheme of availability for items. Much thought has been given to an analogous problem arising in existing operating systems: How does one provide efficient graduated availability for pages of files which are implemented in multi-level storage? The study of graduate availability for "item-sized" entities in an operating system is a topic of current research. It will not be discussed further here. However, it must be noted that the viability of the model is directly dependent upon finding a satisfactory solution of this problem.

7.5. Tokens.

The global search used in the model to determine a token can be avoided by implementing tokens as write-once "indirect" words. (That they are write-once permits them to be implemented using hardware which would not support cells — for example, optical laser-exposed film storage.) Potential references provide access to those words. Determining a token is done by writing a reference to the determination into the indirect word. Potential references are not altered when a token is determined. Because tokens may not be determined using tokens, there is no possiblility of "indirect chains" developing. Therefore accessing an item through a potential reference always is effected by accessing the indirect word to obtain the regular reference stored there, and then accessing the item using that reference. The implementation must arrange that such accessing should cause an error if nothing has been written in the word.

It is possible to regard the use of an indirect word as a technique for dealing with the transient condition of not knowing where in the system the

potential references for a token are. The location of potential references is discovered when they are used to locate an item. When this happens, the potential reference can be replaced with the regular reference stored in the indirect word. When all potential references have been replaced (see the next section), the indirect word can be discarded. This will happen only if all potential references are eventually used to access the items to which they have been determined to refer.

7.6. Deleting Items.

The formal definition of the model never discards anything. In practice, however, since the retention of items uses resources, it is desirable to discard items as soon as there is no chance that a process of the model will ever access them again.

A conservative test for this condition is developed by considering in any state the union of the closures of all items to which identifiers are bound in the environments of all activations of all processes in that state. If an item is not in that union, then it will certainly never be accessed again[1]. This condition requires computing closures of items. In general the time needed to make such computations is dependent upon the number of items in the state. Such "global" computations are impractical in existing systems.

However there are techniques which can be used to partially solve this problem. For example, if selector items are represented only in their references (see Sections 7.2 and 7.3), then discarding all references to a selector automatically discards the item.

---

1. The converse is false, for an item may be in this union because an identifier which will never be used again is bound to it in some environment. To discover that such items may also be discarded, an analysis of the control structures of the activations of the processes must be done. The question of whether an item is no longer necessary is therefore undecidable.

For non-selector items, reference counts can be kept. These counts are set at one when the item is created, are incremented when a reference is copied, and decremented when a reference is destroyed. When the reference count of an item reaches zero the item may be discarded. This implements a "conservative" retention strategy. It does so at the cost of locating an item every time references to it are manipulated. The evaluation of this trade-off depends on the expense of locating an item from its reference.

The reference count deletion scheme is said to be conservative because it fails to detect all the items which can be deleted. If all references to a set of items are included in items of that set, that set is said to be disconnected. An item in a disconnected set may be deleted, yet its reference count may not be zero. In particular, discarding all references to al invulnerable self-referential item makes the closure of that item into a disconnected set. So self-referential items will never be deleted using only a reference count scheme.

One approach to the problem of undiscovered disconnected sets is to ignore it, and be prepared to provide more storage as items collect. If a multi-level virtual memory is being used, the older an item gets, the slower will be the level of memory in which it will be kept. If an item is indeed in a disconnected set, then the only cost of retaining it unnecessarily is the price of storing it on magnetic tapes (say).

Another approach is to search the (low levels of) memory for undiscovered disconnected sets. In general this is hard, because processes may be changing the patterns of reference as the search is going on. In the model, however, only the references in cells can change. Therefore any searches on non-cell items need not worry about processes changing things.

Thus the hardest problem involved in deleting items is finding the cycles which include cells. Various forms of interlocking between the garbage collector and the processes of the model can be imagined. To aid in this activity, additional fields can be added to those representations of cells used in the slower levels of memory.

The design and implementation of such deletion schemes is an open problem. As noted above, it is related to the task of implementing the reference context. It is currently under investigation.

7.7. Control Structures and Environments.

The task of translating a control structure representation (CSR) is either trivial or extensive: If hardware is constructed to help, translation could consist merely of getting the items in the CSR near each other in the physical memory. As the items of CSR's are immutable, they could be copied into a compact form.

If existing hardware were to be used, the tree structured form would probably be linearized. Jump or subroutine instructions could be used to achieve iteration and conditionals.

Environments could be implemented using associative memory. Such facilities are not common in existing hardware, however. For this reason, control structures have been designed so that environments of activations can be implemented in conventional memory with a conventional stack discipline. Identifiers can be compiled out; that is, they can be replaced in control structures by offsets from the base of a stack frame. The translation would have to assign offsets to both external and local identifiers. The local identifiers could then be forgotten. The external identifiers would be retained so that, in binding, partial stack frames could be prepared with references at appropriate offsets. On starting an evaluation, the partial stack frame of a functional would be copied onto the stack, and the instructions representing the control structure executed.

7.8. Practical Computing.

This section mentions a number of additions which an implementor would probably wish to make to the model to improve its speed. Many of these additions were parts of earlier versions of the model. They were left out of the version presented here because they presented no new intellectual content, and therefore would have served only to add unnecessary volume to this report. This section has also been included to demonstrate how the model has been "trimmed down" for the purpose of this study.

Additional elementary types would be desirable: floating point and interval arithmetic for example. Also, additional PO's for the existing selector types are needed: greater-than, greater-than-or-equal, etc.

A short-cut PO would save building sheaves one component at a time. This generalized form of augment would take a sheaf and 2n additional operands which are n selector/component pairs. It would yield a single augmented sheaf without constructing all the intermediate sheaves. An analogous short-cut PO which binds functionals would be equally useful. Notice that both of these short-cuts require making PO's be able to take a variable number of arguments.

Other short-cut PO's which might be useful are those which create specialized structures of items. For example,

row{n,x}

might yield a row of length n each component of which is x, while

array{n,x}

might generate a row of length n each component of which is a cell whose contents is x.

An extension to permit references to elementaries in control structures would also be a practical extension to the model. These elementary literals correspond to the data in the "immediate" instructions of current hardware.

The next chapter discusses a number of other additions to the model which are also needed to make it a machine that is practical for real

programming. In contrast to the additions of this section, those of the next chapter are not well understood; they are suggested as topics for further study.

Chapter 8. Conclusions.

The goal of this work, as set out in Chapter 1, has been to design an abstract machine which is an ideal modular programming system. The scope of the work has been limited by restricting the requirements which have been adopted to characterize "ideal". This chapter examines the Binding Model, as defined and studied in Chapters 2 through 7, to see whether it meets these requirements.

There follows a review of the sources of the ideas underlying the design of the Binding Model.

Finally, this chapter discusses topics for further study. These arise from two sources: The open questions of Chapters 5, 6, and 7, and the design of other features which might make the model a "more ideal" modular programming system.

8.1. Review of the Goals.

Section 1.2 set out the features desirable in an ideal modular programming system. These included the requirements of modularity, and some further requirements adopted to define "ideal".

Intuitively, a computer system is modular if, within it, programs employing existing programs can be constructed using only knowledge of the behaviour of those programs. This intuition was developed in Section 1.1 into four requirements. The following paragraphs review these.

The first requirement of modularity is that the system must provide mechanisms which permit any module to employ any other. In any programming system constructed from the Binding Model, functionals fill the role of module. The PO <u>bind</u> is used to employ one functional in another. The PO's

bind, and eval or fork, permit invocation of an employed module. Sharing of modules and data is made possible through the use of a namespace of references and a single context in which references are resolved to items. This context is managed by the model, thus avoiding conflicts in the binding of references. Bindings in this context are immutable, thus guaranteeing the reliability of the use of references for naming items. As the demonstration system (DS) of Section 6.3 illustrated, systems can be created in the model which support a collection of programmers who can share modules.

The second requirement of modularity is that the employer of a module should not have to worry about the pieces which that module needs in order to realize its behaviour. That is, every module should be self-sufficient. In the Binding Model, it can be arranged that items which are needed by a functional can be enclosed in it through the use of binding. Additional binding and evaluation can be carried out using the functional with no need to be aware of the earlier binding. Thus functionals are self-sufficient.

The third requirement of modularity is that no conflict should arise when modules are used together. That is, any set of modules must be compatible. Since in the Binding Model the environment of each functional is a separate context, and since any identifier can be bound to any item, any set of items may be enclosed in the same environment provided only that a different identifier is bound to each. This compatability of modules results primarily because there is never a requirement to combine contexts.

The last requirement of modularity is that the behaviour of a module be the same for all employers of that module. That is, modules in the system must be non-discriminatory. In the Binding Model, a functional's behaviour depends solely upon its control structure and the items to which identifiers in its environment are bound. It is not dependent upon which other functional invokes eval or fork to cause that functional to be evaluated. Thus functionals are non-discriminatory.

In addition to the requirements for modularity, this research has adopted four further requirements to characterize an "ideal" programming system. These requirements were developed in Section 1.2. The next few

paragraphs review the model in light of these further requirements.

The first requirement of an ideal programming system is that it should support sophisticated computations. Designing and choosing the primitive types for the model was a major part of the research reported here. Each type is included to provide some facility needed to support sophisticated programming. As far as possible, the types serve independent purposes, reflecting the independence of the facilities needed. However this was not always possible; certain primitive types interact with one another to provide subtle but important effects. The achievement of synergistic interaction of the primitive types of the model was an important part of this work.

In the model, logic and arithmetic are supported by logical and integer items. Sheaves support structuring. Cells support mutability. Functionals support procedures and functions. Symbols support naming, and in conjunction with sheaves support text manipulation. Bundles support restricted access. The control structures of functionals support sequencial, conditional, and iterative computations. Compute expressions support self-reference.

The second requirement for an ideal programming system is that it should support multiple concurrent processes. Chapter 6 showed in detail how the Binding Model does this.

The third requirement for an ideal programming system is that it should support definitions of the special behaviours which mirror intuitions about the special behaviours. Also there must exist large, structurally-defined classes of modules which satisfy those definitions. Chapter 5 and Section 6.5 developed such definitions for the Binding Model.

The fourth requirement for an ideal programming system is that the machine should be implementable by reasonable extensions of current hardware. Chapter 7 discusses techniques for implementing the Binding Model. It concludes that implementing the model involves solving some problems which are topics of current study. However it is anticipated that current research will soon indicate how these problems can be solved; if

this is so, efficient implementation of the model will become possible with extensions to current hardware.

8.2. Review of Related Work.

In essence, the contributions of this work are two in number. Firstly, there is the design of the Binding Model. It is a coherent, self-consistent organization of certain existing ideas. Consequently it provides a fresh view of some aspects of programming languages and operating systems. Secondly, taking advantage of this fresh view, there is the examination of the special behaviours. This study gives a formal basis to, among other things, the conventional wisdom concerning the dependency of behaviour on time: if modules don't make use of any internal mutable storage then they will behave the same way every time they are invoked.

This section reviews the ideas underlying the design of the Binding Model so that this work may find in its proper place in the history of computer science.

The idea of a single system-managed namespace for items is an essential part of any graph model. It is implicit in the capabilities of [Dennis, 66] and [Vanderbilt, 69] and in the acyclic graphs of [Dennis, 68]. It is explicitly mentioned in the discussion of implementing capabilities in the CAL Time Sharing System [Lampson, 69].

The elementary types are found in most programming languages (see, for example, ALGOL 60 and LISP). Sheaves correspond to the non-elementary nodes of [Dennis, 68], the STRUCT'S of, for example, ALGOL 68 [Van Wijngaarden, 69], and ECL [Wegbreit, 72], and the structure definitions of [Landin, 65].

Keys are similar to Morris' seals [Morris, 73], and the creation of bundles with keys as extractors is similar to the use of such seals to create sealed objects. The concept of keys is somewhat more general than the mechanisms of Morris, for the those mechanisms can be effected using keys and bundles. Bundles were also developed with the idea of

representing instances of particular data types by clusters of operations, an idea arising from the work on the language CLU [Liskov, 74].

The combining of program and data as is done to create functionals occurs as the creation of "closures" in a number of programming languages (see [Landin, 65], and [Moses, 70]) and in the system designed by Vanderbilt [Vanderbilt, 69]. Incremental binding is provided in ECL [Wegbreit, 74] and in the "fixed parameters" of operators in the CAL Time-Sharing System [Lampson, 69].

Cells occur implicitly in the descriptions of most programming languages (see, for example, the "objects" of EL1 [Wegbreit, 74]). They are discussed explicitly in describing the programming languages PAL [Wozencraft, 71], ALGOL 68 [Van Wijngaarden, 69], GEDANKEN [Reynolds, 70].

Semaphores are well known [Dijkstra, 65b].

The notion of putting all these items at the same level and making that level the system level is suggested by the APEX file system [Stowe, 66] and Dennis' acyclic graphs [Dennis, 72]. However, the Binding Model introduces cells at the system level for the first time.

The forms chosen for control structures follow those of Landin [Landin, 65], PAL [Wozencraft, 71], and work on structured programming [Dahl, 72]. In particular, the "compute" expression in its simple form is patterned after the "valof/res" constructs of PAL. The notion of tokens, and the full compute expression are new, however. The return of multiple items was suggested by thoughts on the SUE System Language [Clark, 71]. Both the independence of processes from one another, and their separation from the graph of items, follow Vanderbilt [Vanderbilt, 69].

8.3. Topics for Further Study.

A number of areas for further work arise as loose ends of the research reported here. Chapter 5 lists a number of open questions concerning aspects of the special behaviours. Of particular interest are the well-definedness of the maximal well-behaved set, and the equivalence of classes of functionals. Chapter 6 re-raises these same questions for the version of the model extended to support concurrent processes. Chapter 7 leaves open the critical implementation question mentioned in Section 8.1.

There are also a number of extensions to the model which could prove fruitful areas of research. The following questions need answering before the model is really practical.

How should input and output be added to the model? The discussion of the Demonstration System (see Section 6.3) suggested that this be done by adding to the model some new primitive type. What should its design be? Can, and should, it also be used between processes to support inter-process communication? If so, how would it interact with the interprocess-communication through shared cells discussed in Chapter 6?

Can processes be interrogated? If not, how is debugging accomplished? If so, how does such interrogation affect the restricted access achieved by functionals and bundles?

As discussed in Sections 3.8 and 5.8, the model can represent instances of programmer-defined types. Can, or should, programmer-defined types themselves be supported at the primitive level of the model?

Should cells be typed?

How should error handling mechanisms be added to the model? Is an error an item of some sort, or is it a control concept (for example, a non-normal return, or inter-process communication)?

Process in the model are independent and uninterruptible. Should some means of interrupting them be provided? What is an interrupt, and how does it work? Can user-intervention be modelled as an interrupt? How? Should

it be possible to destroy a process? What are the effects of interrupts on modularity and the special behaviours?

As the model supports modularity, it is natural for the creators of modules to want compensation for the sharing and use of them. What extensions to the model are needed to support the marketing and sale of the services of modules?

With extensive sharing, how are charges made for the use of system resources, and how are such usages controlled when those resources are limited? What affects do these decisions have upon modularity and the special behaviours?

Inevitably humans make errors. Thus references to sensitive items will inevitably get into the wrong hands. Can this be policed by the system? What is the role of extra-system intervention in destroying such references?

It is inevitable that hardware will fail. Most existing systems meet the challenge of serious hardware failure by dividing the system into "volatile" and "non-volatile" parts. When a system "crashes", only the volatile parts are lost. This leaves the non-volatile parts from which to re-construct a version of the system which is a reasonable approximation to the one existing before the crash. How does all this relate to implementing the Binding Model? Should the model be augmented with some form of "check-point" facilities, or should the programs of a system constructed in the model do that themselves? If the latter, how are such programs written? In either case, how is check-pointing conceptualized and implemented?

The Binding Model is a simple, yet sophisticated, base for the study of these questions.

Bibliography.

[Amerasinghe, 72]
Amerasinghe, S. N., The Handling of Procedure Variables in a Base Language, M.S. Thesis, Department of Electrical Engineering, MIT (1972).

[Boebart, 65] Boebart, W. E., "Toward a modular programming system", Modular Programming: Proceedings of a National Symposium, Symposium Preprint, Barnett, T. O. (ed.), Information and Systems Press, Cambridge, Massachusetts, (1968). [Out of business].

[Clark, 71] Clark, B. L., and Horning, J. J., "The System Language for Project SUE", SIGPLAN Notices 6,9 (October 1971) 79-88.

[Clingen, 69] Clingen, C.T., "Program Naming Problems in a Shared Tree-structured Hierarchy", NATO Science Committee Conference on Techniques in Software Engineeering, 1, Rome, Italy (October, 1969).

[Crisman, 65] Crisman, P. A., (ed.), The Compatible Time-Sharing System: A Programmer's Guide, 2nd ed., MIT Press, Cambridge, Massachusetts (1965).

[Dahl, 72] Dahl, O-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, London, England (1972).

[Dennis, 66] Dennis, J. B., and Van Horn, E. C., "Programming Semantics for Multi-programmed Computations", CACM 9,3 (march 1966) 143-155.

[Dennis, 68] Dennis, J. B., Programming Generality, Parallelism, and Computer Architecture, Computation Structures Group Memo 32, Project MAC, MIT (August 1968).

[Dennis, 71] Dennis, J. B., "Name Management", An Undergraduate Course on Operating Systems Principles, Interim Report, COSINE Committee, Committee on Education, National Academy of Engineering, Washington, D.C. (June 1971) Module 5.

[Dennis, 72] Dennis, J. B., Modularity, Computation Structures Group Memo 70, Project MAC, MIT (June 1972).

[Dijkstra, 65a]
Dijkstra, E. W., "Programming Considered as a Human Activity", Proc. IFIPS , Spartan Books, Washington, D.C. (1965) 213-217.

[Dijkstra, 65b] Dijkstra, E. W., Cooperating Sequential Processes, Report EWD123, Department of Mathematics, Technological University at Eindhoven, The Netherlands (1965).

[Earley, 73a] Earley, J., Relational Level Data Structures for Programming Languages, Department of Computer Science, University of California, Berkeley, California (1973).

[Earley, 73b] Earley, J., Naming Structure and Modularity in Programming Languages, Technical Report 17, Department of Computer Science, University of California, Berkeley, California (1973).

[Ellis, 74] Ellis, D. J., Semantics of Data Structures and References, MAC-TR-134, Project MAC, MIT (August 1974).

[Falkoff, 68] Falkoff, A. D., Iverson, K. E., APL/360: User's Manual, IBM Corp. (August 1968).

[Forgie, 65] Forgie, J. W., "A Time- and Memory-Sharing Executive Program for Quick-Response On-line Applications", Proc. FJCC, AFIPS Press, Montvale, New Jersey (1965).

[Hammer, 69] Hammer, M. M., and Jorrand, P., The formal definition of BASEL, CA-6908-1511, Massachusetts Computer Associates, Wakefield, Massachusetts (1969).

[Henderson, 67] Henderson, Jr., D. A., Graphics Display Language, M.S. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois (1967).

[Henderson, 69] Henderson, Jr., D. A., A Description and Definition of Simple AMBIT/G - a graphical programming language, CA-6904-2811, Massachusetts Computer Associates, Wakefield, Massachusetts (April 1969).

[Johnson, 71] Johnson, J. B., "The Contour Model of Block Structured Processes", SIGPLAN Notices 6,2 (February 1971) 55-82.

[Lampson, 69] Lampson, B. W., An Overview of the CAL Time Sharing System, Computer Center, University of California, Berkeley, California (1969).

[Landin, 65] Landin, P. J., A Correspondence between ALGOL 60 and Church's Lambda notation: Part I" CACM 8,2 (February 1965) 80-100, and "Part II", CACM 8,3 (March 1965) 158-165.

[Liskov, 72] Liskov, B. H., "A Design Methodology for Reliable Software Systems", Proc. FJCC , AFIPS Press, Montvale, New Jersey (1972) 191-199.

[Liskov, 74] Liskov, B. H., and Zilles, S. N., "Programming with Abstract Data Types", SIGPLAN Notices 9,4 (April 1974) 50-59.

[Lucas, 68] Lucas, P., Lauer, P., Stigleitner, H., Method and Notation for the Formal Definition of Programming Languages, TR25.087, IBM Vienna Laboratory (June 1968).

[McCarthy, 62] McCarthy, J., et al., The LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts (August 1962).

[Lomet, 73a] Lomet, D. B., A Cellular Storage Model for Programming Languages, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1973).

[Lomet, 73b] Lomet, D. B., An Operator Driven Model Of Program Execution, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1973).

[Morris, 72] Morris, Jr., J. H., "Protection in Programming Languages", CACM 16,1 (January 1973) 15-21.

[Moses, 70] Moses, J., "The Function of FUNCTION in LISP", SIGSAM Bulletin (July 1970) 13-27.

[Neuhold, 71] Newhold, E. J., "The Formal Description of Programming Languages", IBM Systems Journal 10,2 (1971) 86-112.

[Organick, 72] Organick, E. I., The Multics System: An Examination of its Structure, MIT Press, Cambridge, Massachusetts (1972).

[Parnas, 72a] Parnas, D. L., "A Technique for Software Module Specification with Examples", CACM 15,5 (May 1972). 330-336.

[Parnas, 72b] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", CACM 15,12 (December 1972) 1053-1058.

[Reynolds, 70] Reynolds, J. C., "GEDANKEN- A Typeless Language based on the Principle of Completeness and the Reference Concept", CACM 13,5 (May 1970) 308-319.

[Rhodes, 73] Rhodes, J., "Tackle Software with Modular Programming", Computer Decisions, (October 1973) 21-25.

[Rovner, 69] Rovner, P. D., and Henderson, Jr., D. A., "An Implementation of AMBIT/G: a graphical programming language", Proc. ICAI , Washington, D.C. (May 1969).

[Stowe, 66] Stowe, A. N., Wiesen, R. A., Yntema, D. B., and Forgie, J. W., "The Lincoln Reckoner: An Operation-Oriented On-line Facility with Distributed Control", Proc. FJCC, AFIPS Press, Montvale, New Jersey (1966). 433-444.

[Sutherland, 71] Sutherland, W. R., Myer, T. H., Thomas, E. L., Henderson, Jr., D. A., "A Route Oriented Simulation System for Air Traffic Control Studies", Proc. Fifth Conference on Applications of Simulation (December 1971).

[Thomas, 72] Thomas, R. H., and Henderson, Jr. D. A., "McROSS - a Multi-Computer Programming System", Proc. SJCC (1972) 281-293.

[Vanderbilt, 69] Vanderbilt, D. H., Controlled Information Sharing in a Computer Utility, MAC-TR-67, Project MAC, MIT (1969).

[Van Wijngaarden, 69] Van Wijngaarden, A., (ed.), Report On the Algorithmic Language ALGOL 68, MR 101, Mathematisch Centrum, Amsterdam, The Netherlands (February 1969).

[Wegbreit, 72] Wegbreit, B., The ECL Programmer's Manual, 21-72, Center for Research in Computing Technology, Harvard University (1972).

[Wegbreit, 74] Wegbreit, B., et al., "Procedure Closure in EL1", The Computer Journal 17,1 (February 1974) 38-43.

[Wozencraft, 71] Wozencraft, J. M., Evans, JR., A., Notes on Programming Linguistics, Department of Electrical Engineering, MIT (February 1971).

[Wulf, 74] Wulf, W., et al., "HYDRA: The Kernel of a Multiprocessor Operating System", CACM 17,6 (June 1974) 337-345.

[Zilles, 75] Zilles, S. N., Data Algebra: A Specification Technique for Data Structures, Ph.D Thesis, Department of Electrical Engineering, MIT (forthcoming).

Appendix A. Tables.

This appendix tabulates the primitive operations (PO's) in various ways: those which create, yield or manipulate items of each type or group of types (selectors, and any), those which can manipulate tokens, and all of them alphabetically. For convenience, it also reviews the forms of control structure reprentations by including tables of control directives and the expressions.

The following abbreviations are used to indicate the nine primitive types, the two groups of types, and the group of all items and tokens.

log	logical
int	integer
sym	symbol
key	key
shf	sheaf
cel	cell
fnl	functional
bnd	bundle
sem	semaphore
sel	selector (log,int,sym,key)
itm	item (log,int,sym,key,shf,cel,fnl,bnd,sem,)
any	any (log,int,sym,key,shf,cel,fnl,bnd,sem,token)

Type constraints of PO's are given using the following syntax:

PO-name{argument types} → result types

The notation "any*" means a non-negative number of items or tokens.

A.1. PO's for Logicals.

Created:

not{log} → log
and{log,log} → log
or{log,log} → log
equal{sel,sel} → log
less-than{sel,sel} → log
same{cell,cell} → log
is-selector{shf,sel} → log
is-extractor{bnd,sel} → log
is-logical{itm} → log
is-integer{itm} → log
is-symbol{itm} → log
is-key{itm} → log
is-sheaf{itm} → log
is-cell{itm} → log
is-functional{itm} → log
is-bundle{itm} → log
is-semaphore{itm} → log
identical{itm,itm} → log

Used:

not{log} → log
and{log,log} → log
or{log,log} → log
(also used as test item in conditional control directives)

A.2. PO's for Integers.

Created:

add{int,int} → int
subtract{int,int} → int
multiply{int,int} → int
divide{int,int} → int
length{shf} → int

Used:

add{int,int} → int
subtract{int,int} → int
multiply{int,int} → int
divide{int,int} → int
selector{shf,int} → sel
new-semaphore{int} → sem

A.3. PO's for Symbols.

Created:
compose{shf} → sym

Used:
decompose{sym} → shf
bind{fnl,sym,any} → fnl
(also used in control structure represention)

A.4. PO's for Keys.

Created:
new-key{} → key

Used:
none

A.5. PO's for Sheaves.

Created:
decompose{sym} → shf
empty-sheaf{} → shf
augment{shf,sel,any} → shf

Used:
compose(shf) → sym
augment{shf,sel,any} → shf
is-selector{shf,sel} → log
select{shf,sel} → any
length{shf} → int
selector{shf,int} → sel
install{shf} → fnl
new-bundle{shf} → bnd
(also used in control structure representations)

A.6. PO's for Cells.

Created:

new-cell{any} → cel

Used:

contents{cel} → any
update{cel,any} →

A.7. PO's for Functionals.

Created:

install{shf} → fnl
bind{fnl,sym,any} → fnl

Used:

bind{fnl,sym,any} → fnl
eval{fnl} → any*
fork{fnl} →

A.8. PO's for Bundles.

Created:

new-bundle{snf} → bnd

Used:

is-extractor{bnd,sel} → log
extract{bnd,sel} → any

A.9. PO's for Semaphores.

Created:

new-semaphore{int} → sem

Used:

psem{sem} →
vsem{sem} →

A.10. PO's for Selectors.

Created:

selector{shf,int} → sel

Used:

equal{sel,sel} → log
less-than{sel,sel} → log
augment{shf,sel,any} → shf
is-selector{shf,sel} → log
select{shf,sel} → any
is-extractor{bnd,sel} → log
extract{bnd,sel} → any

A.11. PO's for Items.

Created:

none

Used:

is-logical{itm} → log
is-integer{itm} → log
is-symbol{itm} → log
is-key{itm} → log
is-sheaf{itm} → log
is-cell{itm} → log
is-functional{itm} → log
is-bundle{itm} → log
is-semaphore{itm} → log
identical{itm,itm} → log

A.12. PO's for Items and Tokens.

Created:

select{shf,sel} → any
contents{cel} → any
eval{fnl} → any*
extract{bnd,sel} → any

Used:

augment{shf,sel,any} → shf
new-cell{any} → cel
update{cel,any} →
bind{fnl,sym,any} → fnl

A.13. Alphabetical List of PO's.

```
add{int,int} → int
and{log,log} → log
augment{shf,sel,any} → shf
bind{fnl,sym,any} → fnl
compose{shf} → sym
contents{cel} → any
decompose{sym} → shf
divide{int,int} → int
empty-sheaf{} → shf
equal{sel,sel} → log
eval{fnl} → any*
extract{bnd,sel} → any
fork{fnl} →
identical{itm,itm} → log
install{shf} → fnl
is-bundle{itm} → log
is-cell{itm} → log
is-extractor{bnd,sel} → log
is-functional{itm} → log
is-integer{itm} → log
is-key{itm} → log
is-logical{itm} → log
is-selector{shf,sel} → log
is-semaphore{itm} → log
is-sheaf{itm} → log
is-symbol{itm} → log
is-extractor{bnd,sel} → log
is-selector{shf,sel} → log
length{shf} → int
less-than{sel,sel} → log
multiply{int,int} → int
new-bundle{shf} → bnd
new-cell{any} → cel
new-key{} → key
new-semaphore{int} → sem
not{log} → log
or{log,log} → log
p{sem} →
same{cell,cell} → log
select{shf,sel} → any
selector{shf,int} → sel
subtract{int,int} → int
update{cel,any} →
v{sem} →
```

A.14. Control Structure Representation.

Figures A.14.1 and A.14.2 give the forms for control directives and expressions respectively. All identifiers and PO names are represented by the corresponding symbols. Placeholders are represented by empty sheaves.

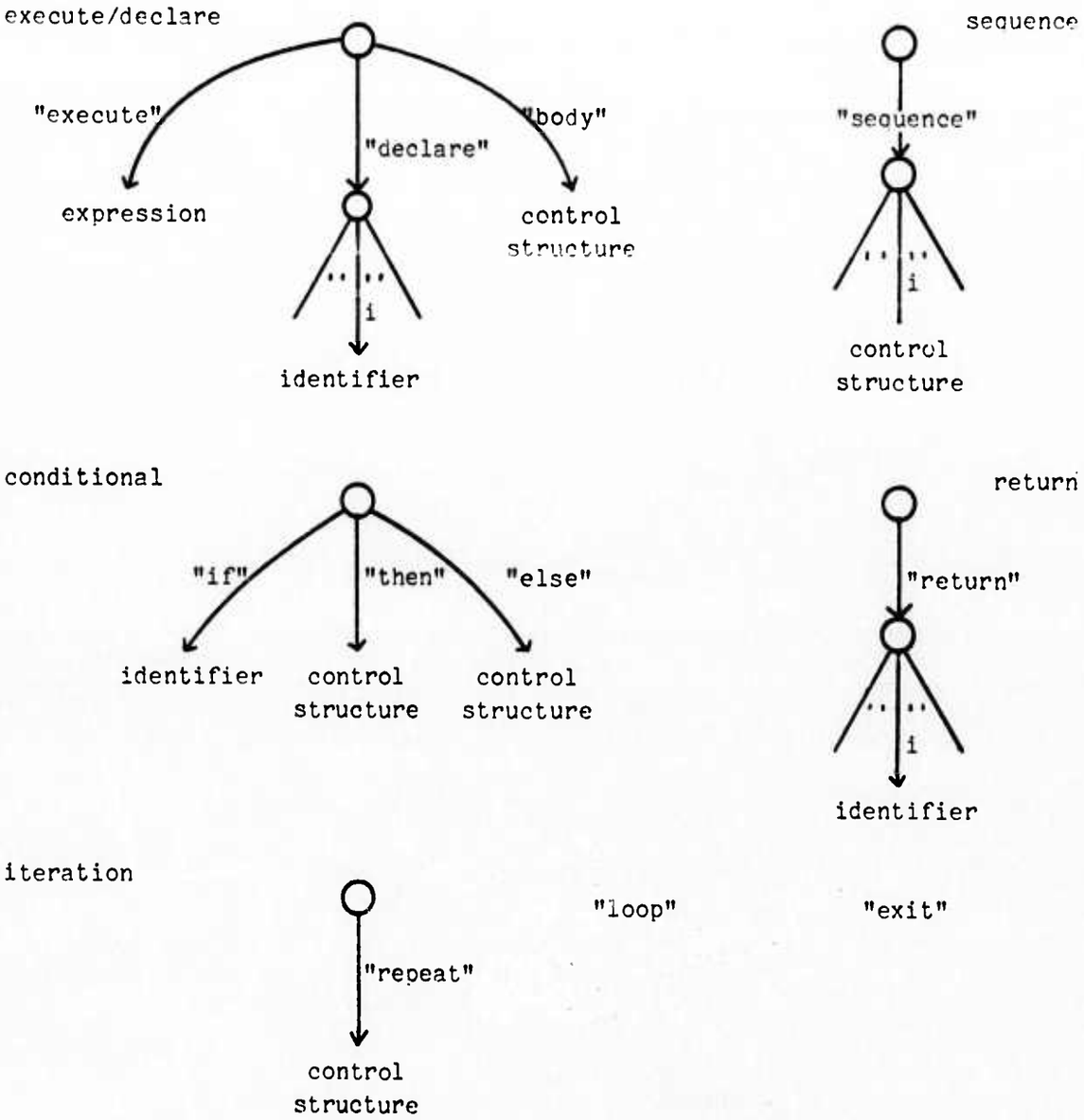


Figure A.14.1. Forms for control directives.

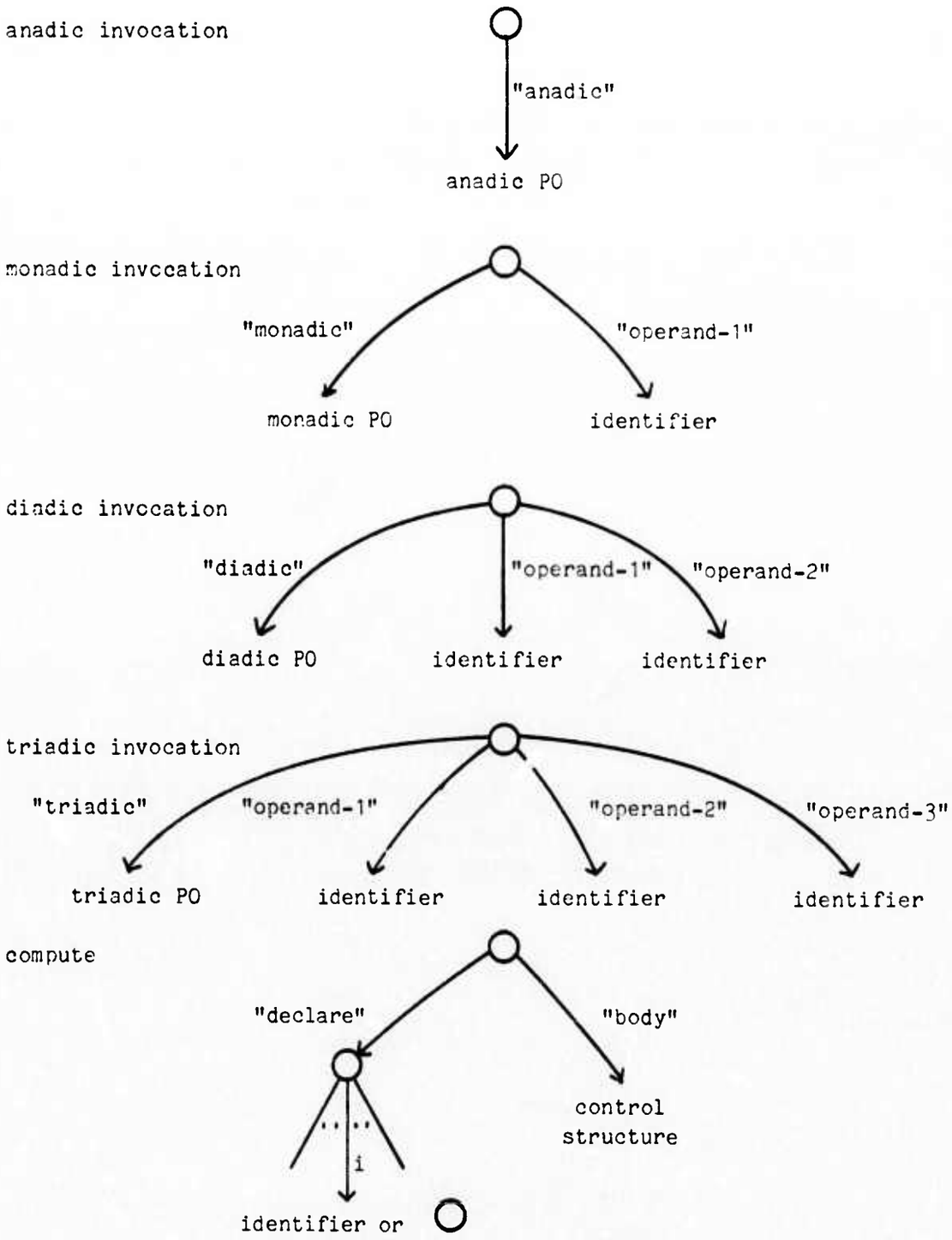


Figure A.14.2. The forms for expressions.

Appendix B. VDL Predicates.

This appendix gives the primitive and derived predicates used in defining the objects and instruction schemata of the formal VDL definition of the Binding Model.

The forms of the objects used in the definition are discussed and illustrated in chapters 4 and 6 of this report. The instruction schemata of the definition are given in Appendix C and are also discussed in chapters 4 and 6.

This appendix includes the predicates necessary for the formal definitions of both the single-process and extended versions of the model. Those predicates or parts of predicates which are needed only for the extended version are enclosed in brackets [ ].

B.1. States.

```
is-state=          (<s-items:     is-items>,
                    <s-tokens:    is-<>-list>,
                    <s-processes: is-processes>,
                    <s-c:         is-ct>)
```

B.2. Items and References.

```
  is-items=          (<s-logicals:    is-log-list>,
                      <s-integers:    is-log-list>,
                      <s-symbols:     is-sym-list>,
                      <s-keys:        is-<>-list>,
                      <s-sheaves:     is-shf-list>,
                      <s-cells:       is-ref-list>,
                      <s-functionals: is-fnl-list>,
                      <s-bundles:     is-shf-list>,
[                     <s-semaphores:  is-sem-list>                    ]
                           )

  is-sym=            is-char-list

  is-shf=            is-sc-list

  is-fnl=            (<s-cs:          is-cs>,
                      <s-environment: is-affiliation-list>)

[ is-sem=            (<s-lock:         is-log>,                      ]
[                     <s-count:        is-int>,                      ]
[                     <s-next-process: is-int>)                      ]

  is-sc=             (<s-selector:  is-sel-ref>,
                      <s-component: is-ref>)

  is-affiliation=    (<s-identifier:  is-symbol>,
                      <s-reference:   is-ref>)


  is-ref=            is-pref or is-reg-ref

  is-pref=           (<s-type:is-rtptok>,<s-index:is-int>)

  is-reg-ref=        is-sel-ref or is-comp-ref

  is-sel-ref=        is-logical or is-integer
                       or is-symbol or is-key

  is-comp-ref=       is-sheaf or is-cell
                         or is-functional or is-bundle
[                          or is-semaphore                            ]

  is-logical=        (<s-type:is-rtplog>,<s-index:is-int>)

  is-integer=        (<s-type:is-rtpint>,<s-index:is-int>)

  is-symbol=         (<s-type:is-rtpsym>,<s-index:is-int>)
```

```
  is-key=            (<s-type:is-rtpkey>,<s-index:is-int>)
  is-sheaf=          (<s-type:is-rtpshf>,<s-index:is-int>)
  is-cell=           (<s-type:is-rtpcel>,<s-index:is-int>)
  is-functional=     (<s-type:is-rtpfnl>,<s-index:is-int>)
  is-bundle=         (<s-type:is-rtpbnd>,<s-index:is-int>)
[ is-semaphore=      (<s-type:is-rtpsem>,<s-index:is-int>)                ]
```

B.3. Control Structures and Directives.

```
  is-cs=             is-execution or is-sequence
                       or is-conditional or is-iteration
                         or is-loop or is-exit or is-return
  is-execution=      (<s-execute: is-expression>,
                      <s-declare: is-identifier-list>,
                      <s-body:    is-cs>)
  is-sequence=       (<s-sequence: is-cs-list>)
  is-conditional=    (<s-if:   is-identifier>,
                      <s-then: is-cs>,
                      <s-else: is-cs>)
  is-iteration=      (<s-repeat: is-cs>)
  is-loop=           is-ref-loop
  is-exit=           is-ref-exit
  is-return=         (<s-return: is-identifier-list>)
```

B.4. Expressions.

```
is-anadic=          (<s-anadic: is-anadic-op>)

is-monadic=         (<s-monadic:   is-monadic-op>,
                     <s-operand-1: is-identifier>)

is-diadic=          (<s-monadic:   is-monadic-op>,
                     <s-operand-1: is-identifier>,
                     <s-operand-2: is-identifier>)

is-triadic=         (<s-monadic:   is-monadic-op>,
                     <s-operand-1: is-identifier>,
                     <s-operand-2: is-identifier>,
                     <s-operand-3: is-identifier>)

is-compute=         (<s-declare: is-opidentifier-list>,
                     <s-body:    is-cs>)
```

B.5. Identifiers.

```
is-opidentifier=    is-identifier or is-<>

is-identifier=      is-symbol
```

B.6. Processes and Activations.

```
is-processes=       (<s-processes: is-process-list>,
[                    <s-done:      is-int>                    ]
                          )

is-process=         (<s-activations:  is-activation>,
                     <s-task:         is-ref>,
[                    <s-next-process: is-int>                 ]
                          )

is-activation=      (<s-environment: is-affiliation-list>,
                     <s-activation:  is-activation>)
                    or is-null
```

B.7. Control Directive Representations.

```
is-execution-rep=    (<elem(1):(<s-selector:is-ref-body>,
                                <s-component:is-ref>)>,
                      <elem(2):(<s-selector:is-ref-declare>,
                                <s-component:is-ref>)>,
                      <elem(3):(<s-selector:is-ref-execute>,
                                <s-component:is-ref>)>)

is-sequence-rep=     (<elem(1):(<s-selector:is-ref-sequence>,
                                <s-component:is-ref>)>)

is-conditional-rep=  (<elem(1):(<s-selector:is-ref-else>,
                                <s-component:is-ref>)>,
                      <elem(2):(<s-selector:is-ref-if>,
                                <s-component:is-ref>)>,
                      <elem(3):(<s-selector:is-ref-then>,
                                <s-component:is-ref>)>)

is-iteration-rep=    (<elem(1):(<s-selector:is-ref-repeat>,
                                <s-component:is-ref>)>)

is-return-rep=       (<elem(1):(<s-selector:is-ref-return>,
                                <s-component:is-ref>)>)
```

B.8. Expression Representations.

```
is-anadic-rep=      (<elem(1):(<s-selector:is-ref-anadic>,
                               <s-component:is-ref>)>)

is-monadic-rep=     (<elem(1):(<s-selector:is-ref-monadic>,
                               <s-component:is-ref>)>,
                     <elem(2):(<s-selector:is-ref-operand-1>,
                               <s-component:is-ref>)>)

is-diadic-rep=      (<elem(1):(<s-selector:is-ref-diadic>,
                               <s-component:is-ref>)>,
                     <elem(2):(<s-selector:is-ref-operand-1>,
                               <s-component:is-ref>)>,
                     <elem(3):(<s-selector:is-ref-operand-2>,
                               <s-component:is-ref>)>)

is-triadic-rep=     (<elem(1):(<s-selector:is-ref-triadic>,
                               <s-component:is-ref>)>,
                     <elem(2):(<s-selector:is-ref-operand-1>,
                               <s-component:is-ref>)>,
                     <elem(3):(<s-selector:is-ref-operand-2>,
                               <s-component:is-ref>)>,
                     <elem(4):(<s-selector:is-ref-operand-3>,
                               <s-component:is-ref>)>)

is-compute-rep=     (<elem(1):(<s-selector:is-ref-body>,
                               <s-component:is-ref>)>,
                     <elem(2):(<s-selector:is-ref-declare>,
                               <s-component:is-ref>)>)
```

B.9. Primitive Operations.

```
  is-anadic-op=     is-ref-new-key          or is-ref-empty-sheaf

  is-monadic-op=    is-ref-not              or is-ref-compose
                or  is-ref-decompose        or is-ref-new-key
                or  is-ref-length           or is-ref-new-cell
                or  is-ref-contents         or is-ref-install
                or  is-ref-cval             or is-ref-new-bundle
                or  is-ref-is-logical       or is-ref-is-integer
                or  is-ref-is-symbol        or is-ref-is-key
                or  is-ref-is-sheaf         or is-ref-is-cell
                or  is-ref-is-functional    or is-ref-is-bundle
[               or  is-ref-fork             or is-ref-p                 ]
[               or  is-ref-new-semaphore    or is-ref-v                 ]
[               or  is-ref-is-semaphore                                 ]

  is-diadic-op=     is-ref-equal            or is-ref-and
                or  is-ref-or               or is-ref-add
                or  is-ref-subtract         or is-ref-multiply
                or  is-ref-divide           or is-ref-less-than
                or  is-ref-select           or is-ref-selector
                or  is-ref-is-selector      or is-ref-update
                or  is-ref-same             or is-ref-is-extractor
                or  is-ref-extract          or is-ref-identical

  is-triadic-op=    is-ref-bind             or is-ref-augment
```

B.10. Result Structures.

```
  is-rs=            is-normal-rs or is-loop-rs
                or  is-exit-rs   or is-return-rs

  is-normal-rs=     (<s-signal:is-signormal>)

  is-loop-rs=       (<s-signal:is-sigloop>)

  is-exit-rs=       (<s-signal:is-sigexit>)

  is-return-rs=     (<s-signal:is-sigreturn>,
                     <s-results:is-ref-list>)
```

B.11. Distinguished Symbols.

The pattern for distinguished symbols is demonstrated with the predicate for the symbol "less-than" .

is-ref-less-than= (<s-type:rtpsym>,<s-index:i>)

where (if S is the state),

elem(i)^s-symbols^s-items(S)= (<elem(1):is-l>,
<elem(2):is-e>,
<elem(3):is-s>,
<elem(4):is-s>,
<elem(5):is-->,
<elem(6):is-t>,
<elem(7):is-h>,
<elem(8):is-a>,
<elem(9):is-n>)

The other distinguished symbols are:

execute, declare , sequence, if, then, else, repeat, loop, exit, return

anadic, monadic, diadic, triadic, operand-1, operand-2, operand-3, compute, body

new-key, empty-sheaf, not, compose, decompose, length, new-cell, contents, install, eval, new-bundle, is-logical, is-integer, is-symbol, is-key, is-sheaf, is-cell, is-functional, is-bundle

equal, and, or, add, subtract, multiply, divide, less-than, select, selector, is-selector, update, same, is-extractor, extract, identical

bind, augment

[fork, new-semaphore, p, v, is-semaphore]

B.12. Primitive Predicates.

```
is-log=          the logicals (true and false)

is-int=          the integers

is-char=         is-a or is-b or is-c or ...
                  ... or is-x or is-y or is-z
                 or is-- or is-. or ... etc...

is-rtp=          is-rtplog or is-rtpint or is-rtpsym
                 or is-rtpkey or is-rtpshf or is-rtpcel
                 or is-rtpshf or is-rtpfnl or is-rtpbnd
[                or is-rtpsem                                ]
                 or is-rtptok

is-sig=          is-signormal or is-sigloop
                 or is-sigexit or is-sigreturn
```

Appendix C. VDL Instruction Schemata.

This appendix contains the instruction schemata for the formal definitions of two versions of the Binding Model. The version with many processes (see Chapter 6) is an extension of the version with a single process (see Chapter 4). The extensions to the simpler definition are enclosed in brackets [ ] in this appendix. Chapter 4 discusses the single process definition. Chapter 6 explains the extensions.

To simplify the typography of this report, the Greek characters which are used in the standard notation for VDL definitions have been replaced with upper case Roman characters, or distinguished words. Some special symbols of the standard notation have also been replaced. Figure C.0.1 tabulates these replacements.

ξ	S	state
α	A	compound selector shorthand
β	B	compound selector shorthand
γ	C	compound selector shorthand
π	P	compound selector shorthand
χ	X	compound selector shorthand
Ω	null	the VDL null object
μ_0	M	VDL object constructor
$\sim$	not	operation for logical values
$\wedge$	and	operation for logical values
$\vee$	or	operation for logical values
o	^	compound selector concatenator

Figure C.0.1. Replacements in VDL notations.

C.1. Begin and End.

Names of routines have been chosen to indicate what the routines do. There are several cases where a collection of routines is need, one each for every element of some set of things. The names of the routines in these collections are similar. These cases are:

1. eval-xxx for control directives,expressions and PO's.
2. translate-xxx for representations of control directives, expressions, and identifiers.
3. make-xxx for delayed references and primitive types.
4. get-xxx for primitive types.
5. check-xxx for symbols, selector items as a group, and rows.

Parameter names are relatively short, but are also chosen for mnemonic value. They reflect the structure or semantics of the objects expected. Figure C.0.2 is a list of these names.

Where more than one item of some form is used in a single schema, these abbreviations are suffixed with an integer. For example, in the schema eval-equal in Section C.7, elem1 and elem2 appear. Prefixes are also used, as in the abbreviation nref for "new reference" used in many of the routines for evaluating PO's.

Many instruction schemata have the process number "i" as a parameter.

log	-	logical item
int	-	integer item
sym	-	symbol item
key	-	key item
shf	-	sheaf item
cell	-	cell item
fnl	-	functional item
bnd	-	bundle item
elem	-	elementary item
sel	-	selector item
ref	-	reference
pref	-	potential reference
opref	-	optional reference (ref or <>)
oppref	-	optional potential reference (pref or <>)
rs	-	result structure
sig	-	signal
cs	-	control structure
exp	-	expression
id	-	identifier
pos	-	primitive operation symbol
env	-	environment

Figure C.0.2. Parameter names.

```
[ run-processes(i)=                                                    ]
[   C(S)=length(P(S))→ null                                            ]
[   i≤length(P(S))→ run-process(i)                                     ]
[                   run-processes(i+1)                                 ]
[   true→ run-processes(i)                                             ]
[     with C=s-done^s-processes                                        ]
[          P=s-processes^s-processes                                   ]

  run-process(i)=
    true→ null
[           finish()                                                   ]
              throw-away(rs)
                rs:eval-cs(i,s-cs^fnl)
                  initialize-process(i,s-environment(fnl))
                    fnl:get-functional(A(S))
      with A=s-task^elem(i)^P
           P=s-processes^s-processes
```

```
  initialize-process(i,env)=
    s-environment^B: env
    s-activation^B: null
[   s-next-process^elem(i)^P: -1                                    ]
      with B=s-activations^elem(i)^P
           P=s-processes^s-processes

  throw-away(rs)=
    is-normal-rs(rs)→ null
    is-exit-rs(rs)→ error
    is-loop-rs(rs)→ error
    is-return-rs(rs)→ null

[ finish()=                                                         ]
[   C: C(S)+1                                                       ]
[     with C=s-done^s-processes                                     ]
```

C.2. Evaluate a Control Structure.

```
  eval-cs(i,cs)=
    is-execution(cs)→ eval-execution(i,cs)
    is-sequence(cs)→ eval-sequence(i,cs)
    is-conditional(cs)→ eval-conditional(i,cs)
    is-iteration(cs)→ eval-iteration(i,cs)
    is-loop(cs)→ eval-loop(i,cs)
    is-exit(cs)→ eval-exit(i,cs)
    is-return(cs)→ eval-return(i,cs)
```

C.2.1

```
  eval-execution(i,cs)=
    true→ pass(rs1)
           reset-environment(i,env)
             rs1:eval-cs(i,s-body(cs))
               declare-set(i,s-declare(cs),s-results(rs))
                 env:current-environment(i)
                   rs:eval-expression(i,s-execute(cs))

  declare-set(i,opids,oprefs)=
    length(opids)≠length(oprefs)→ error
    true→ declare-set-rest(i,opids,oprefs,1)
```

```
declare-set-rest(i,opids,oprefs,j)=
  j>length(opids)→ null
  true→ declare-set-rest(i,opids,oprefs,j+1)
         declare(i,elem(j)(opids),elem(j)(oprefs))

declare(i,opid,opref)=
  opid=<>→ null
  is-symbol(opid)→ declare-1(i,opid,opref,k)
                   k:search(s-environment^A(S),
                            s-identifier,
                              s-identifier(opid))
  true→ error
    with A=s-activations^elem(i)^P
         P=s-processes^s-processes

declare-1(i,opid,opref,k)=
  k≠null→ error
  true→ elem(1+length(A(S)))(A(S)): M(<s-identifier:opid>,
                                      <s-reference:opref>)
    with A=s-environment^s-activations^elem(i)^P
         P=s-processes^s-processes

current-environment(i)=
  PASS: s-environment^A(S)
    with A=s-activations^elem(i)^P
         P=s-processes^s-processes

reset-environment(i,env)=
  s-environment^A: env
    with A=s-activations^elem(i)^P
         P=s-processes^s-processes
```

C.2.2

```
eval-sequence(i,cs)=
  true→ eval-seq-elem(i,s-sequence(cs),1)

eval-sequence-elem(i,cs,j)=
  j≤length(cs)→ eval-sequence-rest(i,cs,j,rs)
                rs:eval-cs(i,elem(j)(cs))
  true→ pass(M(<s-signal:signormal>))

eval-sequence-rest(i,cs,j,rs)=
  is-normal-rs(rs)→ eval-sequence-elem(i,cs,j+1)
  true→ pass(rs)
```

C.2.3

```
eval-conditional(i,cs)=
  true→ eval-conditional-1(i,cs,log)
          log:get-logical(ref)
            ref:eval-identifier(i,s-if(cs))

eval-conditional-1(i,cs,log)=
  log→ eval-cs(i,s-then(cs))
  true→ eval-cs(i,s-else(cs))
```

C.2.4

```
eval-iteration(i,cs)=
  true→ eval-iteration-rest(i,cs,rs)
          rs:eval-cs(i,s-repeat(cs))

eval-iteration-rest(i,cs,rs)=
  is-normal-rs(rs)→ eval-iteration(i,cs)
  is-exit-rs(rs)→ pass(M(<s-signal:signormal>))
  is-loop-rs(rs)→ eval-iteration(i,cs)
  is-return-rs(rs)→ pass(rs)

eval-loop(i,cs)=
  PASS: M(<s-signal:sigloop>)

eval-exit(i,cs)=
  PASS: M(<s-signal:sigexit>)
```

C.2.5

```
eval-return(i,cs)=
  s-return(cs)=<>→ return-0-refs()
  true→ pass(M(<s-signal:sigreturn,s-results:refs>))
         {elem(j)(refs):eval-identifier(i,
                                        elem(j)^s-return(cs))
                                    | 1≤j≤length(s-return(cs))}
```

C.3. Evaluate an Expression.

```
eval-expression(i,exp)=
  is-anadic(exp)→ eval-anadic(i,exp)
  is-monadic(exp)→ eval-monadic(i,exp)
  is-diadic(exp)→ eval-diadic(i,exp)
  is-triadic(exp)→ eval-triadic(i,exp)
  is-compute(exp)→ eval-compute(i,exp)
```

C.3.1

```
eval-anadic(i,exp)=
  true→ eval-anadic-1(i,s-anadic(exp))

eval-anadic-1(i,pos)=
  is-ref-empty-sheaf(pos)→ [illegible]1-empty-sheaf()
  is-ref-new-key(pos)→ [illegible]w-key()
```

C.3.2

```
  eval-monadic(i,exp)=
    true→ eval-monadic-1(i,s-monadic(exp),ref)
           ref:eval-identifier(i,s-operand-1(exp))

  eval-monadic-1(i,pos,ref)=
    is-ref-not(pos)→ eval-not(ref)
    is-ref-compose(pos)→ eval-compose(ref)
    is-ref-decompose(pos)→ eval-decompose(ref)
    is-ref-length(pos)→ eval-length(ref)
    is-ref-new-cell(pos)→ eval-new-cell(ref)
    is-ref-contents(pos)→ eval-contents(ref)
    is-ref-install(pos)→ eval-install(ref)
    is-ref-eval(pos)→ eval-eval(i,ref)
    is-ref-new-bundle(pos)→ eval-new-bundle(ref)
    is-ref-is-logical(pos)→ eval-is-logical(ref)
    is-ref-is-integer(pos)→ eval-is-integer(ref)
    is-ref-is-symbol(pos)→ eval-is-symbol(ref)
    is-ref-is-key(pos)→ eval-is-key(ref)
    is-ref-is-sheaf(pos)→ eval-is-sheaf(ref)
    is-ref-is-cell(pos)→ eval-is-cell(ref)
    is-ref-is-functional(pos)→ eval-is-functional(ref)
    is-ref-is-bundle(pos)→ eval-is-bundle(ref)
[   is-ref-fork(pos)→ eval-fork(ref)                                    ]
[   is-ref-new-semaphore(pos)→ eval-new-semaphore(ref)                  ]
[   is-ref-p(pos)→ eval-p(i,ref)                                        ]
[   is-ref-v(pos)→ eval-v(i,ref)                                        ]
[   is-ref-is-semaphore(pos)→ eval-is-semaphore(ref)                    ]
```

C.3.3

```
eval-diadic(i,exp)=
  true→ eval-diadic-1(i,s-diadic(exp),ref1,ref2)
        ref1:eval-identifier(i,s-operand-1(exp))
        ref2:eval-identifier(i,s-operand-2(exp))

eval-diadic-1(i,pos,ref1,ref2)=
  is-ref-equal(pos)→ eval-equal(ref1,ref2)
  is-ref-and(pos)→ eval-and(ref1,ref2)
  is-ref-or(pos)→ eval-or(ref1,ref2)
  is-ref-add(pos)→ eval-add(ref1,ref2)
  is-ref-subtract(pos)→ eval-subtract(ref1,ref2)
  is-ref-multiply(pos)→ eval-multiply(ref1,ref2)
  is-ref-divide(pos)→ eval-divide(ref1,ref2)
  is-ref-less-than(pos)→ eval-less-than(ref1,ref2)
  is-ref-is-selector(pos)→ eval-is-selector(ref1,ref2)
  is-ref-select(pos)→ eval-select(ref1,ref2)
  is-ref-selector(pos)→ eval-selector(ref1,ref2)
  is-ref-same(pos)→ eval-same(ref1,ref2)
  is-ref-update(pos)→ eval-update(ref1,ref2)
  is-ref-is-extractor(pos)→ eval-is-extractor(ref1,ref2)
  is-ref-extract(pos)→ eval-extract(ref1,ref2)
  is-ref-identical(pos)→ eval-identical(ref1,ref2)
```

C.3.4

```
eval-triadic(i,exp)=
  true→ eval-triadic-1(i,s-triadic(exp),ref1,ref2,ref3)
        ref1:eval-identifier(i,s-operand-1(exp))
        ref2:eval-identifier(i,s-operand-2(exp))
        ref3:eval-identifier(i,s-operand-3(exp))

eval-triadic-1(i,pos,ref1,ref2,ref3)=
  is-ref-bind(pos)→ eval-bind(ref1,ref2,ref3)
  is-ref-augment(pos)→ eval-augment(ref1,ref2,ref3)
```

C.3.5

```
eval-compute(i,exp)=
  true→ eval-compute-1(i,env,opprefs,rs)
        rs:eval-cs(i,s-body(exp))
          declare-set(i,s-declare(exp),opprefs)
            opprefs:get-opprefs(s-declare(exp))
            env:current-environment(i)

eval-compute-1(i,env,opprefs,rs)=
  is-normal-rs(rs)→ eval-compute-1(i,env,opprefs,rs1)
                    rs1:return-0-refs()
  is-exit-rs(rs)→ error
  is-loop-rs(rs)→ error
  is-return-rs(rs)→ pass(rs)
                    reset-environment(i,env)
                      determine-opprefs(opprefs,s-results(rs))

get-opprefs(opids)=
  opids=<>→ pass(<>)
  true→ pass(opprefs)
        {elem(j)(opprefs):get-oppref(elem(j)(opids))
                        | 1≤j≤length(opids)}

get-oppref(opid)=
  opid=<>→ pass(<>)
  true→ pass(pref)
        pref:make-token()

determine-opprefs(opprefs,refs)=
  length(refs)≠length(opprefs)→ error
  true→ {determine-oppref(elem(j)(opprefs),elem(j)(refs))
                        | 1≤j≤length(opprefs)}

determine-oppref(oppref,ref)=
  oppref=<>→ null
  s-pref(ref)→ error
  true→ {X: ref | X(S)=oppref}
```

C.4. Evaluate an Identifier.

```
eval-identifier(i,id)=
  true→ eval-identifier-1(i,index)
         index:search(s-environment^A(S),s-identifier,id)
    with A=s-activations^elem(i)^P
         P=s-processes^s-processes

eval-identifier-1(i,index)=
  index=null→ error
  true→ pass(s-reference^elem(index)^s-environment^A(S))
    with A=s-activations^elem(i)^P
         P=s-processes^s-processes
```

C.5. Evaluate Anadic PO's.

```
eval-new-key()=
  true→ return-1-ref(nref)
         nref:make-key()

eval-empty-sheaf()=
  true→ return-1-ref(nref)
         nref:make-sheaf(<>)
```

C.6. Evaluate Monadic PO's.

```
eval-not(ref)=
  true→return-1-ref(nref)
       nref:make-logical(not(log))
         log:get-logical(ref)

eval-compose(ref)=
  true→ return-1-ref(nref)
         nref:make-symbol(sym)
           sym:compose-rest(shf,<>,1)
             shf:get-sheaf(ref)
               check-row(ref)
```

```
compose-rest(shf,sym,j)=
  j>length(shf)→ pass(sym)
  true→ compose-rest(shf,sym2,j+1)
         syms:concatenate(sym,sym1)
           sym1:get-symbol(s-component^elem(j)(shf))

concatenate(sym1,sym2)=
  true→ pass(M({<elem(j):elem(j)(sym1)> ¦ 1≤j≤length(sym1)},
              {<elem(j+length(sym1)):elem(j)(sym2)>
                                  ¦ 1≤j≤length(sym2)}))

eval-decompose(ref)=
  true→ return-1-ref(nref)
         nref:make-sheaf(shf)
           shf:list(sym)
             sym:get-symbol(ref)

list(sym)=
  true→ pass(shf)
         {elem(j)(shf):M(<s-selector:make-integer(j)>,
                         <s-component:make-symbol(
                             M(<elem(1):elem(j)(sym)>))>)
                                      ¦ 1≤j≤length(sym)}

eval-length(ref)=
  true→ return-1-ref(nref)
         nref:make-integer(length(shf))
           shf:get-sheaf(ref)

eval-new-cell(ref)=
  true→ return-1-ref(nref)
           nref:make-cell(ref)

eval-contents(ref)=
  true→ return-1-ref(elem(index)^s-cells^s-items(S))
         index:get-cell-index(ref)

eval-install(ref)=
  is-sheaf(ref)→ return-1-ref(nref)
                   nref:make-functional(M(<s-text:cs>,
                                          <s-environment:<>>))
                     cs:translate-cs(ref)
  true→ error
```

```
eval-eval(i,ref)=
  true→ eval-eval-1(i,rs)
        rs:eval-cs(i,s-cs(fn1))
          push-activation(i,s-environment(fn1))
            fn1:get-functional(ref)

eval-eval-1(i,rs)=
  is-normal-rs(rs)→ eval-eval-1(i,rs1)
                          rs1:return-0-refs()
  is-exit-rs(rs)→ error
  is-loop-rs(rs)→ error
  is-return-rs(rs)→ pass(rs)
                          pop-activation(i)

push-activation(i,env)=
  A: M(<s-activation:A(S)>,
       <s-environment:env>)
    with A=s-activations^elem(i)^P
         P=s-processes^s-processes

pop-activation(i)=
  A: s-activation^A(S)
    with A=s-activations^elem(i)^P
         P=s-processes^s-processes

eval-new-bundle(ref)=
  true→ return-1-ref(nref)
        nref:make-bundle(shf)
          shf:get-sheaf(ref)

eval-is-logical(ref)=
  is-pref(ref)→ error
  true→ return-1-ref(nref)
        nref:make-logical(is-logical(ref))

eval-is-integer(ref)=
  is-pref(ref)→ error
  true→ return-1-ref(nref)
        nref:make-logical(is-integer(ref))

eval-is-symbol(ref)=
  is-pref(ref)→ error
  true→ return-1-ref(nref)
        nref:make-logical(is-symbol(ref))
```

```
eval-is-key(ref)=
  is-pref(ref)→ error
  true→ return-1-ref(nref)
         nref:make-logical(is-key(ref))

eval-is-sheaf(ref)=
  is-pref(ref)→ error
  true→ return-1-ref(nref)
         nref:make-logical(is-sheaf(ref))

eval-is-cell(ref)=
  is-pref(ref)→ error
  true→ return-1-ref(nref)
         nref:make-logical(is-cell(ref))

eval-is-functional(ref)=
  is-pref(ref)→ error
  true→ return-1-ref(nref)
         nref:make-logical(is-functional(ref))

eval-is-bundle(ref)=
  is-pref(ref)→ error
  true→ return-1-ref(nref)
         nref:make-logical(is-bundle(ref))

[ eval-fork(ref)=                                                    ]
[   is-functional(ref)→ return-0-refs()                              ]
[                       fork-process(ref)                            ]
[   true→ error                                                      ]

[ fork-process(ref)=                                                 ]
[   s-task^elem(length(P(S))+1)^P: ref                               ]
[     with P=s-processes^s-processes                                 ]

[ eval-new-semaphore(ref)=                                           ]
[   true→ eval-new-semaphore-1(int)                                  ]
[          int:get-integer(ref)                                      ]

[ eval-new-semaphore-1(int)                                          ]
[   int<0→ error                                                     ]
[   true→ return-1-ref(nref)                                         ]
[          nref:make-semaphore(M(<s-lock:true>,                      ]
[                               <s-count:int>,                       ]
[                               <s-next-process:0>))                 ]
```

```
[ eval-p(i,ref)=                                                          ]
[   true→ return-0-refs()                                                 ]
[          release-and-wait(i,index,cnt)                                  ]
[            cnt:decrement-semaphore(index)                               ]
[              lock(s-lock^elem(index)^s-semaphores^s-item)                ]
[                index:get-semaphore-index(ref)                           ]

[ decrement-semaphore(index)=                                             ]
[   PASS: C(S)-1                                                          ]
[   C: C(S)-1                                                             ]
[     with C=s-count^elem(index)^s-semaphores^s-item                      ]

[ release-and-wait(i,index,cnt)=                                          ]
[   cnt≥0→ unlock(C)                                                      ]
[   true→ wait(i)                                                         ]
[          unlock(C)                                                      ]
[            enqueue-process(i,index)                                     ]
[     with C=s-lock^elem(index)^s-semaphores^s-item                       ]

[ enqueue-process(i,index)=                                               ]
[   C(S)=0→ C: i                                                          ]
[            s-next-process^elem(i)^P: 0                                  ]
[   true→ put-after(i,pro)                                                ]
[          pro:end-of-queue(C(S))                                         ]
[     with C=s-next-process^elem(index)^s-semaphores^s-item               ]
[          P=s-processes^s-processes                                      ]

[ end-of-queue(pro)=                                                      ]
[   C(S)= 0→ pass(pro)                                                    ]
[   true→ end-of-queue(C(S))                                              ]
[     with C=s-next-process^elem(pro)^P                                   ]
[          P=s-processes^s-processes                                      ]

[ put-after(i,pro)=                                                       ]
[   s-next-process^elem(pro)^P: i                                         ]
[   s-next-process^elem(i)^P: 0                                           ]
[     with P=s-processes^s-processes                                      ]

[ wait(i)=                                                                ]
[   s-next-process^elem(i)^P(S)≥0→ wait(i)                                ]
[   true→ null                                                            ]
[     with P=s-processes^s-processes                                      ]
```

```
[ eval-v(i,ref)=                                                    ]
[   true→ return-0-refs()                                           ]
[          unlock(C)                                                ]
[            resume-process(index,cnt)                              ]
[              cnt:increment-semaphore(index)                       ]
[                lock(C)                                            ]
[                  index:get-semaphore-index(ref)                   ]
[     with C=s-lock^elem(index)^s-semaphores^s-item                  ]

[ increment-semaphore(index)=                                       ]
[   PASS: C(S)+1                                                    ]
[   C: C(S)+1                                                       ]
[     with C=s-count^elem(index)^s-semaphores^s-item                 ]

[ resume-process(index,cnt)                                         ]
[   cnt>0→ null                                                     ]
[   cnt≤0→ dequeue-process(index,C(S))                              ]
[     with C=s-next-process^elem(index)^s-semaphores^s-item         ]

[ dequeue-process(index,pro)=                                       ]
[   B: C(S)                                                         ]
[   C: -1                                                           ]
[     with B=s-next-process^elem(index)^s-semaphores^s-items        ]
[          C=s-next-process^elem(pro)^P                             ]
[          P=s-processes^s-processes                                ]

[ eval-is-semaphore(ref)=                                           ]
[   is-pref(ref)→ error                                             ]
[   true→ return-1-ref(nref)                                        ]
[          nref:make-logical(is-semaphore(ref))                     ]
```

C.7. Evaluate Diadic PO's.

```
eval-equal(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-logical(ref1 = ref2)
           check-selector(ref1)
           check-selector(ref2)

eval-and(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-logical(log1 and log2)
           log1:get-logical(ref1)
           log2:get-logical(ref2)

eval-or(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-logical(log1 or log2)
           log1:get-logical(ref1)
           log2:get-logical(ref2)

eval-add(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-integer(int1 + int2)
           int1:get-integer(ref1)
           int2:get-integer(ref2)

eval-subtract(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-integer(int1 - int2)
           int1:get-integer(ref1)
           int2:get-integer(ref2)

eval-multiply(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-integer(int1 * int2)
           int1:get-integer(ref1)
           int2:get-integer(ref2)

eval-divide(ref1,ref2)=
  true→ eval-divide-1(int1,int2)
         int1:get-integer(ref1)
         int2:get-integer(ref2)

eval-divide-1(int1,int2)=
  true→ return-2-refs(nref1,nref2)
         nref1:make-integer(int1 / int2)
         nref2:make-integer(int1 rem int2)
```

```
eval-less-than(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-logical(log)
           log.sort-less-than(ref1,ref2)
             check-selector(ref1)
             check-selector(ref2)

eval-is-selector(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-logical(index ≠ null)
           index:search(shf,s-selector,ref2)
             shf:get-sheaf(ref1)
             check-selector(ref2)

eval-select(ref1,ref2)=
  true→ eval-select-1(shf,index)
    0     index:search(shf,s-selector,ref2)
            shf:get-sheaf(ref1)
            check-selector(ref2)

eval-select-1(shf,index)=
  index=null→ error
  true→ return-1-ref(s-component^elem(index)(shf))

eval-selector(ref1,ref2)=
  true→ eval-selector-1(shf,int)
         shf:get-sheaf(ref1)
         int:get-integer(ref2)

eval-selector-1(shf,int)=
  int<1→ error
  int>length(shf)→ error
  true→ return-1-ref(s-selector^elem(int)(shf))

eval-same(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-logical(index1 = index2)
           index1:get-cell-index(ref1)
           index2:get-cell-index(ref2)

eval-update(ref1,ref2)=
  true→ return-0-refs()
         change-contents(index,ref2)
           index:get-cell-index(ref1)

change-contents(index,ref)=
  elem(index)^s-cells^s-items: ref
```

```
eval-is-extractor(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-logical(index≠null)
           index:search(bnd,s-selector,ref2)
             bnd:get-bundle(ref1)
             check-selector(ref2)

eval-extract(ref1,ref2)=
  true→ eval-extract-1(bnd,index)
         index:search(bnd,s-selector,ref2)
           bnd:get-bundle(ref1)
           check-selector(ref2)

eval-extract-1(bnd,index)=
  index=null→ error
  true→ return-1-ref(s-value^elem(index)(bnd))

eval-identical(ref1,ref2)=
  true→ return-1-ref(nref)
         nref:make-logical(ref1 = ref2)
```

C.8. Evaluate Triadic PO's.

```
eval-bind(ref1,ref2,ref3)=
  true→ eval-bind-1(fnl,ref2,ref3,index)
        index:search(s-environment(fnl),s-identifier,ref2)
          fnl:get-functional(ref1)
          check-symbol(ref2)

eval-bind-1(fnl,ref2,ref3,index)=
  index≠null→ error
  true→ return-1-ref(nref)
        nref:make-functional(
                M(<s-cs:s-cs(fnl)>,
                   <s-environment:
                      M({<elem(j):elem(j)^s-environment(fnl)>
                                 | 1≤j≤length(s-environment(fnl))},
                         <elem(length(s-environment(fnl))+1):
                            M(<s-identifier:ref2>,
                              <s-reference:ref3>)>)>) )

eval-augment(ref1,ref2,ref3)=
  true→ return-1-ref(nref)
        nref:make-sheaf(
                M({<elem(j):elem(j)(shf)> | 1≤j<n},
                   <elem(n):M(<s-selector:ref2>,<s-component:ref3>)>,
                    {<elem(j+1):elem(j)(shf)> | n≤j≤length(shf)})
          n:sort(shf,ref2)
            shf:get-sheaf(ref1)
            check-selector(ref2)

sort(shf,ref)=
  true→sort-after(shf,ref,1)

sort-after(shf,ref,j)=
  j>length(shf)→ pass(j)
  s-selector^elem(j)(shf)=ref→ error
  true→ sort-after-1(shf,ref,j,log)
        log:sort-less-than(ref,s-selector^elem(j)(shf))

sort-after-1(shf,ref,j,log)=
  log→ pass(j)
  true→ sort-after(shf,ref,j+1)
```

C.9. Ordering of Selector References.

```
sort-less-than(ref1,ref2)=
  ref1=ref2→ pass(false)
  true→ sort-less-than-1(ref1,ref2,tpref1,tpref2)
         tpref1:type-of(ref1)
         tpref2:type-of(ref2)

sort-less-than-1(ref1,ref2,tpref1,tpref2)=
  tpref1>tpref2→ pass(false)
  tpref1<tpref2→ pass(true)
  is-logical(ref1)→ pass(log)
                     log:get-logical(ref2)
  is-integer(ref1)→ pass(int1<int2)
                     int1:get-integer(ref1)
                     int2:get-integer(ref2)
  is-symbol(ref1)→ pass(log)
                    log:lex-sort(sym1,sym2,1)
                     sym1:get-symbol(ref1)
                     sym2:get-symbol(ref2)
  is-key(ref1)→ pass(s-index(ref1)<s-index(ref2))

type-of(ref)=
  is-logical(ref)→ pass(1)
  is-integer(ref)→ pass(2)
  is-symbol(ref)→ pass(3)
  is-key(ref)→ pass(4)

lex-sort(sym1,sym2,j)=
  j>length(sym1)→ pass(true)
  j>length(sym2)→ pass(false)
  elem(j)(sym1)<<elem(j)(sym2)→ pass(true)
  elem(j)(sym2)<<elem(j)(sym1)→ pass(false)
  true→ lex-sort(sym1,sym2,j+1)
```

C.10. Translate CSR's into Control Structures.

In this section the following shorthand notations are used:

C1=s-component^elem(1)

C2=s-component^elem(2)

C3=s-component^elem(3)

Cj=s-component^elem(j)

C.10.1

```
translate-cs(ref)=
  is-ref-loop(ref)→ pass(ref)
  is-ref-exit(ref)→ pass(ref)
  true→ translate-compound-cs(shf)
         shf:get-sheaf(ref)

translate-compound-cs(shf)=
  is-execution-rep(shf)→ translate-execution(shf)
  is-sequence-rep(shf)→ translate-sequence(shf)
  is-conditional-rep(shf)→ translate-conditional(shf)
  is-iteration-rep(shf)→ translate-iteration(shf)
  is-return-rep(shf)→ translate-return(shf)
  true→ error

translate-execution(shf)=
  true→ translate-execution-1(exp,syms,cs)
         exp:translate-expression(C3(shf))
         syms:translate-syms(row)
           row:get-sheaf(C2(shf))
             check-row(C2(shf))
         cs:translate-cs(C1(shf))

translate-execution-1(exp,row,cs)=
  length(row)=0→ pass(M(<s-execute:exp>,
                        <s-declare:<>>,
                        <s-body:cs>))
  true→ pass(M(<s-execute:exp>,
               <s-declare:res>,
               <s-body:cs>))
         {elem(j)(res):translate-identifier(Cj(row))
                                  | 1≤j≤length(row)}
```

```
translate-syms(row)=
  true→ pass(syms)
        {<elem(j)(syms):translate-identifier(Cj(row))>
                              | 1≤j≤length(row)}

translate-sequence(shf)=
  true→ translate-sequence-1(row)
        row:get-sheaf(C1(shf))
          check-row(C1(shf))

translate-sequence-1(row)=
  length(row)=0→ pass(M(<s-sequence:<>>))
  true→ pass(M(<s-sequence:cs>))
        {elem(j)(cs):translate-cs(Cj(row)
                              | 1≤j≤length(row)}

translate-conditional(shf)=
  true→ pass(M(<s-if:id>,<s-then:cs1>,<s-else:cs2>))
        id:translate-identifier(C2(shf))
        cs1:translate-cs(C3(shf))
        cs2:translate-cs(C1(shf))

translate-iteration(shf)=
  true→ pass(M(<s-repeat:cs>))
        cs:translate-cs(C1(shf))

translate-return(shf)=
  true→ translate-return-1(row)
        row:get-sheaf(C1(shf))
          check-row(C1(shf))

translate-return-1(row)=
  true→ pass(M(<s-return:res>))
        {elem(j)(res):translate-identifier(Cj(row))
                              | 1≤j≤length(row)}
```

C.10.2

```
translate-expression(ref)=
  true→ translate-expression-1(shf)
        shf:get-sheaf(ref)
```

```
translate-expression-1(shf)=
  is-anadic-rep(shf)→ translate-anadic(shf)
  is-monadic-rep(shf)→ translate-monadic(shf)
  is-diadic-rep(shf)→ translate-diadic(shf)
  is-triadic-rep(shf)→ translate-triadic(shf)
  is-compute-rep(shf)→ translate-compute(shf)
  true→ error

translate-anadic(shf)=
  true→ pass(M(<s-anadic:pos>))
         pos:translate-anadic-op(C1(shf))

translate-monadic(shf)=
  true→ pass(M(<s-monadic:pos>,<s-operand-1:op1>))
         pos:translate-monadic-op(C1(shf))
         op1:translate-identifier(C2(shf))

translate-diadic(shf)=
  true→ pass(M(<s-diadic:pos>,
                  <s-operand-1:op1>,<s-operand-2:op2>))
         pos:translate-diadic-op(C1(shf))
         op1:translate-identifier(C2(shf))
         op2:translate-identifier(C3(shf))

translate-triadic(shf)=
  true→ pass(M(<s-triadic:pos>,<s-operand-1:op1>,
                  <s-operand-2:op2>,<s-operand-3:op3>))
         pos:translate-triadic-op(C1(shf))
         op1:translate-identifier(C2(shf))
         op2:translate-identifier(C3(shf))
         op3:translate-identifier(C4(shf))

translate-compute(shf)=
  true→ pass(M(<s-declare:opsyms>,<s-body:cs>))
         opsyms:translate-opsyms(row)
           row:get-sheaf(C2(shf))
             check-row(C2(shf))
         cs:translate-cs(C1(shf))

translate-opsyms(row)=
  true→ pass(syms)
         {<elem(j)(syms):translate-opsym(Cj(row))>
                          | 1≤j≤length(row)}
```

```
translate-opsym(ref)=
  is-symbol(ref)→ pass(ref)
  is-sheaf(ref)→ translate-opsym-1(shf)
                  shf:get-sheaf(ref)
  true→ error

translate-opsym-1(shf)=
  length(shf)=0→ pass(<>)
  true→ error
```

C.10.3

```
translate-identifier(ref)=
  is-symbol(ref)→ pass(ref)
  true→ error
```

C.10.4

```
translate-anadic-op(ref)=
  is-anadic-op(ref)→ pass(ref)
  true→ error

translate-monadic-op(ref)=
  is-monadic-op(ref)→ pass(ref)
  true→ error

translate-diadic-op(ref)=
  is-diadic-op(ref)→ pass(ref)
  true→ error

translate-triadic-op(ref)=
  is-triadic-op(ref)→ pass(ref)
  true→ error
```

C.11. Reference Creators.

```
make-logical(log)=
  true→ make-logical-1(log,index)
      index:search(C(S),null,log)
    with C=s-logicals^s-items

make-logical-1(log,index)=
  index≠null→ pass(M(<s-type:rtplog>,<s-index:index>))
  true→ make-logical-2(log)

make-logical-2(log)=
  PASS: M(<s-type:rtplog>,<s-index:length(C(S))+1>)
  elem(length(C(S))+1)^C: log
    with C=s-logicals^s-items

make-integer(int)=
  true→ make-integer-1(int,index)
      index:search(C(S),null,int)
    with C=s-integers^s-items

make-integer-1(int,index)=
  index≠null→ pass(M(<s-type:rtpint>,<s-index:index>))
  true→ make-integer-2(int)

make-integer-2(int)=
  PASS: M(<s-type:rtpint>,<s-index:length(C(S))+1>)
  elem(length(C(S))+1)^C: int
    with C=s-integers^s-items

make-symbol(sym)=
  true→ make-symbol-1(sym,index)
      index:search(C(S),null,sym)
    with C=s-symbols^s-items

make-symbol-1(sym,index)=
  index≠null→ pass(M(<s-type:rtpsym>,<s-index:index>))
  true→ make-symbol-2(sym)

make-symbol-2(sym)=
  PASS: M(<s-type:rtpsym>,<s-index:length(C(S))+1>)
  elem(length(C(S))+1)^C: sym
    with C=s-symbols^s-items

make-key()=
  PASS: M(<s-type:rtpkey>,<s-index:length(C(S))+1>)
  elem(length(C(S))+1)^C: <>
    with C=s-keys^s-items
```

```
make-sheaf(shf)=
  PASS: M(<s-type:rtpshf>,<s-index:length(C(S))+1>)
  elem(length(C(S))+1)^C: shf
    with C=s-sheaves^s-items

make-cell(ref)=
  PASS: M(<s-type:rtpcel>,<s-index:length(C(S))+1>)
  elem(length(C(S))+1)^C: ref
    with C=s-cells^s-items

make-functional(fnl)=
  PASS: M(<s-type:rtpfnl>,<s-index:length(C(S))+1>)
  elem(length(C(S))+1)^C: fnl
    with C=s-functionals^s-items

make-bundle(bnd)=
  PASS: M(<s-type:rtpbnd>,<s-index:length(C(S))+1>)
  elem(length(C(S))+1)^C: bnd
    with C=s-bundles^s-items

[ make-semaphore(sem)=                                              ]
[   PASS: M(<s-type:rtpsem>,<s-index:length(C(S))+1>)               ]
[   elem(length(C(S))+1)^C: sem                                     ]
[     with C=s-semaphores^s-items                                   ]

make-token()=
  PASS: M(<s-type:rtptok>,<s-index:length(s-tokens(S))+1>)
  elem(length(s-tokens(S))+1)^s-tokens: <>
```

C.12. Reference to Item.

```
get-logical(ref)=
  is-logical(ref)→ pass(elem(s-index(ref))^B(S))
  true→ error
    with B=s-logicals^s-items

get-integer(ref)=
  is-integer(ref)→ pass(elem(s-index(ref))^B(S))
  true→ error
    with B=s-integers^s-items

get-symbol(ref)=
  is-symbol(ref)→ pass(elem(s-index(ref))^B(S))
  true→ error
    with B=s-symbols^s-items
```

```
  get-sheaf(ref)=
    is-sheaf(ref)→ pass(elem(s-index(ref))^B(S))
    true→ error
      with B=s-sheaves^s-items

  get-cell-index(ref)=
    is-cell(ref)→ pass(s-index(ref))
    true→ error

  get-functional(ref)=
    is-functional(ref)→ pass(elem(s-index(ref))^B(S))
    true→ error
      with B=s-functionals^s-items

  get-bundle(ref)=
    is-bundle(ref)→ pass(elem(s-index(ref))B(S))
    true→ error
      with B=s-bundles^s-items

[ get-semaphore-index(ref)=                                    ]
[   is-semaphore(ref)→ pass(s-index(ref))                      ]
[   true→ error                                                ]
```

C.13. Checking the Form of Objects.

```
  check-symbol(ref)=
    is-symbol(ref)→ null
    true→ error

  check-selector(ref)=
    is-sel-ref(ref)→ null
    true→ error

  check-row(ref)=
    true→ check-row-1(shf)
           shf:get-sheaf(ref)

  check-row-1(shf)=
    true→ {check-row-elem(j,shf) | 1≤j≤length(shf)}

  check-row-elem(j,shf)=
    not(is-integer(s-selector^elem(j)(shf)))→ error
    s-index^s-selector^elem(j)(shf)≠j→ error
    true→ null
```

C.14. Mutual-exclusion Semaphore.

```
[ lock(sel)=                                                    ]
[   true→ lock-1(sel,log)                                       ]
[            log:attempt-to-lock(sel)                           ]

[ lock-1(sel,log)=                                              ]
[   log→ null                                                   ]
[   true→ lock(sel)                                             ]

[ attempt-to-lock(sel)=                                         ]
[   PASS: sel(S)                                                ]
[   sel: false                                                  ]

[ unlock(sel)=                                                  ]
[   sel: true                                                   ]
```

C.15. Search.

```
search(list,selector,object)=
  true→ pass(k)
         {k:search-elem(list,selector,object,j) | 1≤j≤length(list)}

search-elem(list,selector,object,j)=
  selector^elem(j)(list)=object→ pass(j)
  true→ pass(null)
```

C.16. Build Result Structures for Return.

```
return-0-refs()=
  PASS: M(<s-signal:sigreturn,s-results:<>)

return-1-ref(ref)=
  PASS: M(<s-signal:sigreturn,s-results:M(<elem(1):ref>)>)

return-2-refs(ref1,ref2)=
  PASS: M(<s-signal:sigreturn,
          s-results:M(<elem(1):ref1>,<elem(2):ref2>)>)
```

C.17. Pass.

```
pass(v)=
  PASS: v
```

Glossary.

The words listed here are used as technical terms in this report. For each, a brief definition and the section(s) where it is introduced are given in brackets [ ].

activation — 1. (general) the activity resulting from invoking a module [1.1]. 2. (specific) the information needed to evaluate a functional [2.1].

activity — an interval encompassing the evaluation of a PO [5.3], a control structure [5.3], or a functional [5.3].

activity specification — a PO, control structure, or functional[5.3].

add — a PO; implements integer addition [2.5].

admissible set — those functionals which will be permitted in the inputs to functionals for (conceptual) testing of the special behaviours [5.3].

anadic PO — PO taking no operands [2.4].

and — a PO; implements the logical function "and" [2.5].

augment — a PO; creates a new sheaf having one more component than an existing sheaf [2.7].

behaviour — all the information which are needed to characterize the service a module provides [1.1].

bind — 1. (general) to arrange that a context map a name to a module [1.3]. 2. (specific) to use the PO bind to create a new functional [2.9]. 3. a PO; creates a new functional by binding an identifier in an existing functional [2.9].

binding — the relationship between an identifier and (a reference to) an item in an environment [2.2].

Binding Model — an abstract machine which is intended as the kernel of a class of modular programming systems [1.6].

body — 1. part of an execute/declare control directive; the "scope" of the declaration [2.11]. 2. part of a compute expression; the scope of declarations [2.13].

bundle — a non-elementary, usually structured, constant item used to provide grouping with controlled access to components in the model [2.2].

cell — a non-elementary, structured, mutable item used to provide mutability in the model [2.2].

cells(x) — the cells in the closure of x [5.2].

cell-free — 1. Enclosing no cells [5.5]. 2. (proxy) all conditions enclose no cells [5.7].

closure — 1. (of item) the items and tokens that item encloses [5.2]. 2. (of package) the union of the closures of the item constituents of the package.

combined package — a package having as constituents all the contituents of an input package for a functional plus the functional itself [5.4].

compatible — Describing a set of modules which can be employable by another module without conflict. Describing [1.1].

compiled out — the removed from the representations of control structures of the strings of characters defining identifiers.

component — One of the collection of items which, by interpretation, are grouped by a sheaf or bundle [2.2].

compose — a PO; implements concatenation on the characters of a row of symbols [2.5].

compute — an expression used for intra-activation returns and generation of cyclic patterns of reference [2.13].

condition — see proxy [5.7].

conditional — a control directive; used to choose the evaluation of one of two control structures based on a logical item [2.11].

conflict of names — Multiple conflicting attempts to bind a name in a context [1.3].

constant — see mutable [1.1].

constituent — named member of a package [5.2].

contents — 1. the item directly enclosed by a cell [2.2]. 2. a PO; accesses the contents of a cell [2.8].

context — a partial function mapping the names of a namespace into modules [1.3].

control directive — a "machine instruction" of the moduel; a node in the tree of directives forming a control structure [2.1, 2.11].

control point — Part of an activation; the indication of a control directive in the control structure of an activation [2.1].

control structure — a "machine language program" of the model; a tree of control directives; part of functionals [2.2] and associated with activations through control points [2.1].

control structure representation — a sheaf whose closure represents a control structure [2.9].

co-ordinated version — Modified version of a functional which uses a mutual exclusion semophore to insure that at most one process is evaluating it at a time.

correspond — 1. two references in two items correspond when they fill the same role in the structure of those items [5.2]. 2. two constituents in two packages correspond when they are images of the same tag [5.2].

cover — a similarity covers every item of its domains [5.2].

creatively extend — two activity specifications which extend a similarity on two input packages so that the items add to the domains of the similarity to arrive at its extension are all created by the evaluation of those activity specifications on those packages [5.3].

creator — the human being or module that causes another module to come into being [1.1].

CSR — see control structure representation [2.9].

declare — to add a binding for an identifier to the environment of the current activation [2.1].

decompose — a PO: makes a row of single-character symbols from the characters of one symbol [2.5].

derived package — Package whose tags are the identifiers of an environment, and whose constituents are the items or tokens to which those identifiers are bound [5.2].

determine — to replace all references to a token with references to some item [2.13].

determination — the item used to determine a token [2.13].

diadic PO — Po taking two operands [2.4].

directly enclose — One item directly encloses another when there is an arc emanating from the first and terminating on the second [2.2, 4.12].

disconnected set — a collection of items all references to members of which are included in those members [7.6].

discriminatory — see non-discriminatory [1.1].

divide — a PO; implements integer division [2.5].

elementary item — an item whose information content determines its identity: a logical, integer or string [2.2].

employ — One module employs another if the first is built using the other. the employed module serves the employer.

enclose — One item encloses another if there is a path with zero or more arcs in it leading from the first to the second [2.2. 4.12].

environment — a context mapping identifiers into references; occur as parts of activations [2.1] and of functionals [2.2].

equal — a PO; implements "identical" on the selectors [2.6].

error — the distinguished final state indicating that an illegal action has been requested of the model [2.4].

establish — a relation mapping one item [5.2] or package [5.2] to another as required to show that they are similar establishes that they are similar.

eval — a PO; invokes the evaluation of a functional [2.9].

evaluate — the act of the (a) process in carrying out the activities specified by either a control structure [2.1] or a control directive.

execute — the act of the (a) process in carrying out the activiy specified by an expression [2.1], in particular by a PO [2.4].

execute/declare — a control dirctive; used to invoke the execution of an expression and the declaration of local identifiers [2.11].

exit — a control structure; used to force the termination of an iteration control structure [2.11].

extend — two activity specifications extend a similarity on two (input) packages wehn there is similarity which is a relational extension and covers the output packages [5.3].

the extension — the intersection of all extensions of a similarity on a pair of packages [5.3].

extract — 1. to access a component of a bundle [2.2]. 2. a PO; accesses a component of a bundle [2.7].

extractor — an elementary item or key used to name a component of a bundle [2.2].

final state — an ending state for the model; one in which. by interpretation, all evaluation has been completed [2.1].

flexible — the criterion for modular programming systems demanding that any module be employable by any other without constraining the use of names in the employer [1.1].

fork — a PO; used to create a new process [6.1].

graph of items — the collection of all the items of the model, visualized as forming a directed graph [2.1].

handle — an item, or environment of an activation which includes a reference to an item in the closure of a second item has handle on that second item in the first item [5.5].

identical — 1. the equality test for items (nodes in the graph) [2.2, 4.12]. 2. a PO; test the identity of two items [2.4].

include — an item which has a reference as part of its information content is said to include that reference [4.12].

information content — the information defining the activity of an item in the model [2.2].

initial condition — the starting condition for a proxy [5.7].

initial state — a starting state for the model; one in which, by interpretating, no evaluation has taken place [2.1].

input package — the package regarded as input to a PO [5.3], control structure [5.3], or a functional [5.3].

install — a PO; creates a new unbound function [2.9].

integer, integer item — an elemenary (unstructured, constant) item used to provide arithmetic in the model [2.2].

interval — a set of adjacent states in the model [5.3].

invoke — to request service from an imployed module [1.1].

invulnerable functional — a functional which encloses no items at which it is vulnerable [5.6].

is-extractor — a PO; tests whether an item is an extractor of a particular bundle [2.7].

is-bundle — a PO; used to test whether an item is a bundle or not [2.4].

is-cell — a PO; used to test whether an item is a cell or not [2.4].

is-functional — a PO; used to test whether an item is a functional or not [2.4].

is-integer — a PO; used to test whether an item is a integer or not [2.4].

is-key — a PO; used to test whether an item is a key or not [2.4].

is-logical — a PO; used to test whether an item is a logical or not [2.4].

is-semaphore — a PO; used to test whether an item is a semaphore or not [2.4].

is-sheaf — a PO; used to test whether an item is a sheaf or not [2.4].

is-symbol — a PO; used to test whether an item is a symbol or not [2.4].

is-selector — a PO; tests whether an item is a selector of a particular sheaf [2.7].

item — a passive (data) entity in the model; one node of the graph of items [2.1].

iteration — a control structure; used to repeatedly evaluate one control structure [2.11].

key — a non-elementary, unstructure, constant item used to provide non-counterfeitable identity in the model [2.2].

length — a PO; yields the number of components in a sheaf [2.7].

less-than — a PO; implements a total ordering on the selectors [2.6].

limited functional — a functional whose control structure contains no invocations of the PO's identical new-key, and install [5.4]. in the multi-process version of the model, leave out fork as well [6.5].

logical, logical item — an elementary (unstructured, constant) item used to provide logical computation in the model [2.2].

loop — a control structure; used to force a new iteration of an iteration control structure [2.11].

match — 1. two items or tokens match if they have the same information content not counting most references [5.2]. 2. two packages match if they are defined for the same set of tags [5.2].

minimal similarity — the smallest similarity establishing that two items [5.2] or packages [5.2] are similar.

module — a combination of programs and data which can be invoked to provide services and employed in the construction of other modules [1.1].

monadic PO — PO taking one operand [2.4].

multiply — a PO; implements integer multiplication [2.5].

mutable — 1. (general) Describing a module whose behaviour includes input/output characteristics which vary with time; the opposite of constant [1.1]. 2. (specific) an item whose information content can be changed [2.2, 2.8, 4.12].

mutate — to change the information content of a cell or semaphore [6.5].

name — an element of a namespace, used in conjunction with contexts to identify modules [1.3].

namespace — see name [1.3].

new bundle — a PO; generates a new bundle from a sheaf of the same structure [2.7].

new-cell — a PO; generates a new cell [2.8].

new-key — a PO; creates a new key [2.6].

new item — an item added by the action of a PO to the graph of items [2.4].

new-semaphore — a PO, used to create a new semaphore.

non-discriminatory — a module which presents the same behaviour to all employeing modules [1.1].

non-elementary item — see elementary item [2.2].

not — a PO; implements the Logical funtion "not" [2.5].

operand — a refernce used to indicate which items in the graph a Po is to operate upon [2.4].

or — a PO; implements the logical function "or" [2.5].

output package — the package regarded as output from a PO [5.3]. control structure [5.3] or a functional [5.3].

p — a PO; used to decrement a semaphore [6.1].

package — a collection of named items and tokens; used for capturing the idea of "all the inputs (outputs) of an activity specification" [5.2].

placeholder
part of the declaration list of a compute expression used to indicate that no token is to be created [2.13].

PO — see primitive operation [2.4].

potential reference — a reference to a token [2.13].

primitive operations — the fundamental operations of the model; these define the primitive types [2.4].

primitive type — the type of an item [2.4].

procedural encapsulation — the scheme of representing an instance of a user-defined type by the collection of its operations [3.8].

process — an active entity in the model which evaluates functionals of the model [2.1].

program — Functional represented in a programing language [3.1].

programmer — human being who creates modules in a programming system [1.1].

programming language — a scheme by which functionals can be respresented by strings of characters [3.1].

proper functional — a functional all of whose enclose functionals are limited [5.4].

proxy — a repeatable functional which has the same behaviour as a sequence-repeatable functional provided it is handed a set of conditions (characterizing the state of the sequence-repeatable functional) [5.7].

refer to — a reference bound to an item in the reference context refers to that item [2.3].

reference — a name used with the reference context to locate an item [2.3].

reference context — a single context, managed by the model in which references are bound to items [2.2].

rep — the collection of items shared by the operations representing an instance of a user-defined type [3.8].

repeatable — 1. (general) property of a module, that it be constant [1.2]. 2. (specific) property of a functional, that "the similar inputs yield similar outputs" [5.1, 5.5].

resolve — to apply a context to a name [1.3].

result structure — the result of the evaluation of a control structure; indicates why the evaluation was terminated [2.11].

return — a control directive; used to signal the completion of an activation or the creation of a compute expression [2.11].

same — a PO; implements the identity test on cells [2.8].

select — 1. to access a component of a sheaf [2.2]. 2. a PO; accesses a component of a sheaf [2.7].

selector — 1. an elementary or key used to name a component of a sheaf [2.2]. 2. a PO; used for enumerating the selectors of a sheaf [2.7].

the selectors — the set of all elementary items and keys [2.2].

self-referential item — one enclosed in some item which it directly encloses [2.13].

self-sufficient — Describing a module which supplies itself with all modules needed to support its behaviour [1.1].

separate — an interval separates two applications of a functional if it starts when the first stops, stops when the second begins, and includes no other application of the functional [5.6].

sequence — a control directive; used to invoke the evaluation of a sequence of control structures [2.11].

sequence-repeatable — 1. (general) property of a module; that it have the kind of mutability which make it repeatable when viewed as operating on sequences of inputs to produce sequences of outputs [1.2]. 2. (specific) property of a functional; that "similar sequences of inputs yield similar sequences of outputs" [5.1, 5.6].

serve — see employ [1.1].

sharing pattern
: the pattern in the closure of an item of reference to cells [5.1].

sheaf
: a non-elementary, usually structured, constant item used to provide grouping in the model [2.2].

side-effects
: the updating by the evaluation of a functional of the contents of cells enclosed in its input [5.1].

similar
: 1. two items are similar if they "look alike" and have the same sharing patterns and identical keys [5.2]. 2. two packages are similar if corresponding constituents are similar under a single similarity [5.2.]. 3. two environments are similar if their derived packages are similar.

similarly vulnerable
: two functionals are similarly-vulnerable if they are similar and they are vulnerable at corresponding items in their closures [5.5].

similarity
: a relation establishing for every item in its domains its similarity to some other such item [5.2].

simple CS
: one containing no invocations of the PO's identical, new-key, and eval.

spawned
: a process whose creation resulted the evaluation of a functional, or from the activity of another spawned process [6.5].

specially-behaved functional
: a repeatable or sequence-repeatable functional.

state
: a snapshot of the complete status of the model at some point in time [2.1].

state transition
: the passage from one state to (one of) its successor states [2.1, 6.1].

state transition rule
: a specification defining for each state its (collection of) successor state(s) [2.1, 6.1].

structured items
: an item with one or more arcs emanating from it [2.2].

subsimilarity
: a similarity which is only part of a larger one [5.2].

subtract — a PO: implements integer subtraction [2.5].

successor state — the (a) state after this one; defined for each state by the state transition rule [2.1].

symbol — an elementary (unstructured, constant) item used to provide naming and text manipulation in the model [2.2].

tag — a namespace used to name items or tokens in packages [5.2].

task — the functional from which the environment and control structure of the (a) process are taken for the first state in which the process exists (for the single process version, for the initial state) [2.1].

text — string used to represent a program in a programming language [3.1].

token — an incomplete node of the graph of items used to stand for some item; used for generating cyclic patterns of reference [2.13].

tokens (x) — the tokens in the closure of x [5.2].

triadic PO — PO taking three operands [2.4].

type — a characterization of a class of entities which defines the operations that manipulate those entities [2.4].

type-list — a component of the VDL states used to hold the information content of all the items in the state which are of that particular type [4.2].

unrelated process — a process which is not spawned by the evaluation of a functional, is not the process carrying out that evaluation [6.5].

unreliable name — Use of a name to employ a module when the context for resolving it may not reliably be obtained later where the binding in that context may change [1.3].

unstructured item — see structured item [2.2].

update — a PO; changes the contents of a cell [2.8].

user — human being who invokes the modules of a programming system [1.1].

v — a PO; used to increment a semaphore [6.1].

variable — an identifier (usually locally declared) bound to a cell [2.11].

VDL state — VDL objects which are of the form acceptable as states of a VDL definition [4.1].

vulnerable — a functional is vulnerable at an enlose item if there exists a handle on that item in functional [5.5].

well-behaved set of functionals — a set whose members, when applied to inputs all of whose enclosed functionals are also members of the set, yield outputs all of whose enclosed functionals are also in the set [5.4].

yield — a PO's act of producing references as results [2.4].

Biographical Note.

D. Austin Henderson, Jr. was born in London, Ontario, Canada on January 25, 1943. He was raised in Toronto, and attended Brown Public School and Upper Canada College (1956-61). While at Upper Canada he was active in sports, a lieutenant in the Cadet Corps and head prefect of Jackson's House. An active interest in Mathematics led to participation in, and winning of, the Ontario Regional Science Fair (1960). He spent summers camping and canoeing.

From 1961 to 1965, he attended Queen's University at Kingston, Ontario, and pursuing an honours degree in Mathematics and Physics on a Williamson Memorial Scholarship for General Proficiency. While there, he was active in student government, music, and intramural sports. Summer vacations were spent travelling in Europe (1962), and doing applications programming for Canada Life Assurance Co. of Canada (Toronto: 1963), and International Business Machines and Avon Products of Canada (Montreal: 1964, 1965).

From September 1965 to June 1966, Austin and a friend travelled around the world, covering 52,000 miles and visiting 22 countries. This trip included two months of skiing in the Alps.

Austin spent the academic year 1966-67 at the University of Illinois in Champaign-Urbana, Illinois. He studied on a Fellowship in Computer Science, and obtained his Masters of Science in August, 1967. during the summer of 1967, he also was employed by the High Energy Physics Division of Argonne National Laboratory doing applications programming for the Polly Project (bubble chamber film analysis).

Since September 1967, Austin has been a graduate student at MIT, and a research assistant at Project MAC. He has been an instructor in a course in Computational Linguistics (6.253, Spring Term, 1967-68), and has helped in the development of material for a course in Programming Linguistics

(6.231, 1968-1971). He has worked in the Programming Linguistics Group (no longer active), and in the Computations Structures Group. He was instrumental in bringing about, and carrying out, a review of Graduate Education in Computer Science in the Electrical Engineering Department at MIT which ultimately resulted in high level organizational changes within the department. He has also undertaken summer employment (1968, 1969) and consulting (to present) with MIT's Lincoln Laboratory, working primarily on computer graphics at the TX-2 computer facility. He has also been employed part-time with Bolt Beranek and Newman Inc. doing computer simulation of vehicular traffic, and early experiments on software designed to run on computer networks.

His publications include his Master's Thesis ([Henderson, 67]), reports on the graphical language AMBIT/G ([Henderson, 69] and [Rovner, 69]), and reports on single- and multi-computer versions of a simulation system ([Sutherland, 71] and [Thomas, 72]).

His active interests now include playing the piano, and sports: tennis, squash, skiing, and kayaking.