Toward Relaxing Assumptions in Languages and Their Implementations

Mary Shaw and Wm. A. Wulf

Computer Science Department
Carnegie Mellon University
Pittsburgh, Pa.

2 January 1980

Abstract

Language implementors frequently make pre-emptive decisions concerning the exact implementations of language features. These decisions constrain programmers' control over their computations and may tempt them to write involuted code to obtain special (or efficient) effects. In many cases, we can distinguish some properties of a language facility that are essential to the semantics and other properties that are incidental. Recent abstraction techniques emphasize dealing with such distinctions by separating the properties that are necessary to preserve the semantics from the details for which some decision must be made but many choices are adequate. We suggest here that these abstraction techniques can be applied to the problem of pre-emptive language decisions by specifying the essential properties of languages facilities in a skeleton base language and defining interfaces that will accept a variety of implementations that differ in other details.

This research was sponsored by the National Science Foundation under Grant MCS77-03883 and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

1. Introduction

Traditionally, the designers and implementors of programming languages have made a number of decisions about the nature and representation of various language features that the authors feel are unnecessarily pre-emptive. For example, such decisions are commonly made about arrays: most languages support only rectangular arrays, and each implementation generally uses some particular representation, such as row-major order, for all arrays. Neither of these choices is logically necessary: a language could, for example, permit triangular arrays or arrays in which each row has a different length, and there are many possible representations even for simple rectangular arrays.

In many, even most, situations, the kinds of language and implementation decisions to which we refer are beneficial. The programmer usually doesn't care what representation is chosen for arrays, for example, and the default decisions have been refined through long experience to yield representations that are broadly acceptable. Unfortunately, precisely the same decisions occasionally have a detrimental effect on both program clarity (structure) and efficiency. The authors have seen numerous examples of FORTRAN programs that store two triangular matrices (one of them transposed) in the same FORTRAN array. The resulting program is generally extremely difficult to understand since a term like *A(I,J)* may refer to an element of either matrix. In addition, it is error-prone because the correspondence of subscripts to rows and columns is not consistent. Such programs may also be slower than one might like since most implementations are tuned to varying the same subscript position most rapidly for all arrays; since one matrix is transposed in the array, access to one or the other is necessarily non-optimal.

Although one may question the programmer's wisdom in packing two matrices together in this way, the fault actually lies more with FORTRAN and its implementations than with the programmer. The programmer *needed* a compactly implemented abstraction (triangular arrays), but the language/implementation combination neither provided it nor provided a way to define it. In at least some of these cases, the space that would have been wasted by using two full rectangular arrays would have prevented the program from running at all! Thus, the programmer really had little choice.

The purpose of this note is to advocate a somewhat different philosophy of language design. We will explore a collection of pre-emptive decisions, and we will observe that the abstraction facilities of modern languages such as Alphard [Wulf 76], CLU [Liskov 77], Euclid [Lampson 77], Gypsy [Ambler 77], Ada [Ichbiah 79], and so on provide an abstraction facility adequate to express the kinds of decisions that have traditionally been pre-empted. These abstraction facilities, coupled with a philosophy that the decisions should not be pre-empted by the language design, can substantially enhance the extent to which languages permit us to express well-structured and efficient programs.

In many cases, we can readily distinguish some properties of a language feature that are essential to its semantics and other properties that are incidental. An array implementation, for example, must establish a one-to-one correspondence between subscript values and memory locations. However, neither the order in which the locations are laid out in memory nor the algorithm used to achieve the mapping is essential to achieving the desired effect. Since abstraction facilities are concerned with precisely the distinction between specification (semantics) and implementation, we shall advocate a language design philosophy in which *only* the essential semantics of language facilities are defined for the base language, and a data abstraction mechanism permits the programmer to provide a variety of implementations that differ only in semantically inessential ways[1]. The semantics of the language therefore become "relative" to the semantics of the programmer-supplied components of the implementation, and the correctness of the whole will depend on the correctness of the programmer-supplied components. Thus, the "essential semantics" of the language includes a collection of properties that must be proven for the programmer-supplied implementations. The proofs of these properties are no different from those for programmer-supplied definitions that extend the language, so the safety of the resulting system is in no way compromised.

It is perhaps worth noting the difference between the philosophy that we shall espouse and one that we believe to be prevalant in the data abstraction language community. In large measure, the popular image of data abstraction mechanisms is that they provide a weak, but important, form of language extension. That is, they provide the means by which one extends a language "upward" to include new data types not present in the base language (which is generally chosen to be roughly the level of Pascal). The authors wholeheartedly subscribe to the idea of upward extension; it is an important ingredient of modern notions of software engineering. However, we also believe that control of decisions *below* the level of a Pascal-like language are also important -- and that is the issue we wish to explore in this note.

[1]Note that default definitions should still be provided for the programmer who chooses not to be concerned with these details.

2. Traditionally Pre-empted Decisions

In this section we shall simply list a set of decisions that have traditionally been pre-empted -- unnecessarily, we believe. This list is not intended to be exhaustive; the reader may well be able to add other decisions. Rather, it is intended as a basis for discussing some ways in which data abstraction and language changes could help to bring some of the decisions under the programmer's control. We shall also give some brief examples of facilities that have *not* pre-empted these decisions; the intent of these comments is likewise not to be exhaustive, but rather to illustrate that pre-emption is not a necessary property of language design. Finally, note that not all of these issues arise in all languages. In particular, some appear only in languages that provide a way for a programmer to define new data types. We include those issues here, however, because an important class of new languages is involved and because they illustrate some of the interactions among features that may occur in the process of language design.

- *Storage layout*: Several decisions are actually involved here, including the treatments of scalar representations, array representations, record element representations, and packing. There are several alternative choices for each of these, and it is not *a priori* clear which is best; indeed it *is* clear that no single one is best in all cases. Various degrees of control over these decisions have been provided in languages. Many older languages (e.g., FORTRAN and ALGOL) provided no control at all. Some languages have provided limited control in the form of specification that a record is to be "packed" -- but not a detailed specification of the packing itself. Other languages, such as Ada, have permitted detailed control over packing strategy, size of variables, internal values for elements of an enumeration, and so on; even Ada, however, does not allow (re)definition of array representations. In contrast, Bliss substantially departs from the "no control" approach -- in Bliss the programmer *must* provide a macro-like definition of the accessing algorithm for every new data structure, and thus *must* specify every detail of the representation (Section 4.2 elaborates on this).

- *Declarations, initialization, finalization*: In most programming languages, the declaration of a variable may cause several things to happen: allocation of space for the variable, binding of the name to the address (or offset) of this space, and initialization of the value of the variable. In addition, it may imply some "finalization" actions when control leaves the scope in which the variable is declared, most notably deallocation of the space. Unfortunately, these actions are usually only defined for the base types of the language and are simply extended on a default basis for programmer-defined types. To support some aspects of modern programming methodology, however, it is necessary for the programmer to control these actions. For example, type-specific initialization actions may be necessary in order to establish an invariant property of the type (that is, to assure a valid initial value). Alphard provides full initialization and finalization facilities; Ada provides limited initialization facilities that can be used to achieve full initialization with minor circumlocution.

- *Built-in operators and the semantics of assignment and equality*: The readability of a program is substantially enhanced when infix notation can be used for operators, particularly when the the newly defined types are familiar mathematical ones (e.g., complex). Many languages [Schuman 71] have provided ways to extend, or overload, the built-in operators; both Algol 68 and Ada provide this facility, for example. Unfortunately, these languages have provided no way to control the additional properties that are usually, but not always, assumed for the built-in operators (for example, is an overloading of '+' commutative and associative?). Even in languages that permit overloading, overloading of assignment and equality are often subject to special restrictions or prohibited entirely; we suspect that this is because the "normal" semantics of these operations are so important to a program. However, type-specific definitions of these operations can be made safely and are sometimes essential to preserve the semantics of a new type.

- *Dynamic storage allocation*: Storage allocators typically incorporate policies concerning search strategy, garbage collection, collapsing adjacent free cells to limit fragmentation, and so on. Problem-specific characteristics strongly influence the best decisions about these policies. For example, in some cases prior knowledge about request sizes or order of allocation and deallocation may make extremely efficient allocation possible. Euclid provides *zones* to allow programmers to define specific allocation algorithms; this mechanism is discussed in Section 4.3. Ada permits rudimentary control over dynamic storage allocation via a mechanism for determining the size of the storage pool for each type of dynamically-allocated variable, but the algorithm for managing this storage pool is fixed for each implementation.

- *Loop control*: When a program iterates systematically over a data structure, the designer of that data structure is in a much better position than the language designer to know the most appropriate or efficient order for processing the elements of the structure. In addition, different traversal patterns may be preferred in various situations. Alphard and CLU provide means for the designer of a data structure to provide algorithms for supplying elements to loops. The Alphard scheme is discussed in Section 4.1.

- *Scheduling and synchronization*: In specific systems, the programmer may have the need or desire to express the relative priorities or deadlines of separate tasks. Alternatively, the programmer may have knowledge of code or data sharing that makes co-scheduling of certain tasks vastly more efficient than independent scheduling. Similarly, certain synchronization and communication schemes may be both more natural and more efficient for certain problem decompositions. Parallelism, of course, has not been common in languages other than those for simulation and real-time applications. Languages such as Ada, Concurrent Pascal [Brinch Hansen 75] and Modula [Wirth 77] have followed the traditional approach and provided single facilities and implementations -- the maximum variability being the ability to define the relative priority of processes.

There are, of course, many other candidates for non-preemptive decisions; among them are type-specific input/output (including the mapping to and from literals of a user-defined type), the details of procedure invocation and parameter binding, and the processing of exceptions (especially the policy for locating a handler).

3. Some Consequences of Pre-emptive Decisions

In the introduction we alluded to the negative consequences of pre-emptive decisions; in this section we will amplify on those remarks. In reading this, the reader should remember that often the default decisions made by language designers and implementors are perfectly adequate. We are *not* recommending that *all* low level decisions should be made by *all* programmers for *every* program they write. Rather, we are recommending that, in those cases where the default decision may be inappropriate, it should be possible for the programmer to override it in a safe and structured manner. In practice, we expect that the decisions to change implementations selected by the language designers will usually be made as part of the tuning process that goes on in the final stages of a project; in addition, we expect that the modifications will generally be made by specialists in such matters, not by all programmers as a matter of course.

The fatal flaw of pre-emptive language decisions arises from their conflict with one of the most fundamental precepts of structured programming, originally enunciated by Parnas [Parnas 72]: the *order* in which design decisions are made is crucial. One must *first* make those global decisions that are least likely to be changed; one should *postpone* those decisions that are most likely to be changed. Decisions cannot be postponed forever, of course, but one should wait until the maximal information is available. This is the essence of the "top-down" design methodology. One first makes (only) high-level organizational decisions. Only through refinement does one work down to the lowest level.

In a purely top-down design, the last decisions to be made are usually the lowest level representational choices. The decisions we have termed "pre-emptive" are also, generally speaking, relatively low level; indeed, the conventional rationale for pre-empting them is that they are so low-level that "the programmer shouldn't need to worry about them". Unfortunately, while there is a good deal of truth in this, it is precisely the point on which the traditional approach to language design runs afoul of a top-down approach to program design. Making representational choices at language design or implementation time is about as early as possible -- *not* as late as possible. Consequently, they are necessarily made with only a vague image of their typical use -- not a detailed knowledge of their use in a specific program.

We can see many consequences of the general argument above:

- *Introducing circumlocutions*: Situations such as the packing of two triangular matrices in one array, mentioned in the introduction, are circumlocutions forced upon the programmer. The basic algorithm of a program, although inherently simple, is obscured by the need to "program around" a limitation of either a language or its implementation. Had the programmer really been able to follow a top-down design strategy, that is follow it further "down", the program would have been much clearer.

- *Preventing feasible optimizations*: A good optimizing compiler can substantially improve the efficiency of a program. However, the most important improvements in a program's efficiency derive from good data structure and algorithm choices. If a language does not provide the appropriate structure the programmer will be forced either to use a less efficient algorithm or to encode the structure explicitly in terms of the structures that are available. Unfortunately, the latter alternative carries its own set of problems. A compiler, especially an optimizing one, must. preserve the semantics of the language constructs; moreover, compilers cannot deduce a programmer's *intent*. Thus, directly and explicitly encoding one structure in terms of another usually results in much less effective optimization than would have been possible if the original structure had been defined in a straight-forward manner. This is particularly true if complicated access algorithms must also be explicitly encoded. Again, if the programmer had really been allowed to follow a top-down design, the efficient representation would have been used -- leading to an intrinsically better program as well as one that the optimizer can manage better.

- *Discouraging the use of high-level languages*: Although most people now agree that the use of high-level languages is desirable, the fact remains that many major systems are still written in assembly language. There are, of course, many reasons for this -- only some of which can be addressed by the present proposal. However, in many cases the reasons for the choice of assembly language are related to the absolute need for both greater efficiency and more control over low-level decisions than is provided by most contemporary languages.

In addition to the methodological and pragmatic arguments above, there is an analogy with the theoreticians' experience with specifications: if a specification contains more detail than is absolutely necessary, it may constrain the implementation in such a way as to eliminate reasonable alternatives. Moreover, a specification is a guarantee; all implementations of the specification are obligated to follow *all* of it. Thus, if the specification contains too much detail -- and someone comes to depend on that detail -- all future implementations will be obligated to provide that detail[2]. Both of these effects are precisely what we observe when language designs, or their implementations, bind decisions too early.

[2]Recall, for example, the trauma induced by the decision to change the original definition of FORTRAN arrays as being "backwards" in memory.

4. A Proposed Approach and Examples

Faced with the arguments above, one might follow any of several paths. One possibility is to avoid high-level languages altogether; indeed, Parnas has advocated the use of a powerful macro processor instead of a high-level language [Parnas 74]. Such a processor would, presumably, leave all representational decisions under programmer control. However, it would provide no guidance about good organization or style, and it would support idiosyncratic notation rather than standard syntax with uniform interpretations. This extreme seems neither necessary nor desirable. Let's consider an alternative.

As we noted in the introduction, research on abstraction facilities has focused on a particular form of language extension. It has been concerned primarily with facilities that permit the programmer to define new, application-specific data types in terms of a predefined set supplied by the language. Among the facilities that are now commonly provided by data abstraction languages are:

- *Separation of specification and implementation*: While not strictly necessary from a logical standpoint, this separation aids the human reader/writer and, in particular, helps to hide the implementation [Parnas 71] and define a module boundary. When the specification is used as the sole source of information about a module, maintainability is enhanced because assumptions shared between the user and the implementor are explicit.

- *Encapsulation*: Encapsulation permits the definer of an abstraction to more tightly control the properties (notably representations and operations) that are visible to the user. Encapsulation facilities of a language can enforce the policy of separating specification and implementation information.

- *Overloading*: Overloading of operation names permits the definer of an abstraction to mask the distinction between those abstractions that are primitive to the language and those that are programmer-defined. This substantially enhances readability.

- *Generic definitions*: Again, while not strictly necessary, generic definitions permit the abstraction definer to cover a broader class of abstractions with a single definition. Indeed, the presence of the generic facility further focuses ones attention on the essential properties of a definition without constraining irrelevant, but visible detail.

Now let us consider using the same basic approach and the same basic abstraction features for the design of a language. The design will consist of several components:

(1) A syntactic definition. The base syntax is not of itself particularly crucial; presumably it will be similar to the Pascal derivatives in the data abstraction milieu.

(2) A semantic definition. Unlike the semantic definitions of most contemporary languages, the semantics for this language will be "incomplete" in that it will specify essential properties, but not all details, for some constructs. For any application, the semantics of those constructs will be fleshed out by implementations that preserve the essential properties.

(3) A list of specifications of "essential properties" for the constructs left incomplete in (2). These specifications are not merely "suggestions to the implementer"; they place formal constraints on the possible implementations. Any abstract definition whose specification assures *at least* these properties of a construct will provide an acceptable implementation of that construct. Some of these may be simple (such as a specification for "boolean"); others may be generic (such as that for a "generator" -- see [Shaw 77] and below).

(4) A useful implementation for each of the abstractions listed above. These will be the default implementations; they correspond to the pre-empted decisions in traditional languages.

Let's consider a simple example of this sort of design approach. The RED candidate for Ada [Nestor 79] defined a data type *data_lock* (essentially a mutex semaphore) and a region statement. Informally, the operations on *data_locks* were *Lock* and *UnLock* with semantics similar to Dijkstra's *P* and *V* on boolean (mutex) semaphores. The form and semantics of the region statement are illustrated by the following example:

```
var L: data_lock:
. . .
region L do . . . end region
```

The region statement implicitly performs a *Lock* on *L* before executing its body and guarantees that it will perform an *UnLock* on L on *any* exit from the body[3].

RED did not, however, demand that the variable mentioned in the region statement (*L* above) be of the pre-defined type, *data_lock*. Rather, the language merely defined that the region statement guaranteed to invoke *Lock(L)* and *UnLock(L)* at the appropriate places -- and defined what the semantics of these operations had to be. The pre-defined type *data_lock* satisfied these semantics and thus made the region statement immediately useful, but the user was free to define another type that satisfied these specifications and use it with the *region* statement. Doing so gave the programmer control over, for example, scheduling and resource allocation decisions that would normally have been pre-empted by the implementation. At the same time, the language *did* predefine *data_locks* so that the

[3]This guarantee includes exits caused by exceptions that are not handled by the body, return, exit, and goto statements in the body, abnormal terminations of the task in which the body is executing, and so on.

programmer who did not need more elaborate facilities or policies needn't be concerned with defining them.

In the following subsections we will explore a number of examples that illustrate how data abstraction can be used to avoid pre-emptive decisions. Each of these examples shares several attributes of the <u>region</u> statement:

- The steps involved in defining a flexible facility whose details are under programmer control will usually be (1) to reduce a distributed effect -- such as dynamic storage allocation, synchronization, or iteration -- to a (small) set of events, (2) to carefully delineate the effects that must take place in those events (the "essential semantics") and the variability that can be accommodated, and then (3) to give the programmer control over what happens at those points within the stated limits of variability.

- In order to turn control of incidental effects over to the programmer, we will define both a feature (generally an expression or statement) and one or more related abstract types. The semantics of the statement will be defined in terms of operations on the type(s). Constraints on the operations that the programmer is permitted to supply will enforce the necessary semantics.

Ideally, a full implementation of our proposal would involve a formal semantic definition of each of the types, their operations and the statements that relate to them. Moreover, ideally, programmer-defined implementations would be mechanically verified against these specifications, thus ensuring the validity of the whole. Alas, the technology does not seem quite up to either of these ideals -- as *yet*. The lack of this technology, however, does not prevent us from anticipating it by adopting the language design approach proposed here.

4.1. Loop Control

In [Shaw 77], the authors (with Ralph London) discussed a good example of the kind of language design approach that we are advocating here. The point at issue is iteration statements such as <u>DO</u> in FORTRAN and <u>for</u> in the Algol-Pascal family; both the definitions and the implementations of these constructs unnecessarily pre-empt too many decisions. Consider the Pascal program fragment,

```
sum := 0;
for i:=1 to n do
    sum := sum + A[i];
```

Clearly the purpose of this fragment is to form the sum of the elements of *A* in *sum*. However, the code is not an ideal expression of this: it specifies an explicit order for accessing the elements of *A* even though the order is immaterial, it refers to the variable *n* which we must presume is the size of *A*, and it uses the literal 1 which we must presume is the lower bound of the index of *A*. It would have been much better if we could have simply said "for each element of *A*, add that element once to *sum*". Doing so would certainly have been clearer -- and it might have been more efficient as well since most contemporary computers are a bit better at loops that count down to zero, and the less specific loop statement would permit a decrementing implementation.

Now consider another Pascal program fragment for a similar task:

```
sum := 0;
p := B;
while p <> nil do
    begin sum := sum+p^.data; p := p^.link end
```

This fragment forms the sum of the elements of B in *sum*. In this case, however, the data structure is a list, and as much code is devoted to tracing down the list as to forming the sum; the code has specific deficiencies comparable to those of the previous example.

These two examples serve to illustrate an even more serious deficiency. Each fragment makes a very strong suggestion about the representation of the data structure that contains the elements to be summed. This is a violation of the principle that such information should be localized -- every loop that processes the elements of the structure is affected.

As is illustrated by the second loop, the desire to iterate over the elements of a type is not limited to arrays. Indeed, most types, including programmer-defined ones, have one or more natural traversal orders. The natural traversals over the integers, for example, include increasing and decreasing intervals; these give rise to the common "stepping" forms of the iteration statement. In Alphard we provided a means for the programmer to define traversals for arbitrary types. In particular, we defined a for statement whose semantics are relative to an abstraction called a *generator*. By definition, a generator is an abstraction that provides a collection of operations with specified properties. To illustrate, the Pascal loop above could be written in Alphard as:

```
sum := 0;
for x from Invec(A) do sum := sum + x od
```

Here *Invec* is a generator; intuitively, it provides the sequence of values from the array *A*.

The formal definition of generators and the for statement are beyond the scope of this paper, but essentially a generator provides five operations: (1) *start* to initialize the loop, (2) *done* to determine whether the loop is finished, (3) *value* to get the value of the current sequence element, (4) *next* to step to the next sequence element, and (5) *finish* which performs any necessary clean-up. The for statement is defined to invoke these operations at obvious points; its semantics are captured in terms of a proof rule and a set of assumptions about the generator operations.

Of course, Alphard predefined a number of generators, including the traditional "stepping" forms, *Invec* (as used above), and others. However, the programmer was not limited to these alone. Any abstract definition that provided the proper operations could be used as a generator. Of course, in defining a generator, the programmer assumes the responsibility for verifying the assumptions that the for statement makes about generators.

4.2. Storage Layout

It has long been recognized that a compiler's decision about data structure layout isn't always the best one. Early languages did nothing about the problem except occasionally to provide explicit control of the size and position of fields in a record [Cobol 60] [Air Force 76]. More recently, concerns for efficient storage utilization have motivated features, such as Pascal's packed attribute [Jensen 74], that allow the programmer to select from a short menu of packing strategies without determining field placement explicitly. This packing control may also extend to aggregates other than records.

Unfortunately, control over the static placement of fields in records isn't enough. The arrangement of data may change dynamically, or the mapping between indices and elements may be complex, or compile-time binding of names to locations (or even to offsets in the stack) may be inadequate for some other reason. Because of the rich collection of possibilities for mapping data accesses, at the present time it does not appear (to us) that any purely declarative mechanism is adequate for describing the entire, useful collection. Lacking such a declarative mechanism, the only sufficiently powerful mechanism appears to be an algorithmic description of the accessing algorithm -- that is, a means of supplying an arbitrary computation.

One alternative is to allow the programmer to specify arbitrary computations to be performed when a name is accessed in either a right-hand (fetch) context or in a left-hand (store) context. Such a scheme is described by Geschke and Mitchell in [Geschke 75] . Although this clearly supports arbitrary representations it is, in a sense, too rich. That is, no built-in constraints ensure that the computations correspond to the reader's sense of what it is appropriate for an assignment to do. Even the example in [Geschke 75] illustrates this: The authors propose defining a *Vector* data type (point in 2-space) that supports manipulation of both polar and Cartesian interpretations of the value. The effect is to provide what appear to be record fields (*X*, *Y*, *Rho*, *Theta*) that interact in non-obvious ways; the value of, say, *Rho* can be affected not only by assignments to *Rho*, but also by assignments to *X* and *Y*. From the standpoint of program verification, this corresponds to violating the assignment axioms for the four fields.

An intermediate position is to insist on a correspondence between names and variables (locations) and to provide a mechanism for associating an address-calculation algorithm (sometimes called a *selector*) with each name. Additional protection is needed to guarantee the independence of these definitions, but if an address calculation is associated with the program name, the left-side and right-side calculations will at least be consistent. Such a mechanism is provided by structures in Bliss [Wulf 71]. A structure is very much like a macro except that parameters may be bound at the declaration site as well as at the use site. For example, the FORTRAN array

```
DIM A(100,25)
 . . .
A(I,J)=X
```

would be expressed in Bliss as a structure declaration (to define the notion of a two-dimensional array), followed by code analogous to the original FORTRAN.

```
structure FORTARY[a,b] = [a*b](.FORTARY+(a*(.b-1))+(.a-1));
own A:FORTARY[100,25];
 . . .
A[I,J] = X
```

The details of this code are irrelevant (especially the dots)[4], but in the structure declaration, the phrase "[*a***b*]" specifies the size of the area to be allocated and the remainder of the declaration specifies the accessing algorithm. In keeping with the general Bliss philosophy, there is no concern about safety or aliasing. The similar construct in Alphard, a *selector*, must guarantee that there are no extraneous side effects and preserve the properties of the assignment axiom.

Unfortunately, if selectors are prohibited from having side effects, they are not strong enough to handle some reasonable cases. Consider, for example, the problem of storage-efficient implementations of sparse arrays. No problem arises in selecting an element for which space has already been allocated, and a right-hand side access of a zero (or nonexistent) element can be handled by sharing one copy of zero among all such elements. A left-hand side access of an element that is currently nonexistent will, however, require side

[4]In evaluating A[I,J], a and b are the values supplied in the declaration, .a and .b are I and J , and, FORTARY is the address of the first word of A.

effects in the form of storage allocation or rearrangement of the data structure[5]. Thus we believe that some of the current alternatives for managing storage layout are too rich and others are too meager. We suspect that the resolution of the problem lies in finding a way to specify suitable constraints on an existing mechanism, not in the design of yet another mechanism.

4.3. Storage Allocation and Management

Pre-emptive decisions often deal with distributed effects in the program -- that is, with background computations that are not directly triggered by explicit operations in the code. Storage management is one of the most conspicuous of such cases:

- Allocation may be caused by block entry, explicit requests, or as side effects of primitive operations (e.g., CONS in LISP).
- Deallocation may be explicit, but it is more often an implicit effect of block exit or unreachability.
- The housekeeping for anything more complex than nested block structure (i.e., a stack allocation method) often requires processes such as garbage collection that are driven by interactions of individual decisions (i.e., the state of the heap) rather than by the individual allocation decisions.

Many algorithms have been devised for managing dynamic storage; these were classified by [Weinstock 76]. This study confirmed that no single allocation strategy is superior to the others, and that the special knowledge that is often available about particular situations can make a significant difference in the performance of the allocator. Programs written in assembly language can (often must) do their own storage management, but this degree of control is usually sacrificed in the move to a high-level language.

The most extensive language-level response we are aware of is Euclid's facility for collections and zones. In Euclid, dynamically allocated variables draw their storage from pools called *collections*, and each variable of a pointer type is associated with one of these collections. Although the primary motivation for introducing collections was to control aliasing, they have also been used as the units of storage management. Two policies are selected independently for each collection:

- The variables of the collection may be reference-counted (and automatically deallocated when the counts go to zero) or not.

[5]Indeed, in some implementations assignment of a zero value to an element that was previously nonzero may also require side effects to free the element.

- A storage management module, called a *zone*, may be associated with the collection.

Although the Euclid solution does not deal with the problem of distributed effects (it isn't possible to write a garbage-collecting allocator, for example), it does illustrate the decomposition we have in mind here.

To declare a storage pool for dynamically allocated variables of type *Entry* using a privately-defined management algorithm and some pointers into that pool, a Euclid user writes

```
var  Group: collection of Entry in MyZone
type Item = ^Group
var  ThisOne, ThatOne: Item
```

Variables are dynamically associated with *ThisOne* and *ThatOne* via calls on the standard procedure *New* that is associated with the collection *Group*. Since a private storage management module, *MyZone*, has been specified with the collection, the call on *New* will invoke a function *Allocate* that must be provided with *MyZone*. The requirement on the specification of *Allocate* is that it return a pointer to a suitable block of storage and that all such pointers be guaranteed to point to different variables. *MyZone* must also supply a block of storage in which to perform its storage management and a procedure *Deallocate* that will be called when the user invokes standard procedure *Free*. *Deallocate* is not required to do anything in particular, but in most reasonable systems it will return the freed space to the free list.

A completely safe and general solution to the storage allocation problem must also deal with issues of type safety, garbage collection, and specifications about storage usage (i.e., that storage is neither lost nor doubly-allocated).

4.4. Procedure Invocation

Pre-emptive decisions need not be limited to data and data-related aspects of a language. Subprograms are a good example where the language designer selects a particular mix of facilities and the implementor selects a single strategy for implementing that mix. Both the designer and implementor have only notions of typical use available; they are making the decisions too soon.

Some of the decisions that are typically made include:

- Whether to support coroutines or subroutines, or both. Shall the subprogram units be recursive or re-entrant?

- Which parameter binding classes to provide -- e.g., ref, name (as in Algol 60), value, result (as in Algol W), etc.

- When parameters are to be bound and the order of binding them.

The language SL5 [Hanson 78] has provided a decomposition of subprogram invocation into a number of events, and provided the programmer the means to define when those events occur -- and to some extent, what they are. The goals of the SL5 mechanism are not precisely ours (this is reflected in the supporting syntax of their facility), and so it is not a complete example of our proposed design approach. Nevertheless, it is an excellent example of the kind of decomposition we are suggesting be done for many language features.

Briefly, in SL5 (as in Algol68) the textual form of a procedure is viewed as the literal representation of a first-class object of the language -- that is an object that one can apply operations to, that can be assigned, etc. Thus,

```
procedure(x,y) . . . end
```

is a constant of type procedure, and the statements

```
a := procedure(x,y) . . . end;
b := a
```

first assign this object to *a* and then to *b*. A procedure, however, is not an executable entity -- an *environment* is. An environment, or activation, is created from a procedure. Thus, if *a* is the variable above,

```
e := create a;
```

will create an environment for the procedure. Given an environment, arguments can be bound into the environment -- that is, a correspondence can be established between the actual and formal parameters. This is accomplished by the with operator:

e with (e_1, . . . , e_n)

The value of the with operator is the same environment, *e*, but with the formals (re)bound to the actuals e_1 through e_n. Finally, given an environment with bound parameters, one can cause it to begin execution. Since SL5 wishes to make no commitment to a decision between subroutines and coroutines, the operator to initiate execution is called resume.

```
resume e
```

The value of the resume operation is the value "returned" by the invoked procedure[6].

[6]The value returned actually consists of two components -- a value and a signal. This is not essential to our point, however.

Before proceeding, two special cases are worth noting. The familiar syntax

```
f(a,b)
```

for function invocation is treated as a shorthand for

```
resume(create f with (a,b))
```

which, obviously, has the expected semantics. Also, the statement return is considered to mean "resume the last environment that resume'd me", with the caveat that "the last resumer" does not include the last returner"; this too corresponds to the expected semantics.

SL5 also permits user control of the binding class of parameters too. A formal parameter specification is of the form[7]

```
<id>:<exp>
```

where the expression, <exp>, defines something called a "transmitter". When arguments are bound with a with expression, the actual parameters are first "passed" to the transmitter associated with the formal; the value returned by the transmitter is actually bound to the formal. Predefined transmitters include *val* and *ref* for "by-value" and "by-reference" binding respectively. In general, however, any procedure can be used as a transmitter. This provides an extremely powerful facility that can be used for type checking and other forms of parameter validation in addition to the usual notions of binding.

5. Summary

Past investigation into abstraction techniques has concentrated on abstract data types and, in particular, on "building up" -- creating "bigger" things out of "smaller" ones. In the description of the bigger thing we suppress much of the detail about how it is constructed out of the smaller ones; this is the source of our leverage and power. The leverage is increased by generic definitions, which fix the essential properties of a type without constraining irrelevant but visible detail.

We propose using the same point of view for organizing a language and in particular for giving programmers control over invisible details. Instead of providing a language with fully-defined features, let us try to provide skeleton definitions that guarantee the essential semantics together with interface specifications for the parts that need to be filled in. In essence, this amounts to defining "generic language features" with constraints on the abstraction that can be provided to instantiate the generic features.

[7] There is also a scope definition as part of the formal parameter specification, but it is not essential here.

6. References

[Air Force 76]
Department of the Air Force.
Military Standard Jovial (J3).
MIL-STD 1588 (USAF), Rome Air Development Center, June, 1976.

[Ambler 77]
Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, Robert E. Wells.
Gypsy: A Language for Specification and Implementation of Verifiable Programs.
SIGPLAN Notices 12(3), March, 1977.

[Brinch Hansen 75]
Per Brinch Hansen.
The Programming Language Concurrent Pascal.
IEEE Transactions on Software Engineering SE-1, June, 1975.

[Cobol 60]
CODASYL.
Initial Specifications for a COmmon Business Oriented Language
Department of Defense, 1960.

[Geschke 75]
Charles M. Geschke and James G. Mitchell.
On the Problem of Uniform References to Data Structures.
IEEE Transactions on Software Engineering SE-1(2):207-219, June, 1975.

[Hanson 78]
David R. Hanson and Ralph E. Griswold.
The SL5 Procedure Mechanism.
CACM 21(5), May, 1978.

[Ichbiah 79]
J. D. Ichbiah, et al.
Preliminary ADA Reference Manual.
SIGPLAN Notices 14(6A), June, 1979.

[Jensen 74]
K. Jensen and N. Wirth.
Pascal User Manual and Report
1974.

[Lampson 77]
B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek.
Report on the Programming Language Euclid.
SIGPLAN Notices 12(2), February, 1977.

[Liskov 77]
Barbara Liskov, Alan Snyder, Russell Atkinson and Craig Schaffert.
Abstraction Mechanisms in CLU.
CACM 20(8), August, 1977.

[Nestor 79]
J. Nestor and M. Van Deusen.
RED Language Reference Manual
Intermetrics, Inc., 1979.

[Parnas 71]
David L. Parnas.
Information Distribution Aspects of Design Methodology.
In *Proceedings of IFIP Congress*, pages 26-30. IFIP, 1971.
Booklet TA-3.

[Parnas 72]
David L. Parnas.
On the Criteria to be Used in Decomposing Systems into Modules.
CACM 15(12), December, 1972.

[Parnas 74]
D. L. Parnas, J. E. Shore and Elliott.
On the Need for Fewer Restrictions in Changing Compile-time Environments.
NRL Report 7847, Naval Research Lab, November, 1974.

[Schuman 71]
S. A. Schuman, Ed.
Proceedings of the International Symposium on Extensible Languages.
SIGPLAN Notices 6, December, 1971.

[Shaw 77]
Mary Shaw, Wm. A. Wulf and Ralph L. London.
Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators.
CACM 20(8), August, 1977.

[Weinstock 76]
Charles Burr Weinstock.
Dynamic Storage Allocation Techniques.
PhD thesis, Carnegie-Mellon University, April, 1976.

[Wirth 77]
Niklaus Wirth.
Modula: A Language for Modular Programming.
Software -- Practice and Experience 7(1), January, 1977.

[Wulf 71]
Wm. A. Wulf, D. B. Russell and A. N. Habermann.
BLISS: A Language for Systems Programming.
CACM 14(12), 1971.

[Wulf 76]
Wm. A. Wulf, Ralph L. London and Mary Shaw.
An Introduction to the Construction and Verification of Alphard Programs.
IEEE Transactions on Software Engineering SE-2(4), December, 1976.