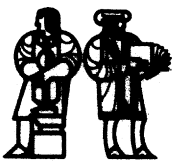
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-190

ABSTRACT DATA TYPES IN STACK BASED LANGUAGES

J. Eliot B. Moss

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

MIT/LCS/TR-190

Abstract Data Types in Stack Based Languages

by

John Eliot Blakeslee Moss

February, 1978

This research was conducted under a graduate fellowship from the National Science Foundation. Additional support was provided by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

Abstract Data Types in Stack Based Languages

by

John Eliot Blakeslee Moss

Submitted to the Department of Electrical Engineering and Computer Science
on November 28, 1977 in partial fulfillment of the requirements
for the degree of Master of Science.

Abstract

Abstract data types are the basis of an emerging methodology of computer programming. The only existing languages supporting abstract data types directly, CLU and Simula, both require compacting garbage collection, and thus they are not suitable for many applications. This thesis presents the design of a new language incorporating abstract data types; the language requires only a run-time stack, and not garbage collection. This new language, called ASBAL (for "A Stack Based Abstraction Language"), is based on CLU, and borrows as many features as possible directly from it. Virtually every significant feature of CLU is carried over into ASBAL in some form, and extensions are included when necessary. For example, the maximum size of objects becomes an issue and is resolved by the addition of size parameters to types. Also, a limited facility for dynamic storage allocation is incorporated in ASBAL to compensate for the removal of a garbage collected heap. This facility allows list and graph structures to be built within the framework of the stack while preventing dangling references as a "side-effect" of compile-time type checking.

Name and Title of Thesis Supervisor: Barbara H. Liskov
Associate Professor of
Computer Science and Engineering

Acknowledgments

This is the appropriate place to thank some people who were generous to me while I worked on this thesis. Let me first thank Bob Scheifler for reading many terrible drafts, and giving detailed and helpful comments on them. The thesis' clarity has been much improved by his work. I also appreciate the drawings and proofreading done by my wife, Hannah Abbott. She produced excellent figures in a rather short time. To all members of the Programming Methodology Group, thank you for your friendship and support, your ideas and dicussions. Several of you listened patiently to many half-baked ideas and helped me separate the wheat from the chaff, and I appreciate your taking the time to help me. Last, I thank the National Science Foundation, which has supported me as a Graduate Fellow for the last two and a half years.

Table of Contents

Abstract 2

Acknowledgments 3

Table of Contents 4

Table of Figures 8

1. Introduction 9

1.1. Motivation 9
1.2. The Goal 10
1.3. CLU and Garbage Collection 11
1.4. Our Approach 12
1.5. Related Work 12
1.6. Outline 13

2. Basic Concepts 14

2.1. Philosophy of Objects and Variables 14
2.2. Variables in ASBAL 15
2.2.1. Declarations and Names 18
2.2.2. Variable Initialization 19
2.2.3. Constants 19
2.2.4. Scope and the Form of ASBAL Modules 20
2.3. Procedure Invocation 22
2.3.1. The Different Classes of Arguments 22
2.3.2. A Simple Scheme 24
2.3.3. Returning Values vs. Passing Variables 24
2.3.4. Multiple Returns 26
2.3.5. Aliasing 27
2.4. Assignment 28
2.4.1. Multiple Assignments 31
2.4.2. Declarations with Initialization 32
2.5. Access to Components of Objects 32
2.5.1. Examples of Selectors 38
2.5.2. Summary 39

2.6. Implementation 40
2.6.1. Variables 43
2.6.2. Selections 45
2.6.3. Nested Blocks 45
2.6.4. Checking if Variables are Initialized 45
2.6.5. Aliasing 46
2.6.6. Summary 47
2.7. Programming Example 49
2.7.1. Bounded Queues of Integers 49
2.7.2. The Representation 50
2.7.3. The Operations 51
2.8. Conclusions 53

3. Two Extensions 54

3.1. Iterators 54
3.1.1. Implementing Iterators for ASBAL 57
3.1.2. Summary 58
3.2. Exception Handling 59
3.2.1. Exception Handling in CLU 59
3.2.2. Exception Handling in ASBAL 63
3.2.3. Summary 65
3.3. An Example - Sorted Bags of Strings 65
3.3.1. The Representation 66
3.3.2. The Operations 67
3.4. Summary 70

4. Parameters 71

4.1. Parameters in CLU 71
4.1.1. Parameters to Type Definitions 71
4.1.2. Restrictions 73
4.1.3. Parameters to Procedures and Iterators 74
4.1.4. Other Kinds of Parameters 75
4.2. Parameters for ASBAL 75
4.3. The Size Parameter Mechanism 77
4.3.1. Size Specifiers 78
4.3.2. The Kinds of Typeprocs 79
4.3.3. How Type Specifications are Used 80
4.4. An Example Cluster - Sequences 88
4.5. Implementation 93
4.5.1. Regular Parameters 93
4.5.2. Implementing Size Parameters 95
4.6. Analysis of Costs of Size Parameters 95
4.7. Summary 99

5. Areas and Pointers 100

5.1. Areas 100
5.2. Pointers 102
5.3. Using Pointers and Areas 102
5.3.1. Area Creation 103
5.3.2. Pointers and Areas in Reps 104
5.3.3. Closing the Loopholes 105
5.3.4. Summary 107
5.4. Pointers and Aliasing 108
5.5. The Copy Problem 109
5.6. Implementing Areas and Pointers 110
5.7. Example One - Queues 112
5.8. Example Two - Sorted Bags 114
5.9. Example Three - Symbol Table 117
5.10. Comparison of Area- and Stack-Based Programming 123
5.11. Summary 123

6. Summary and Conclusions 124

6.1. Suggestions for Further Research 125
6.2. Conclusions 126

Appendix I. Syntax of ASBAL 128

I.1. Formal Syntax 129
I.1.1. Modules 129
I.1.2. Parameters and Restrictions 129
I.1.3. Arguments, Returns, Yields, and Signals 130
I.1.4. Statements 130
I.1.5. Expressions 133
I.1.6. Types 135
I.1.7. Star Types 136
I.1.8. Question Mark or Star Types 136
I.2. Syntactic Sugars 138
I.3. Reserved Words 138
I.4. Terminal Symbols 139

Appendix II. Basic Data Types of ASBAL 140

II.1. Nulls 140
II.2. Booleans 141
II.3. Integers 141
II.4. Characters 142
II.5. Strings 143

II.6. Arrays 145
II.7. Records 148
II.8. Oneofs 150
II.9. Pointers 151
II.10. Areas 151
II.11. Procedures, Iterators, and Selectors 152

References 153

Table of Figures

Figure 1. Scenario of Simple Allocation Scheme 42
Figure 2. Scenario of the Two Stack Method 44
Figure 3. A Possible Layout for Stack Frames in ASSAL 48
Figure 4. Size Comparison 96
Figure 5. The Queue Cluster 113
Figure 6. The Sorted Bag Cluster 115
Figure 7. A Snapshot of a Symbol Table 119

1. Introduction

In recent years the correctness of computer programs has become a topic of growing interest. One approach taken to enhancing correctness is the formulation of design and programming methodologies. It is hoped that correctness will be achieved by using appropriate structure and discipline in the programming process. The use of *abstract data types*[1] is one of the techniques being developed. Abstract data types appear promising for simplifying proofs of program correctness, and seem to be natural for people to use in programming and in communicating among themselves about programs.

1.1. Motivation

To date only two programming languages have been implemented that provide and enforce the abstract data type discipline directly in the language: CLU [Liskov77], and extensions to Simula [Dahl66]. However, both languages require compacting garbage collection. The main difficulty with garbage collection is the "embarrassing pause" which occurs whenever the garbage collector is invoked. Such a pause is intolerable in real-time systems such as operating systems, process control programs, etc. One way to eliminate the pause is to use parallel [Steele75] or incremental [Baker77, Deutsch76, Barth77] garbage collection techniques. These methods have the effect of spreading the pause out uniformly over the normal processing time. Unfortunately, efficient parallel garbage collection probably requires special hardware, and is therefore not suitable for most applications, especially those relying on existing hardware. Incremental garbage collection appears to be more promising, but both parallel and incremental techniques for objects of different sizes (we say *variable size objects*) copy from one object space to another. Each space is one half of the total free memory; thus the maximum amount of memory usable with the parallel or incremental techniques is half of the actual memory provided. This severely restricts the applications possible on most machines, particularly mini- and micro-computers, which have small address spaces to begin

---

1. We assume the reader is familiar with abstract data types. For those less well versed in the recent literature the following papers may be helpful: [Liskov77, Wulf76a, Liskov75, Guttag75].

with. Another drawback to garbage collection is that the memory management system might be a prime candidate for using abstract data types. Clearly the garbage collector is part of the memory management system. If the language which allows one to use abstract data types requires garbage collection, then that language cannot be used to write the garbage collector.[1] One might hope that a useful subset of the language that does not require garbage collection could be used to write the garbage collector. However there is no such subset of either CLU or Simula: they depend on garbage collection entirely.

We feel that the *ultimate* solution to the garbage collection problem is building parallel garbage collection into computer hardware. As the cost of hardware continues to drop, and the cost of of software predominates, it may pay to double the memory required to have parallel or incremental garbage collection, and thus make available elegant and powerful programming languages that ease software development. However, in the interim, and for applications where the added hardware cost cannot be justified or address space is at a premium, we feel another solution is in order. In this thesis we present a new programming language incorporating abstract data types in a manner that does not require garbage collection.

1.2. The Goal

We have designed a new language with abstract data types that will run with only a run-time stack. This stack is similar to those used by other high level languages such as Algol, Pascal, and PL/1.[2] Rather than designing this new language from scratch, we have used CLU as a basis and have concentrated on retaining as many of its features as possible. To understand what we have done first requires a grasp of why garbage collection is necessary in CLU.

---

1. Without special tricks, that is. We feel that tricks of this sort should be avoided and an elegant solution found that does not involves tricks or special cases.

2. For some applications static allocation might be appropriate; we did not investigate that approach. However, see the suggestions for further research in the last chapter for more comments about it.

1.3. CLU and Garbage Collection

The coincidence of several parts of the semantics of CLU makes garbage collection a necessity. The basic unit of information in CLU is an *object*. Every object has a type, and may be manipulated only by the operations of its type: this is the key to abstract data types. Conceptually, objects exist independently of programs, and once created are never destroyed. A *variable* is simply a reference to an object, that is, a name for it. It is important to realize that variables do not contain objects, but rather object references, implemented by pointers or capabilities. Two variables may refer to the *same* object -- we say they *share* that object. Objects may refer to other objects, so objects can be shared by objects as well as by variables. This is useful in building hierarchical, graph, and list data structures. Furthermore, cyclic data structures can be built, thus implying that reference counting will not suffice to reclaim all unused storage. Because variable size objects are used, compacting garbage collection is needed to prevent fragmentation of storage.

In CLU, assignment, argument passing, and the return of results are all accomplished by transmitting object references; no objects are affected. For example, in

```
x := y;
```

the variable *x* is made to refer to the same object as that referred to by *y*. The object referred to by *y* is not affected in any way. In the case of argument passing a similar thing happens. The called routine is given references to the arguments being passed to it. This is not the same as call by reference: the called routine does not have access to the variables of its caller. However, the objects passed are shared between the caller and the called procedure. Therefore any modifications to the objects will be visible to the caller.

Any procedure can create new objects at will, and these objects are stored in the heap.[1] References to objects can be stored in other objects and also returned to the caller directly. We will call the object semantics of CLU the *object-oriented approach*.

In sum, CLU objects must be allocated in a heap because (a) they can be of unpredictable size, (b) their size can grow over time without bound, and (c) their lifetime is

---

1. A heap is a global, garbage collected storage area, like that of Algol 68 [Wijngaarden77].

indefinite. Garbage collection is required because cyclic structures can be built. Compacting garbage collection must be used to prevent fragmentation of memory, because variable size objects are used.

1.4. Our Approach

CLU's major contribution is the abstract data type facility, not the object-oriented view. It appears that the object-oriented view is the source of the requirement for garbage collection, as outlined above. Perhaps we can get the abstract data type facilities without the object-oriented view. As will be explained in more detail in Chapter 2, the semantics we arrive at is a synthesis of the traditional use of variables and CLU's use of objects. Sharing of objects is eliminated, and the lifetime of objects is tied to procedure activations by storing objects in variables. Further discipline is introduced by restricting the ways in which variables may be manipulated. Our purpose is to approximate the object-oriented view as closely as possible. We call our resulting design "A Stack-Based Abstraction Language", and refer to it as ASBAL. We have not attempted to design an entire language suitable for use, but have concentrated on the semantics. The syntax and some of the data types we use could be improved for a production language, but they serve us well in exposition.

1.5. Related Work

The goals of the Alphard language design group [Wulf76a] appear to be very similar to ours: Alphard has abstract data types and runs with only a stack. However, Alphard is still under development and it is not clear how similar Alphard and ASBAL really are. We suspect there will be significant differences because of our adherence to a more object-oriented approach. Furthermore, Alphard seems to concentrate more on provability (see [Wulf76a, Wulf76b, Wulf76c]).

The language Euclid [Lampson77] is also somewhat related to our work. We are especially indebted to Euclid for the concepts of aliasing prevention and collections. Like Alphard, Euclid is not object-oriented. Furthermore, Euclid was not specifically designed to provide abstract data types, although they can be modelled in Euclid to a great extent. Euclid

places more emphasis on provability than we do, and on systems implementation features. Euclid is a more complete language than ASBAL, but our intention was not to design a complete ready-to-use language.

The language Simula is also somewhat related to ASBAL. Simula could be described as CLU's ancestor, and CLU is ASBAL's ancestor, so the relationship is one of progressive development. No specific feature was consciously taken directly from Simula in the design of ASBAL, but much was taken from CLU.

The language most closely related to ASBAL is of course CLU, since it was the starting point of our design.

1.6. Outline

This introductory chapter is followed by five chapters. Chapter 2 introduces the basic semantics of ASBAL, laying the philosophical and semantic foundations for the rest of the design. The third chapter extends the basic features with two mechanisms taken from CLU: iterators and exception handling. Chapter 4 further extends ASBAL by adding parameters to abstractions. The parameter mechanism of CLU is copied, but a significant new feature is added - size parameters. The fifth chapter investigates a topic foreign to CLU: dynamic allocation of objects without requiring garbage collection. In Chapter 6 we summarize our research, draw conclusions, and make suggestions on how our work can be extended, mentioning other approaches to our problem deserving investigation.

There are two appendices, which more completely define our ASBAL. The first appendix gives a context-free grammar with explanations of the various productions. The second appendix outlines the basic data types and their operations.

2. Basic Concepts

This chapter presents many fundamental ideas of ASBAL. We begin with a discussion of the philosophy behind objects and variables in programming languages, point out particular aspects of this philosophy that directly impact our decisions in the design of ASBAL, and arrive at the basic semantics of variables for ASBAL. From this we develop the semantics of procedure invocation, assignment, and selection of components of objects. After a discussion of implementation techniques, we present an example type definition to illustrate the material introduced.

2.1. Philosophy of Objects and Variables

Variables in traditional programming languages serve two major functions: they provide a naming capability, and they provide containers for information (i.e., storage space). CLU, with its object-oriented view, separates these functions. Objects are the information containers of CLU. CLU variables hold only the names of objects. (We generally say the variable *denotes* or *refers to* the object.) Objects have an indefinite lifetime, and may be referred to by many variables at once. Hence, objects are implemented as storage allocated in a heap, with variables being pointers to the object they denote. Objects may refer to other objects, and general graph structures of objects are allowed.

Some objects have time-varying properties; we say such objects are *mutable*. The *state* of a mutable object is the set of properties it has at some point in time. For example, the abstract type *stack* is mutable. The state of a stack is the ordered sequence of objects in it. A push or a pop *mutates* a stack, i.e., gives it a new state. On the other hand, a test for emptiness will not change the state of a stack.

Immutable objects are those whose properties do not vary over time. Most mathematical values are immutable, as are their computer language models. (The *values* not the variables in which they are stored!) For example, integers, real numbers, characters, strings, and boolean values are all immutable. The integer '2' is an immutable object. '2' is '2' no matter how you slice it, and '2' can never be changed to any other integer.

The separation of the naming and storage functions of variables achieved by the

object-oriented view leads to a clean semantics. But probably the most important reason CLU's object-oriented semantics is attractive is that people seem to think in terms of objects. The very structure of language, with nouns (naming objects), adjectives (describing the properties of objects), and verbs (describing the use of objects or their behavior) seems based on this view of the world. If it is indeed true that people think in terms of objects, then linguistic forms that enable people to program directly in terms of objects could lead to better software design and implementation by being more natural for people to use.

Of course, the kind of objects to be found in programming concepts are highly abstract, often mathematical in nature. So, there remains much structure to be built to model real world objects and systems. This lack of structure allows the freedom necessary in a general-purpose language. For domain specific systems (e.g., medical diagnosis), more structure may be desirable because it embodies useful assumptions and prevents "reinventing the wheel" for every specific task. However, ASBAL is to be general-purpose. The abstract data type facilities allow one to build specialized systems by accumulating a library of type definitions and procedures relevant to the application. Our modeling of objects must extend to abstract data types to be useful. For this reason ASBAL is designed from a very general point of view with respect to types. This may make our descriptions of semantic concepts seem very vague. It is hoped that the many small examples we give will help offset the abstract descriptions.

2.2. Variables in ASBAL

As was discussed in the first chapter, our basic constraint in the design of ASBAL is to obviate the need for garbage collection. This, we argued, implies either static- or stack-allocation of storage. We explained that our investigation is restricted to stack allocation. CLU-style objects cannot be stack-allocated in any reasonable way, because they are very general structures. We have decided that the best approach for ASBAL is to store objects in variables similar to traditional variables. The simplicity of the method directed our choice. All other mechanisms we considered were intricate and complex. Storing objects in variables is not as nice as the full-blown object-oriented approach of CLU, but it appears to be the best we can do. The assignment, procedure call, and component selection mechanisms were designed very carefully to help offset the limitations imposed by working in a stack. Here is a summary

of the object interpretation.

A variable contains an object. An object has a type, and so does a variable; a variable may only contain an object whose type is the same as its own. Assignment will be used *only* to change which object is stored in a variable. An assignment effectively destroys whatever object previously existed in a variable, and creates a new object in its place. To change the *state* of an object, the object must be passed (using the procedure invocation mechanism) to an operation of its type.[1] An operation that changes an object's state is said to *mutate* the object.[2]

We emphasize that assigning to a variable is not the same as mutating the object it contains. This is because mutable objects may have properties defined by their creation which may never be changed later. For example, consider an abstraction *car* that models automobiles. At creation the make, model, and serial number are specified; these properties of a car may never be changed after it is created. On the other hand, the number of passengers in a car and location of a car can change quite frequently. Although all the properties of a car are part of its state, only some of these properties can be changed by mutation. However, if a car variable is assigned a new car, *all* the properties might be different from those of the previous car.

Objects may have other objects as components. Components of an object are stored within it, and hence within the variable containing it. This is quite different from CLU, where only references to components are stored in an object.

Several consequences of this object interpretation of variables should be mentioned. In CLU, assignment is system-defined: it is an implicit operation. This is because a CLU assignment entails only copying an object reference, rather like copying a pointer or capability. On the other hand we must construct an entire object in each assignment; this new object replaces the one previously residing in the assigned variable. The consequences of this fact will be explored in the section on assignment.

---

1. Recall that the abstract data type methodology allows only the operations of a type to access or update the representation of objects of that type.

2. The only mutable objects are records and arrays, and objects containing them. All mutation is accomplished by mutation of records or arrays. Mutation is in a sense magical, since the mutating operations are atomic. That is, the fundamental mutating operations cannot be broken down into other, more basic operations.

Another consequence of the object interpretation is that sharing of components is disallowed, because components are directly contained in their "parent" objects, rather than being referred to (pointed to) as in CLU. For example, in CLU two elements of the same array may be the same exact object, and any modification to this shared component object via one access path will be visible via the other access path. Our objects cannot share components in this way. (The addition of pointers to ASBAL restores the ability to share, so we are not giving sharing up completely.)

A rather obvious result of our semantics is that the lifetime of an object is bounded by the lifetime of the variable containing it, rather than being unbounded as in CLU. The major implication of this is that ASBAL routines will not be able to return objects in the sense that CLU procedures do. In CLU, procedures return *references* to objects; hence, previously existing objects may be returned by just copying references to them. In ASBAL we are restricted to constructing new objects to be returned.

The binding of an object's lifetime to that of its containing variable, along with the storing of components within objects rather than separately, requires a new mechanism for selecting components. In CLU, components can be selected by just returning them since only a reference is returned. On the other hand, our returns always create new objects, so returning a component cannot be done in the same sense as in CLU: we can only construct a *copy* of the component. Therefore, without a new mechanism, component objects may never be mutated, although new component objects may be substituted by operations on the containing object. Since we should be able to do anything with component objects that we can do with entire objects, a new mechanism is required to allow mutation of components. A new kind of module, the *selector*, is introduced for this purpose; it will be described in a later section of this chapter.

A last consequence of our semantic model is that objects cannot grow dynamically, at least not without bound, because they are restricted to the storage allocated for the variable containing them. This leads to difficulties when trying to implement abstractions that are conceptually unbounded. The parameter and area mechanisms to be presented later are largely devoted to solving this problem.

To sum up, variables in ASBAL are containers for objects. Objects have a type, which indicates how they may be manipulated, i.e., what operations are allowed on them.

Variables are also given a type, indicating what type of objects they may contain. Variables will be implemented as storage allocated in a stack; the object interpretation we espouse only puts limitations on how that storage may be manipulated. The major differences between our objects and heap allocated objects are:

(1) our objects are stored within variables, so assignment is different – the old object in the variable assigned to is destroyed and a new object created in the old one's place;

(2) because objects are stored in variables, an assignment always involves creating an object;

(3) there can be no sharing of objects among variables, or components of objects among objects and variables;

(4) the lifetime of an object is bounded by the lifetime of the variable containing it;

(5) and, the size of an object is bounded by the size of the variable containing it.

The next few sections of this chapter explore the consequences of these differences in more detail, and present ASBAL's mechanisms for the manipulation of objects.

2.2.1. Declarations and Names

Programs must be able to refer to objects in order to manipulate them. Hence, (some) variables in ASBAL will be given names in programs. These names are called *variable names* to distinguish them from names used for other purposes such as naming types and procedures. It is generally convenient to create a new variable of some type and give it a name at the same time. In ASBAL a declaration statement such as

```
var x: foo;
```

is used to do this. In the example, a new variable of type *foo* is created and given the name *x*. A newly created variable, as constructed by a declaration, contains no object; it is an error to attempt to use it. (We have more to say about initialization of variables below.) One can easily extend the form of the declaration statement to allow declarations of several variables at once:

```
var x: int, y: bool;
```

or

```
var x,y: int;
```

2.2.2. Variable Initialization

We define our declarations to create new variables, that is, ones never before known or used. This definition prevents confusion over whether a "new" variable contains an old object. It does raise two problems, however. The first is that memory allocation is required – this is discussed in the section on implementation later in this chapter. The second problem is that the bits initially in the storage allocated for the variable may not represent a legal object of the type of the variable. There are two solutions to this problem. One is to store a default object of the declared type in the variable as part of the actions taken for the declaration. This can be done for user defined types as well as system-provided ones by requiring each type definition to have a routine of a particular name (*init*, say) which will store an initial object in a variable given to it by the system. This solution guarantees that variables always contain *legal* objects (assuming users do not write crazy *init* routines!). But unfortunately, it cannot guarantee that the objects are *sensible*, since sensibility depends on how a variable is used.

The better solution is to consider attempts to use an uninitialized variable as illegal, and to detect such attempts with a combination of compile-time and run-time checks. Exactly what checks are required is discussed in the section on implementation later in this chapter.

2.2.3. Constants

It is sometimes convenient to have a holder for an object that cannot be assigned to after initialization, and that does not allow the object to be modified. We call such holders *constants* to contrast them with variables; they are similar to constant objects in CLU. However, we allow constants of mutable types, such as constant arrays. Since a constant physically contains the object stored in it, modifications can easily be prevented by disallowing any write operations to the storage allocated to a constant. We will see later that we can pass a variable to a procedure but have the procedure consider it to be a constant. This is the real motivation for constants – prevention of undesired modification to objects.

A *constant definition* is similar to a variable declaration, except that the object to be stored in the constant must be specified. Thus, in a constant definition we give the desired name, type, and the object to be stored in the constant:

```
const n: int = 53;
const i: int = j * k;
const a: array[int] = array$create(0);
```

In an implementation there is little difference between a constant and a variable: a constant is essentially a write-once variable.

2.2.4. Scope and the Form of ASBAL Modules

To gain an understanding of the scope of variable and constant names, we must consider the general form of modules in ASBAL. The two basic modules of ASBAL are the *cluster*, which implements a data abstraction, and the *procedure*, which implements a procedural abstraction.

A cluster defines a data abstraction, by giving a representation (often shortened to *rep*) for the abstraction being defined, and implementations of the operations. The operation implementations take the form of procedures, but have the added ability to convert objects of the abstract type to and from the rep type. Internal operations may be used; the cluster lists which of the operations may be used outside the cluster.

A *procedure* has a header and a body, the body being a list of statements. The header gives the types, and names of the arguments, the types of any results returned, and other information to be described later.

Each abstraction is implemented in terms of lower level abstractions. The overall structure is a hierarchical decomposition, with the highest level abstractions at the top, and the lowest level abstractions being types and procedures built into the language. A module is an implementation of an abstraction.[1] Because an abstraction is entire unto itself, a free standing

---

1. A module may implement a *class* of related abstractions, rather than a single abstraction (see the chapter on parameters).

mathematical object, modules are conceptually separate and independent.[2] For example, there are no free variables in ASBAL modules, because this would represent a dependence on another module to supply those variables.

This model is somewhat contrary to the more common block-structured view of programs in at least two ways. First, the block-structured view leads to large monolithic programs, and the whole goal of modularity is to prevent such large programs. Second, we allow only local variables, not global variables. This supports modularity by making module relationships more explicit: any data that a module wishes to access must be passed as arguments to that module. Since each procedure defines a distinct abstraction, and every abstraction is implemented by distinct modules, nothing is gained by defining procedures within procedures. In the interest of simplicity procedure definitions in procedures are forbidden. However, hierarchical nesting of statement groups within a procedure is quite desirable, so it is allowed and encouraged.

What scoping of names is proper for this modular viewpoint? Without local procedures there is little reason to allow variable names and constant names to be obscured (reused in nested blocks), especially since procedures are not expected to be very large. However, it is often helpful to restrict the scope of certain variable (or constant) names to an inner block, such as a loop, rather than an entire procedure; this helps indicate the purpose of the variable.

Our no-global-variables policy makes programs more modular, but makes some programs a little more awkward when global data is necessary. The main advantage of global data is not having to explicitly pass it to every procedure that might use it. An example of an object normally made global is the symbol table of a compiler. Assume we must implement a compiler in a language forbidding global data. Let us say the compiler parses by recursive descent. Only a few routines directly access the symbol table; however, the symbol table must be created at the highest level and passed explicitly through many routines that never use it at all. These intermediate routines only pass the symbol table down for the lowest levels to use. We

---

2. This has nothing to do with separate compilation, however. Modules may or may not be separately compiled: we do not wish to pin this aspect down.

feel the modularity gained by forbidding global data more than offsets the inconvenience of requiring extra writing for some programs. Removing global data is essential to eliminating implicit module interdependencies. Block structure is not bad in itself; global data is. However, once all data is local, there is little point to block structure for module definitions.

Even though all data should be local, we argue that module names should be global. It is not useful to restrict the scope of modules, and in fact it can be counterproductive – it may force abstractions to be re-implemented needlessly. Therefore we assume that module names are global. We neither require nor prohibit other information regarding the relationships of modules – such module interconnection information is beyond the scope of this thesis.

2.3. Procedure Invocation

The previous section discussed variables and constants, the mechanisms for storing and holding objects. We now continue with procedure invocation, which allows the creation of new objects and the manipulation of old ones. The next section will deal with assignment.

2.3.1. The Different Classes of Arguments

The whole point of procedures is to gain abstraction in actions. A set of actions that form a logical whole is grouped together and viewed as a single abstract action. The basic actions are mutation of objects and assignment to variables. Since all data is local in ASBAL, the key to procedural abstraction will be the argument[1] passing mechanism; that is, the mechanism by which procedures are given access to data to operate upon it.

We can imagine as many as four different classes of arguments in ASBAL. The first class is *constant arguments*. A constant argument to a routine is an input which cannot be directly modified by the routine. We will see later that a procedure cannot count on an constant object's not changing state, because there may be an access path from some other argument to the object that allows it to be mutated. However, in the absence of pointers, a constant argument cannot be mutated by the called procedure in ASBAL. Furthermore, if all

---

1. We reserve the word *parameter* for a future use, and carefully distinguish between arguments and parameters.

arguments to a procedure are constant arguments or result arguments (see below), then the procedure is functional; that is, it does not modify any of its arguments.

The second class of arguments is *object arguments*. An object argument gives access to a particular object, allowing observation and mutation of it. However, the variable containing the object may not be accessed, and therefore may not be assigned to.

The third class of arguments is *variable arguments*. A variable argument is a variable passed by reference. Therefore, assignment to it is allowed, as well as access to (and mutation of) the object it contains. The difference between variable arguments and object arguments is exactly the difference between assignment to a variable and mutation of the object it contains.

The last class of arguments is *result arguments*. A result argument is a variable which may *only* be assigned to. The purpose of result arguments is the construction of new objects in variables, that is, assignment. This includes initialization as well as assignment.

Object and variable arguments (the second and third classes described) are not very much different from each other in implementation. Both would be implemented by passing by reference. The only difference is that a variable argument may be assigned to, and an object argument may not be. This slight distinction is not worth the complexity of two separate argument passing modes. Therefore, we chose to dispense with one and keep the other: we retained object arguments, and eliminated variable arguments, for two reasons. First, this is the more conservative choice in that less access is given to arguments. Second, object arguments more like CLU's argument passing mechanism. In CLU, object references are passed, by value. The effect is as if immutable objects were passed by value, and mutable ones by reference; *except*, the variables of the calling procedure cannot be affected by the called procedure in any way. However, the object passed is shared between the procedures, and hence mutations of it performed by the called procedure will be visible to the calling procedure. The decision of which class of arguments to keep is not all that important in the long run, but has affected later decisions such as the selector mechanism and aliasing detection.

Now that we have settled on the classes of arguments – constant, object, and result – we need to devise a syntax for expressing procedure definitions and invocations. Let us first describe a simple scheme which we will improve upon in a moment.

2.3.2. A Simple Scheme

The simplest approach to defining procedures is to have a header in each definition, much like the procedure headers of Pascal. In the header we state the local name, type, and class of each argument. For example:

p = proc (const w,x: int, var y: array[int], res z: bool);

The above header says that procedure *p* takes four arguments: two constant arguments, *w* and *x*; one object argument, *y*; and one result argument.[1] The procedure *p* is not allowed to mutate or assign to *w* and *x* (integers are not mutable objects anyway); *p* may mutate *y*, but not assign to it; and *p* must assign to *z*, but may not access it beforehand. Call by reference is used to implement all three kinds of arguments; the difference between them is in what the called procedure may do with an argument – not how the argument is passed.

Procedure invocations take the usual form: the name of the procedure followed by a parenthesized list of arguments. For example, a call of the procedure *p* used above might look like this:

p (1, i+5, a, b);

The types of the arguments must match those declared by *p*. Furthermore, access constraints may not be violated. Thus constants may not be passed as var arguments, or expressions as res arguments.

2.3.3. Returning Values vs. Passing Variables

The simple scheme outlined above is perfectly workable, but can easily be improved upon. The main thing to notice is that there is no explicit assignment. *All* assignments are accomplished by passing a variable by res. (Presumably the built-in types have operations to assign to a variable of their type. In a sense these operations are magical, since all other assignments rely on them.) However, the procedure invocations necessary for each assignment are tedious to write out in the simple scheme, and they obscure what is happening since result

---

1. We admit the use of var for object arguments is not the best, but var was used to parallel Pascal. Anyway, we do not wish to get involved in purely syntactic issues.

arguments do not stand out.

It is possible to separate result arguments by writing them on the left-hand side of a ':=' symbol, to signify assignment. For example, we would write:

```
var b: foo, c: bar;
b := q (x, y);
c := r (z);
a := p (b, c);
```

where in the simple scheme we would have written:

```
var b: foo, c: bar;
q (x, y, b);
r (z, c);
p (b, c, a);
```

assuming these to be the types of *p*, *q*, and *r*:

```
p: proctype (var foo, bar, res T1)
q: proctype (var T2, T3, res foo)
r: proctype (var T4, res bar)
```

The use of ':=' shows more clearly what is going on.

We can make a further improvement, however. If we had to declare a variable for every temporary result, our programs would become quite cluttered with extraneous variables and declarations. We can get around this problem by having the compiler allocate temporary variables. Adding this feature allows us to rewrite the above example and eliminate the temporary variables *b* and *c*:

```
a := p ( q(x,y), r(z) );
```

Further, if some procedure returns a result we never use, we need not assign the result to a variable; the compiler will allocate a temporary variable for the procedure to write into, and then the temporary will be thrown away (i.e., never accessed again). So, if the variable *a* were never used again in the example, we could eliminate it, giving:

```
p ( q(x,y), r(z) );
```

The end result of putting res arguments on the left, and having the compiler allocate temporaries, is syntax quite similar in appearance to CLU. In fact, we encourage the programmer to think of procedures as returning objects instead of being passed variables to write into. The overall picture of this final scheme is that the calling procedure gets the effect

of objects being returned, and the called procedure sees variables which must be assigned to. This is a good compromise between abstraction and efficiency. The only constraint is that the size (or at least an upper bound on it) of all objects to be returned must be known before the call, so that the actual variable used can be created. How we deal with this constraint will become clear later.

To encourage thinking in terms of returning objects, we put the description of what a procedure returns in a separate part of the procedure header, as in

```
p = proc (const w,x: int, var y: array[int]) returns (z: bool);
```

The objects to be returned are given names because the procedure being defined views them as variables. Therefore, we now call the result arguments of a procedure *return variables*. Notice that effectively all we have done is segregate the res arguments.

Now let us consider how to express the returning of objects in ASBAL. In principle we could use a return statement like CLU's, which gives a list of objects to return. This would be implemented by implicitly doing assignments to the return variables. However, these implicit assignments might involve the copying of large objects into the return variables. Instead, we allow objects to be built incrementally in the return variables, and simply say

```
return;
```

to return from a procedure. We view the return variables as being uninitialized on procedure entry, and any return statement in the procedure is considered to be a use of all the return variables. This allows us to use whatever mechanism already exists for detecting the use of uninitialized variables to handle return variables as well. In sum then, the underlying mechanism of returning is the passing of variables (whether they be programmer declared or compiler created). However, we arrange the syntax so that people can think of returning objects, a view we feel is more natural.

2.3.4. Multiple Returns

In most languages, procedures may return only zero or one things. We remove this restriction because it is arbitrary and sometimes counterproductive, in that some procedures most naturally return more than one object. Of course, we provide suitable syntactic forms for using this feature. The return statement itself need not be extended since we are depending on

assignments to get the return objects into the return variables, as previously explained. However, some syntactic form is necessary to designate the variables to receive the return objects. The *multiple assignment* statement, which will be discussed in detail in the section on assignment, is used for this purpose. Its general form is

var_1, var_2, ..., var_n := *invocation;*

The header for a procedure returning more than one object would have a returns clause of the form

returns (var_1: $type_1$, ..., var_n: $type_n$)

where the types may be factored. For example:

returns (x,y: int, z: char)

The order of the variables on the left side in the multiple assignment statement is the same as in the returns clause of the procedure header. This parallels the standard correspondence of actual and formal arguments to procedures. The returns clause may be omitted for a procedure returning no objects, or

returns()

may be used.

2.3.5. Aliasing

We have not dealt with the problem that arises when the same object is passed to a procedure in two different var positions, or in both a const and a var position. The problem is that not all procedures are prepared to deal with overlapping variables. The problem is compounded by the fact that there are variables that (effectively) have subvariables (e.g., records and arrays), and overlapping subvariables present the same difficulty. Furthermore, the fact that each argument has a different name in the called procedure tends to make people forget that two names might refer to the same object (or overlapping objects). We call the problem *aliasing* (after Euclid [Lampson77]). We believe that aliasing should be illegal. One very good reason for prohibiting aliasing is that it can cause an argument to mysteriously change into an entirely different object from that passed. Consider the following procedure:

```
p = proc(a: array[t], x: t);
    ...
    a[10] := ... ;
    ...
    a[i] := x;
    ...
    end p;
```

It is reasonable to think that (a) *p* has no effect on *x* since it does not assign to it in the body, and (b) that after the second assignment, *a[i]* equals the argument passed in. However, one could call *p* in this way:

```
p (b, b[10]);
```

Assuming that both arguments are passed by reference, we see that the assignment to *a[10]* in the body of *p* can destroy *x*. See [Lampson??] for more arguments as to why aliasing should be prohibited.[1]

Most cases of aliasing can be detected at compile-time, but some require simple run-time checks, e.g., that two array indexes are different for the call

```
f (a[i], a[j]);
```

and so on. We will explain what must be done to prevent aliasing in the section on implementation, and will expand the rules to cover features that impact on aliasing as we encounter them.

2.4. Assignment

Here we describe how to change which object is stored in a variable – the operation commonly called *assignment*. As mentioned in the discussion of argument passing and procedure invocation, return variables (which were previously result arguments) are the only assignment mechanism. It is easy to see how this works for assignments of the form

```
var := invocation;
```

that is, variables being assigned a computed expression: the variable is passed as the outermost procedure called. (It will be demonstrated that virtually all expressions are really invocations,

1. There are no implicit arguments in ASBAL, unlike Euclid. This should reduce the number of checks required to prevent aliasing.

even if they are not explicitly written out. For example,

```
x + y
```

really means

```
T$add (x, y)
```

where *T* is the type of *x*.) What about assignments of the form

var$_1$:= *var*$_2$; ?

There is no invocation there to pass *var*$_1$ to! This problem can be handled in three ways.

First, there could be a system-defined, automatic copy operation performed. This is what happens in most languages. Ignoring differing storage formats, etc., the implicit copy performed is essentially a bit-for-bit copy of the contents of the storage allocated to *var*$_2$ into the storage of *var*$_1$. We call this operation a *bit-copy*. A bit-copy works fine in the absence of abstract data types, but with their introduction a problem arises. Any assignment creates a new object; a bit-copy creates one with the same state as the one in the right-hand variable. The problem is that not all types should be copied in this way. For example, some types may require all the existing objects of the type to have different states, so that each object is detectably unique. In the presence of pointers, it is not clear whether a pointer which is a component of an object to be copied should itself be copied, or whether the object pointed to should be copied.[1] Thus, an automatic copy primitive is not feasible.

The second solution is to have all assignments

var := *exp*;

mean

var := T$copy(*exp*);

(where *T* is the type of both *exp* and *var*), whether *exp* is a variable or an invocation. This has the unfortunate effect of doing a redundant copy whenever *exp* is not a variable. Furthermore, the redundant copy operation is hard to optimize away because users write the copy operations, and are not constrained to make them easily optimized.

We feel the best solution is to insert no extra copy in assignments of the form

---

1. This is called the *copy problem* and will be further discussed when pointers are added to ASBAL.

var := *invocation*;

and to take

$var_1 := var_2$;

to mean

var_1 := T$copy ($var_2$);

The type of *T$copy* is assumed to be

proctype (const T) returns (T);

If an assignment of one variable to another is written, and the appropriate copy operation does not exist, then the program is in error.

Let us point out a few consequences of the solution we have adopted. First, every operation must provide a copy operation if objects of the type are to be assigned from one variable to another. There is no getting around this; the second solution had it also, and we demonstrated the first solution to be infeasible. Second, the ':=' symbol has a non-uniform meaning. While we sympathize with those that believe symbols should have clearly defined unique meanings, we feel we must compromise that principle in this case. What is gained is a savings in effort at optimization, or computer time in program execution.

There is one more problem with assignments: consider the statement

x := p (x, y);

Here *p* receives *x* as an argument in two positions; one is readable, and one is write-only. Things could get really messed up when *p* starts to write into its return variable. One way to solve this problem is to translate it to

x := T$copy (p (x, y));

similar to the solution of the previous problem. Inserting a copy here is a bad idea, though, because it is nowhere near as obvious as before when an extra copy will be inserted and when it will not. The better solution is to allocate a temporary variable and pass it to *p*. Then after *p* returns, a bit-copy is performed from the temporary into *x*. A bit-copy works because the state of the object in *p* is exactly the state desired for the new object in *x*; furthermore, the object in the temporary is never accessed again.

2.4.1. Multiple Assignments

In a previous section we introduced the idea of returning more than one object from a procedure. We need to be able to assign those objects to variables. The form of assignment statement for this is

var_1, var_2, ..., var_n := *invocation;*

To extend this to its logical (and useful) conclusion, we also allow simultaneous multiple assignments of the form

var_1, var_2, ..., var_n := exp_1, exp_2, ..., exp_n;

Each variable var_i is to be assigned the corresponding expression exp_i, and all these assignments are to take place simultaneously. To prevent confusion we require that each expression either be a variable or return only one object. In case of aliasing, the same trick of using temporaries works fine. For example, in

```
x, y := q (z, r(y), x);
```

a temporary would be allocated for the result destined for *x*. On the other hand, one is not needed for *y*, because *y* is not an argument to *q*.

One particularly nice construct the multiple assignment statement allows is

```
x, y := y, x;
```

It is hard to decide if this should just swap the bits of the objects stored in *x* and *y*, using bit-copies, which is both efficient and semantically correct, or whether it should invoke *t$copy* twice,[1] which is more consistent with our above rule about assignments between variables. We believe it is better to be consistent (i.e., to call *t$copy*). A new operator could be used to swap the objects in variables, but we will not explore such possibilities here.

---

1. For "x, y := y, x;" two temporaries might be required; however, it is not difficult to have a compiler notice that one of them is not needed.

2.4.2. Declarations with Initialization

One last useful assignment statement is a declaration with initialization (or assignment with declaration). This form of statement allows one to declare and assign to a variable in one step. Here are two examples:

```
var x: foo := p (z);

var x: foo, y: bar := q(t), r(u);
```

A declaration with initialization is effectively[1] a shorthand for a declaration followed by an assignment. Thus the second declaration above is equivalent to

```
var x: foo, y: bar;
x, y := q(t), r(u);
```

which is in this case equivalent to

```
var x: foo, y: bar;
x := q(t);
y := r(u);
```

Constant definitions, which were introduced in a previous section, have the same effect as declarations with initialization. The only difference is that constants can never be assigned to again.

2.5. Access to Components of Objects

The previous sections of this chapter have dealt with mechanisms for manipulating objects as a whole; here we discuss the additional mechanisms necessary for manipulating components of objects. There are three actions that can be performed on objects: objects may be created, they may be observed (read), and they may be mutated. We desire to be able to do all three to components of objects as well as to entire objects. Creation is no problem. A component of an object is either created when the object is created, or is created by a

---

1. In Chapter 4 we will see that there can be an important difference between a declaration with initialization and one without. However, for now, consider the declaration with initialization to be equivalent to a declaration followed by an initialization.

(mutating) operation on the object. Records are an example of objects whose components are created with the objects themselves. Arrays exhibit the other behavior: the *addh* and *addl* operations allow new array elements to be created dynamically. (Records and arrays will be described in more detail in a moment.) Abstract data types may display either or both component creation behaviors; they may always possess some components, but create (and possibly destroy) other components dynamically.

Reading components is already taken care of as well. Since all objects having components are built from records and arrays, and records and arrays have operations to read their components, any type can provide operations to read any components it may have. Of course a type may not make all components available externally, and may return information derived from the components rather than the components themselves. However, reading components is always done by returning objects. This is unfortunate, because returned objects are always copies – always new objects. (Remember that return variables must always be assigned to.) Thus, returning does not allow components of objects to be mutated; only *copies* of the components may be manipulated.

It may seem that storing a mutated copy back into a data structure is equivalent to mutating a component of the data structure, and this is often true. However, many data structures do not allow components to be replaced at will in this fashion. As an example, consider queues; perhaps we can observe the member at the front of the queue, but we can only insert new members at the end of the queue. An even better example is items that must be mutated atomically rather than by separate reading then writing; semaphores and other synchronizing data types fall into this category. Copies are sufficient for observing components, but a special mechanism is needed to allow mutation of components.

In a previous section of this chapter we indicated that the operations of an abstract data type are procedures. We now design a new kind of module, the *selector*, which is also allowed as an operation of a type. Here is what a selector does. A selector is given an object from which to select a component, and possibly some *auxiliary arguments* to describe which component is desired. The selector then proceeds to calculate whatever array indexes, etc., are required, and eventually executes a *select* statement. The select statement indicates the component object to be made available for use. What is returned to the caller of a selector is

not a new object, but rather a descriptor of the component selected. (That is, an object reference is returned.) The selected component may be used as a var argument to a procedure, and can thereby be mutated. However, what is selected is an object, and hence may not be assigned to; only variables may be assigned to.

Since a reference to (descriptor of) an object is returned by a selector, we must guard against any dangling references. Potentially, a selector could select one of its local variables rather than a component of the object it is supposed to select from, giving rise to a dangling reference when the descriptor is returned. We prevent this by requiring that selectors never select any of their local variables (or components thereof). Notice that procedure returns cannot create dangling references of this sort. A procedure always creates a new object in its return variables; procedures can never store object references in return variables.

There are two minor points to mention regarding mutability. First, components selected from var's should be var, i.e., mutable, and components of const's should be const. Therefore, a selector does not designate whether its object to select from is const or var; that property is automatically inherited from the incoming object. Furthermore, a selector may not mutate the object being selected from; hence the object is treated as a const inside the selector for checking purposes. The second point is that a selector should not mutate any auxiliary argument. Therefore, all auxiliary arguments are taken to be const.

The form of a *selector definition* is:

name = selector (*name*$_1$: *type*$_1$, *name*$_2$: *type*$_2$, ..., *name*$_n$: *type*$_n$) of *type* from *name*$_0$: *type*$_0$;
  statements;
  end *name*;

The *name*$_i$ for $i > 0$ are the auxiliary arguments; *name*$_0$ is the object to select from. The 'of *type*' part indicates the type of the object to be selected. A select statement (which is only legal in a selector) takes this form:

select *expression*;

The expression cannot designate a local variable or auxiliary argument of the selector.

It is harder to decide on the exact form of a *selection*, the construct that invokes a selector. We could use

selector_name(object to select from, auxiliary arguments)

to be like procedure invocation, but we feel it is better to write

object.selector_name(auxiliary arguments)

to be analogous to records. The latter form also has the advantage of making the object being selected from stand out. If the selector takes no auxiliary arguments, the parentheses may be omitted, leaving

object.selector_name

which is just like a record component selection.

In many cases computing a selection can be expensive. Therefore, we provide a mechanism for saving a selection; it is the with statement:

```
with class name == exp do
   statements
   end with;
```

where *class* is const or var. If the class is var, then the selection must be from a var. The *name* stands for the selected object within the body of the with statement, and is treated according to the declared class. A scope is used because extra checking must be done for safety. To prevent mutations of the containing object from destroying the selected object, all arguments to invocations in the body of the with statement are checked for overlap with the selection.

For example, say (bounded) queues are implemented as arrays. If the front member of a queue is held in a saved selection, then the queue may not be modified until the scope of the with statement is exited. This is because an element of an array (the front member) overlaps with the array itself (the queue). The checking to prevent this aliasing is done using the normal aliasing detection techniques. (The checking may be difficult to accomplish at compile-time, however.) The with statement is similar to the *bind* operation in Euclid.

Now that we have described the essential nature of selectors and selection, let us discuss where selectors are appropriate and where they are not. Selectors are to be used to mutate objects stored in a surrounding data structure without disturbing that structure. The types having selectors will usually be ones that store data items and relationships between them, but do not manipulate the data items directly. Good examples are lists, stacks, queues, trees, graphs, etc. Selectors should definitely not be used to access components that cannot or should not be

mutated. Furthermore, selectors should not be used merely to make access more efficient,[1] for this can lead to (effectively) exposing the representation and thus limit the range of implementations of a data type. For example, consider the functions *real*, *imag*, *abs*, and *arg* on complex numbers. Implementing any of these functions as a selector forces that component of complex numbers to be represented explicitly in the representation. Hence, selectors threaten the uniform reference principle [Geschke75, Ross69]. Thus, the specifier of a type must use caution when deciding whether particular operations should be procedures or selectors.

We now describe records and arrays. It is important to understand their semantics, for they are the principal types used in defining representations of abstract data types. A record type has named fields, each specifying a type. For example,

```
record[a: int,
       b: bool,
       c: ralph]
```

Each field name defines a selector with the specified name; the type of the selector is

seltype () of *type of field* from *record type*

Record components may be changed. The operation 'put_*field_name*' is used to update the named field of the record. The type of 'put_*field_name*' is

proctype (var *record_type*, const *field_type*);

The new object is constructed using the *field_type*$copy operation, which must exist for *field_type* to be usable in a record. For convenience, record *put* operations have a sugar; one may write

exp_1.*field_name* := exp_2;

instead of

record_type$put_*field_name* ($exp_1$, exp_2);

Notice that record put operations are "magical" atomic mutating operations. Records also have *copy* and *equal* operations; records are more fully described in Appendix II.

The only other operation on records is creation. This cannot be written out without giving an order to the fields. We feel it is better to think of the fields as being unordered, and

---

1. Selectors do save a copy operation over procedures returning an object.

so the user may not invoke the record create operation directly. Instead there is a special form called a *record constructor* which allows creation of record objects in an order-independent way. A record constructor takes this form:

$$record\_type\${field\_name_1: exp_1, field\_name_2: exp_2, \ldots field\_name_n: exp_n}$$

The field names must all be present exactly once, but in any order. The fields are computed in the order listed.[1] Several fields may be initialized to (copies of) the same object by writing

$$field\_name_a, field\_name_b, \ldots field\_name_z: exp$$

The record constructor invokes the appropriate copy operations for each expression which is a variable, and for each expression which is stored in more than one field.

An array *object* is a sequence of objects, of a single type, indexed sequentially. The sequence may be empty, and can grow and shrink in size dynamically. Arrays have a selector to index them; it is called *fetch*, but there is a shorthand for indexing arrays. If an element with index i currently exists in the array a, then $a[i]$ selects that element, as does the unsugared form $a.fetch\ (i)$.

An array *variable* can hold only certain array objects of its type. More specifically, each array variable has associated with it an interval of the integers, and only arrays whose indexes are all in that interval may be stored in the array variable. (We emphasize that the indexes of an array object and those allowed for an array variable are both sets of consecutive integers.) The allowed indexes for an array variable are set when it is declared, and never change thereafter. Thus, an array variable of type array[foo;low,high] can be assigned any array object whose elements are foo's, and whose indexes are all greater than or equal to *low* and less than or equal to *high*. The type of the array object is array[foo]. (This difference in the number of parameters and the ';' notation will be explained in the chapter on parameters.)

There are operations on arrays that allow adding and removing elements from the high or low end (i.e., growing or shrinking the array one element at a time at either end) (*addh*

---

1. For ASBAL to be well-defined, the order of evaluation is always specified. Unless explicitly mentioned, that order is left to right.

and *addl*), trimming to a particular range of indexes (*trim*), querying the size (*size*), low index (*low*), and high index (*high*), shifting the elements (*set_low*), and replacing the elements (*store*). This last operation, *store*, has a sugar similar to that for the record *put* operations. We may write

$exp_1[exp_2] := exp_3$;

in place of

array_type$store ($exp_1$, exp_2, exp_3);

Both forms mean "replace the component at index number exp_2 in the array exp_1 with a copy of exp_3". See the appendix for a complete list of array and record operations.

Arrays were designed in this (somewhat unusual) way to be convenient for use as representations of abstract data types, and to prevent access to uninitialized elements. However, they are a bit more expensive than arrays found in most programming languages, in space, and in time.

2.5.1. Examples of Selectors

Suppose we had an abstract type *associative_memory*, which associates pairs of integers. We represent an associative memory as an array of records; each record has two components, one for each integer of the pair. Thus the representation type of the associative_memory cluster is

array[record[first, second: int]; 1, 100]

assuming a maximum of 100 elements is allowed. The associative memory is to have an operation *update* which will change the second element of a pair, based on the first element. *Update* will have in it a statement like

a[index].second := new;

which is a sugared form of

RT$put_second(a.fetch(index), new);

where *RT* is 'record[first, second: int]'. Thus, we have shown how a selector may be used.

Let us now consider an example of a type providing a selector itself: a bank account record file. It is convenient to design the structure used to access the individual account records of a bank independent of designing the records themselves. Of course the two designs

interface in the area of the keys used to search for the records; but except for the keys (and the size of the records), no properties of the records affect the design of the access structure. Likewise, the access structure has no real effect on the properties of the records. Let us suppose the file of all account records is a (rather large) object of type *account_file*, and that the type of the individual records is *account_record*. Since account records are mutable, we design *account_file* with a selector of type

seltype (key_type) of account_record from account_file

This allows us to realize the separation of access from use. This separation contributes to abstraction by reducing dependencies among different types. In the absence of selectors, we would be forced to implement all update actions on account records as operations on account files, and present the appropriate key every time. Furthermore, the access would have to be recomputed every time. Thus not only are more type dependencies created (by making all record updates go through file operations), but performance is reduced as well. (Remember, though, that performance arguments alone do not justify using a selector.)

On the other hand, if a selector is used to access the records, then a restriction is being placed on every implementation, namely, that account records must be represented explicitly in account files, and that it must be possible for programs to access account records directly once the records have been selected.[1]

2.5.2. Summary

We have presented a new module, the *selector*, designed specifically for ASBAL's object interpretation semantics. Selectors allow components of objects to be selected dynamically and passed to procedures to be mutated. A type has the ultimate control over the components of its objects, and need not allow them to be selected. Furthermore, only the object can change the *identity* of its components, since selected components may not be assigned to. (Selections produce objects, *not* variables.) Records and arrays were introduced as prime examples of types

---

1. Notice that selectors do not solve any of the problems associated with accessing objects on external storage; ASBAL assumes all objects exist in a single, uniformly accessible address space.

providing selectors. We argued that selectors are necessary, but are to be used sparingly so as to avoid having types depend on having a particular representation.

2.6. Implementation

Now we come to the question of how to implement all of these features.[1] First, we are going to allow recursive (and mutually recursive) procedures, so a stack of procedure activation records is required. These frames (as we also call the activation records) are very much like those used to implement languages such as Algol and PL/1. Each frame contains the storage for the (local) variables and temporaries of the procedure activation to which it corresponds. Since a finite (and usually small) number of variables are used in a procedure, it is possible to give each variable a fixed offset from the beginning of the frame, which can be very efficient on many machines. As for arguments and return variables, they will be passed by address. The slots for these addresses can also be at fixed (possibly negative) offsets from the start of the frame, since the argument addresses may be put on the top of the stack by the calling procedure before the frame is created.

Using fixed offsets in this way fails only for local variables and temporaries whose size is not known at compile-time. (However, descriptors with slots for pointers to those parts of a variable that are allocated at run-time, can still be stored at fixed offsets from the start of the frame.) Most types have a fixed size, and we will not discuss the mechanisms for using types of varying size until the chapter on parameters. On the other hand, we present the implementation now since it affects other parts of the design of ASBAL.

Most cases can be handled by simply allocating the required amount of storage on the top of the stack as soon as the size is known. (This storage is accessed through a pointer at a fixed offset in the frame.) There are only two situations where this does not work perfectly: declarations with initialization, and temporaries in the middle of expressions. As we will see later, the size of these variables may not be known until just before the procedure which is to

---

1. We assume the reader is fairly familiar with implementation techniques for stack based programming languages, so we often gloss over details that are not novel and do not present special problems of implementation.

initialize them is invoked. Unfortunately this is after all the arguments to the invocation have been computed; if any of those arguments are themselves temporaries, then allocating the space for the return variables at the top of the stack will result in a "hole" when the temporary is freed. Let us present a simple example to demonstrate the creation of these holes in the stack:

```
var x: foo := p (q(y), r(z));
```

where the size of the *foo* is not known until just before *p* is called.

(1) The stack starts as in part (a) of Figure 1, with *y* and *z* in the current stack frame.

(2) A temporary variable *t_q* is allocated, and *q* is called (1b).

(3) Another temporary *t_r* is allocated and *r* is called (1c).

(4) Space for *x* is allocated and *p* is called (1d).

(5) The stack is left as in part (e) of Figure 1, with a hole between *x* and the rest of the variables.

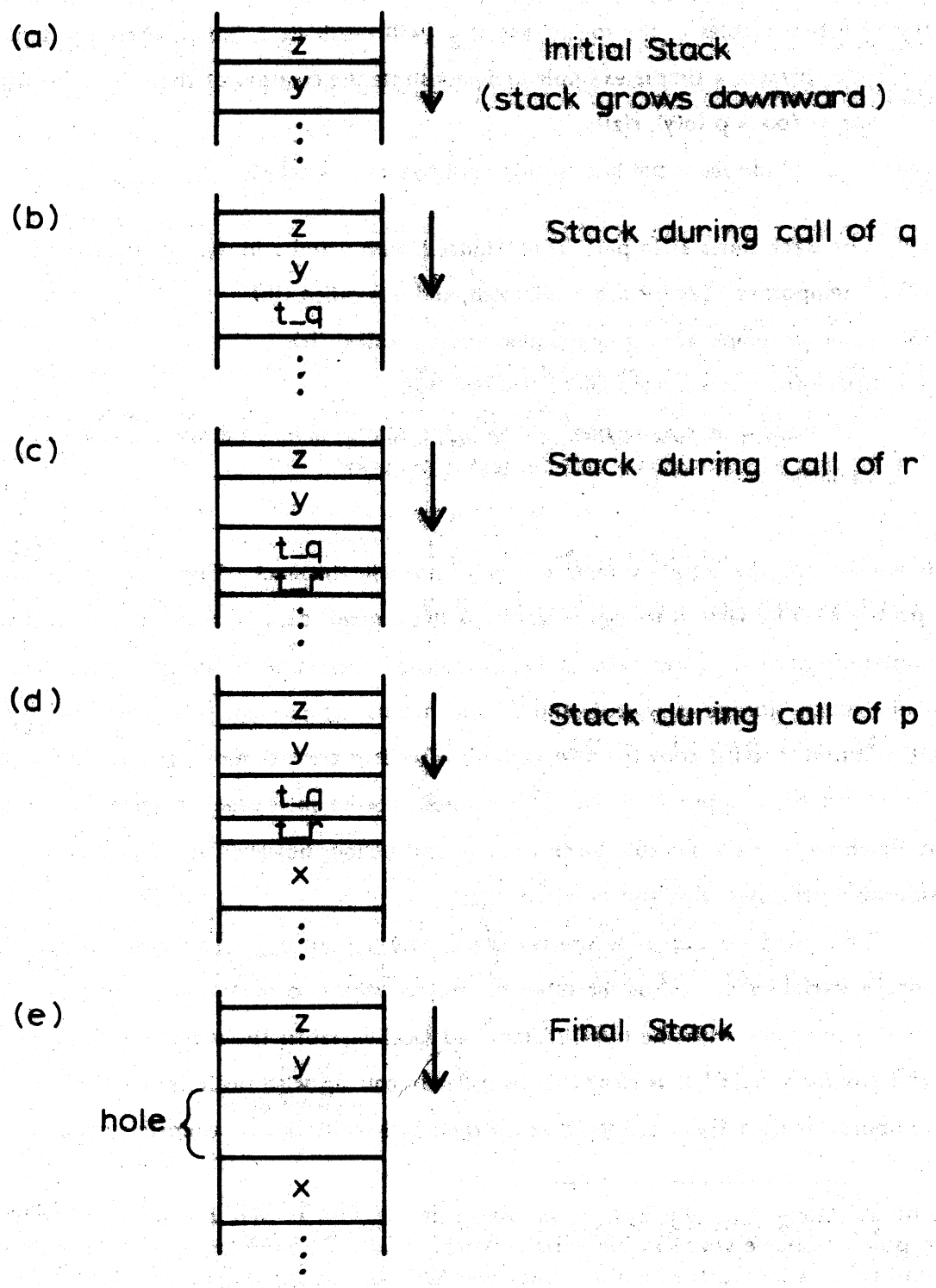
Thus we see that the simple scheme will leave holes in the stack. There are three solutions to this problem. The first is to ignore it; this is not a good idea for more and more holes could accumulate (e.g., in recursive calls) and cause considerable waste of storage. Still, it is not clear just how much storage is wasted, and it may not pay to prevent this particular waste. The second solution is to bit-copy the new variable after it is created, moving it to the beginning of the hole, and thus eliminate the hole. This need not be inefficient in terms of code because many machines have a suitable block transfer instruction; however, the copying might use up considerable processor time and memory cycles.

The third solution is to use two stacks rather than one. The basic idea is to allocate temporary variables on one or the other of the two stacks so that neither ends up with holes. Let us call the stack with the usual frames and local variables the *variable stack*, and the other one the *auxiliary stack*.[1] It is clear that in order to end up with no holes on the variable stack the temporaries used for a call must be on the auxiliary stack. A symmetric argument leads to

---

1. The auxiliary stack will have to be set up into frames as well, but its frame pointers and stack pointer can be saved in the variable stack. Thus all housekeeping information is kept in the variable stack with the auxiliary stack used only for storing temporary variables.

Figure 1. Scenario of Simple Allocation Scheme
Execution of 'var x: foo := p (q(y), r(z));'



the converse fact: that to avoid holes on the auxiliary stack, temporaries needed during the computation of intermediate temporary values must be put on the variable stack. What happens is that we alternate between the stacks according to the nesting depth of a particular temporary in an expression. Let us examine another scenario to illustrate this scheme. We will go through the execution of

```
var a: foo := p ( q(r(), s(t())), u(v()) );
```

The evaluation is strictly left to right. Figure 2 shows a sequence of relevant snapshots of the stacks. It is not at all hard to figure out which temporaries should be put on which stack if one works backwards from the desired final configuration. Note also that the use of the two stacks is purely for the evaluation of expressions within a procedure. Any procedure that is called during the expression evaluation can put its local variables and temporaries on top of either stack so long as it cuts both stacks back to their previous state before returning. Notice also that both the one-stack and two-stack schemes handle multiple returns easily, by allocating more than one variable at once.

It is not too hard to see how to implement two stacks on a computer: one starts at low addresses and grows upward, and the other starts at high addresses and grows down. There is some time and space overhead involved in keeping two stack pointers and frame pointers instead of one each, but there are no severe technical problems. So, we have seen that two stacks are better than one.[1]

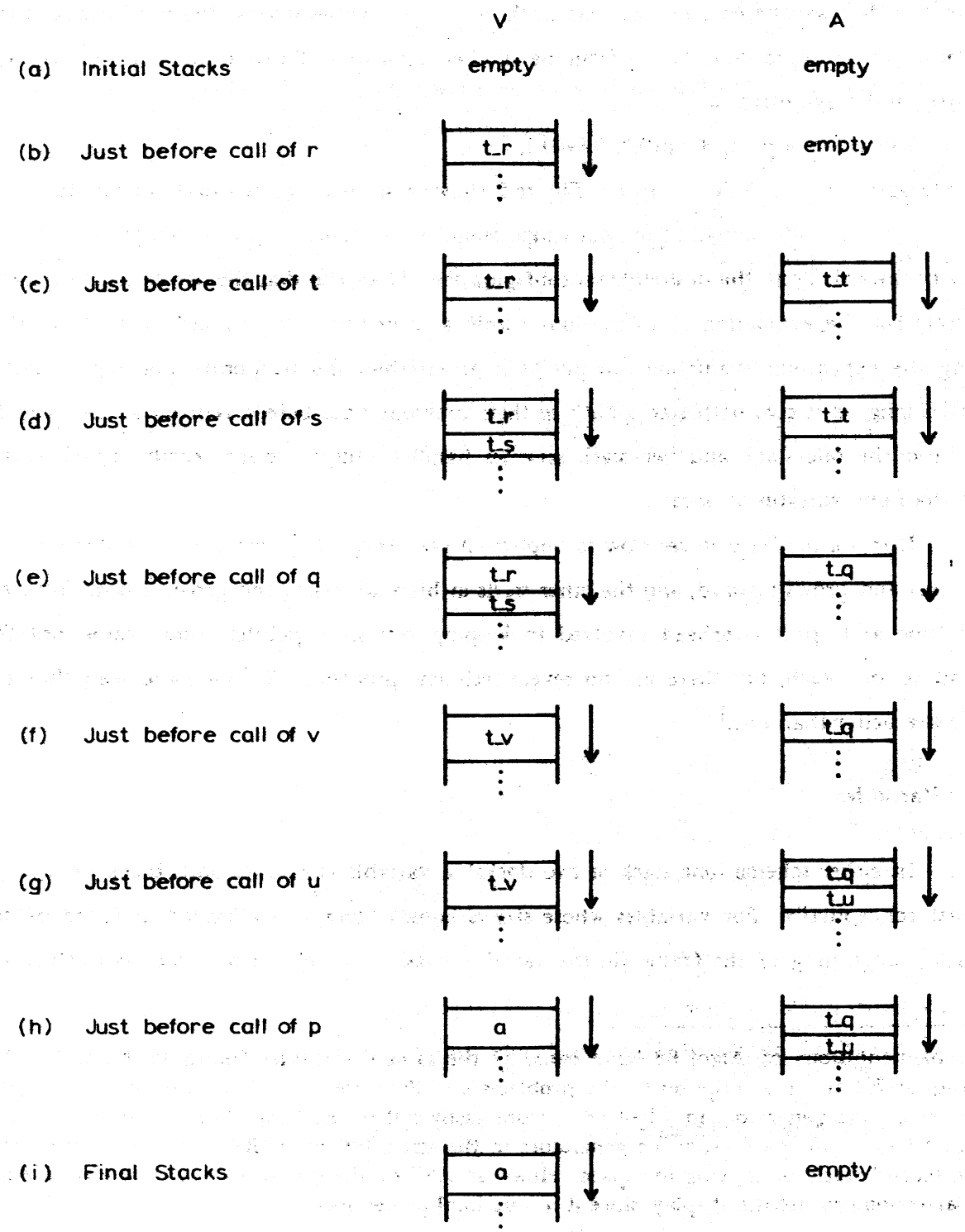
2.6.1. Variables

In either scheme (one stack or two stacks), a variable is a contiguous block of storage, at least conceptually. For variables whose size is known, storage is allocated at fixed offsets from the beginning of the frame (in the variable stack). For those whose size is not known,

---

1. Implementations of Algol 68 have many of the same difficulties found in ASBAL. (See [Branquart70] for a description of the problems and their solution.) For example, some space reserved by loc generators in Algol 68 is more easily put in the heap than on the stack. It is possible to put all space from loc generators in the stack, but in ASBAL we must resort to a heap, second stack, or copying the space. However, ASBAL does have an advantage over Algol in that it does not need a display, since it has no local procedures.

Figure 2. Scenario of the Two Stack Method
Execution of 'var a: foo := p(q(r(), s(t())), u(v()));'



space is reserved at a fixed offset for whatever information is necessary once the variable is created. This can all the variable's fixed size parts, and slots for the sizes and addresses of its variable size parts, which are filled in when the variable is allocated. The figure at the end of this section shows a possible layout for stack frames.

2.6.2. Selections

A selection can be implemented as a pointer to (or descriptor of) the object it denotes. Slots for these pointers are easily allocated at compile-time because they have a fixed size and are of finite number. Even better, the number of selections is apparent from the text of the program. Thus, allocation for selections is no problem.

Checking that selectors do not select a local item, etc., is more challenging. A compiler can perform the checks by analysis of the expression given in the select statement of the selector. The expression must be the object to select from, or (more usually) a selection from that object. The other checks (e.g., that the auxiliary arguments are not mutated) are handled by other checking mechanisms with no special casing. Saved selections (in the with statement) present no more problems than regular ones, and are implemented the same way.

2.6.3. Nested Blocks

Instead of using a full frame for nested blocks, it is probably easiest to append their fixed size space to that of the enclosing blocks, making one large fixed size block. Of course blocks at the same nesting depth can use the storage in different ways since only one of them can be active at once. The part of their storage that is unknown in size can be managed in stack fashion: allocated beyond the space for the enclosing block, and cut back when the nested block is exited. These are well known techniques.

2.6.4. Checking if Variables are Initialized

Now we describe the checks necessary for insuring variables are always initialized before use. First, let us see how much a compiler can check. It is clear that truly sophisticated checking might involve complicated analysis of the control flow of a program. However, we

would like to keep the analysis to a minimum. Exclusive use of so called "structured programming" control-flow statements greatly simplifies the analysis. The critical feature of such statements is that the number of paths through a procedure is kept to a manageable size. The compiler can keep a record of which variables are assigned to in every block. From this it is fairly easy to combine the information, separating variables into three classes:

(1) those definitely initialized at every use;

(2) those definitely uninitialized at some use;

(3) and those possibly uninitialized at some uses, and definitely initialized at the rest.

The first class is all right; the second indicates an incorrect program; and the last class requires the insertion of run-time checks. For each variable of the last class, the compiler allocates one bit of memory in the run-time stack frame to be used as an indicator of whether that variable has been initialized. These bits all start in the no state. At the appropriate places on the questionable paths, code will be generated to set the bit to the yes state, or to test it. Even if a variable is used and assigned to in many places, this extra code will be inserted in only a few. This, along with the fact that the code is short (one or two instructions on most machines), means that there is little run-time overhead. We feel that the overhead is well worth it, particularly when debugging programs. Notice that this same scheme checks for initialization of return variables; all we need do is consider those variables to start uninitialized, and view the return statement as a use of all of the return variables.

2.6.5. Aliasing

The checks required to prevent aliasing are straightforward. They depend on a simple inductive principle: if there is no aliasing when procedure p is called, (i.e., none of its arguments or return variables overlap), and all local variables of p are disjoint (none of them overlap), then we can guarantee that p introduces no aliasing in the invocations it makes. The compiler does this by making sure that no variables subvariables are aliased in the calls p makes.

To implement aliasing detection we need a description of which components of

variables overlap, and which do not. The Euclid report [Lampson77] gives a very detailed definition of which variables overlap in that language. We will be content with a less formal, more intuitive description. First, it is obvious that a variable overlaps with itself. It is also clear that a record overlaps with any of its components, and an array with any of its elements. This carries down through all levels, so an array of records overlaps with any component of any of its element records. On the other hand, if two variables do not overlap, such as two local variables with different names, then none of their subcomponents overlap either. When two variables overlap, one must contain the other; hence, when two variables do not overlap, they are completely disjoint.

How do we extend aliasing detection to general selections? First of all, any selection comes from a particular object in a particular variable. We need only check selections from the same object. An object and any selection from it are considered to overlap. Selections from the same object generally require a run-time check. This check ascertains whether the two selections overlap physically in storage. The starting address for each selection is always available at run-time, but the length of each must be provided in addition to the starting addresses.

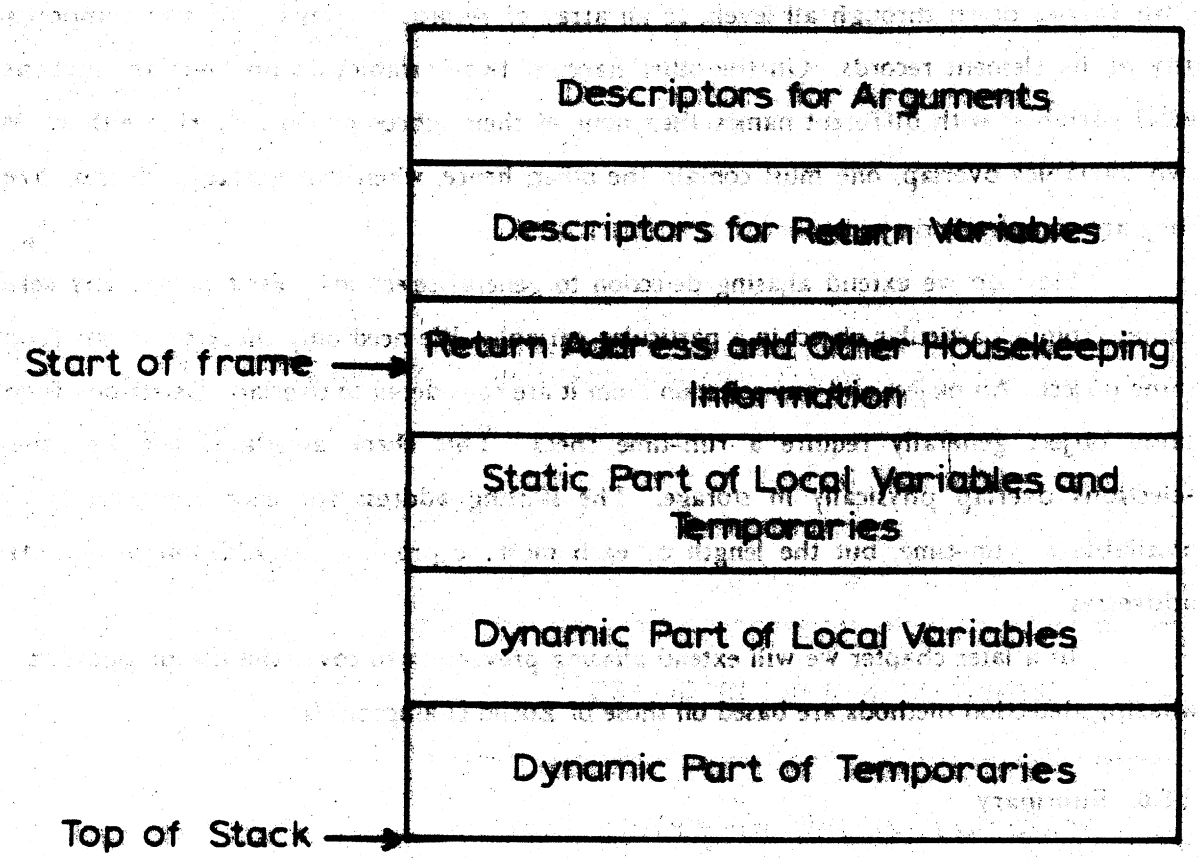
In a later chapter we will extend aliasing prevention to cover the use of pointers. Our aliasing detection methods are based on those of Euclid [Lampson77].

2.6.6. Summary

ASBAL requires one stack to be maintained by its run-time system (but may do better with two). The stack frame for a procedure activation contains the local variables references to the arguments and result variables, and housekeeping information (return address, old frame pointer, etc.). For most variables, fixed offsets into the current frame can be used. Some variables require a certain amount of descriptive information (descriptors or dope vectors), mainly those whose size is not known at compile-time. Figure 3 shows a possible layout for stack frames.

Argument passing is by reference, i.e., the addresses (or descriptors) of arguments are passed to a procedure when it is invoked. Returned results are simply extra argument variables; the addresses of the variables are passed. Most of the checking for aliasing and

Figure 3. A Possible Layout for Stack Frames in ASBAL



uninitialized variables is handled easily at compile-time, and the run-time checks do not amount to much code. We conclude that our scheme is about as efficient as possible given the level of safety we require and the features we want in the language.

2.7. Programming Example

In this section we will present a programming example to help illustrate the fundamental ideas introduced in this chapter. The example is a type definition, but since clusters (the form of type definitions) have procedure and selector definitions inside them, all three module types will be illustrated. Later we will see that using more advanced features allows us to write better definitions for the type we now present, but at this point we are restricted to the most basic of features.

There are two essential parts to a type definition in ASBAL: the *rep* (representation) *type*, and definitions for the operations. As in CLU, we group these together in a single module called the *cluster*. The syntactic form is:

```
type_name = cluster is names_of_operations_exported;
    rep = rep_type;
    operation_name = proc ... ;
    ...
    operation_name = selector ... ;
    ...
    end type_name;
```

The procedures and selectors may be mixed. There also may be internal procedures and selectors; an internal operation is one that can be called only from within the type definition. Internal operations are distinguished by the fact that they do not appear in the list of exported operations.

2.7.1. Bounded Queues of Integers

In this first example, the task is to define and implement a new data type, a bounded queue of integers. The operations of this type and their functionality are listed below.

```
create:     proctype () returns (queue)
            (creates a new, empty queue)

insert:     proctype (var queue, const int)
            (inserts the integer at the end of the queue)

remove:     proctype (var queue) returns (int)
            (removes the front member of the queue)

is_empty:   proctype (const queue) returns (bool)
            (returns true if and only if the queue is empty)

is_full:    proctype (const queue) returns (bool)
            (returns true if and only if the queue is full)

size:       proctype (const queue) returns (int)
            (returns the number of members in the queue)
```

These queues will be bounded in size, and the maximum size will be 100.

2.7.2. The Representation

It is easy to decide what representation to use for this type. An array of 100 integers will hold the members of the queue, and will be managed in circular buffer fashion. One index will be maintained: the position of the first member of the queue. The members will be stored in order of increasing indexes in the array, modulo 100. The size will be kept explicitly. Thus our type definition will begin:

```
queue = cluster is create, insert, remove, is_empty, is_full, size;
   rep = record [first: int,
                 size:  int,
                 q:     at];
   at = array [int; 0, 99];
   ...
   end queue;
```

2.7.3. The Operations

We will write the create operation first. We set the *first* component of the rep to be zero, the size to be zero, and fill the whole array with zeros. (The array is filled for efficiency; it does not matter what it is filled with in this case.) The entire create operation is presented below:

```
create = proc () returns (q: cvt);
    q := rep{first: 0,
             size: 0,
             q:    at$fill (0, 0, 100)};
    end create;
```

The notation cvt (from *convert*) indicates a variable or constant whose type is viewed as the abstract type (i.e., the type being defined) outside the module, and the rep type inside. Of course it is only allowed in a module defining a type, and always means that type by context. The expression *at$fill(i,low,num)* denotes an array object, all of whose elements are copies of *i* (made by using *i$copy*), for all indexes in the range *low* to *low+num-1* inclusive, provided *num* is not negative. (Calling *at$fill* with a negative third argument is an error, and what happens will be explained in the next chapter.) Notice that there is no return statement in the procedure above; for convenience, the end statement of a procedure does an implicit return.

Let us move on to *size*, *is_empty*, and *is_full*.

```
size = proc (const q: cvt) returns (s: int);
    s := q.size;
    end size;

is_empty = proc (const q: cvt) returns (e: bool);
    e := (q.size = 0);
    end is_empty;

is_full = proc (const q: cvt) returns (f: bool);
    f := (q.size = 100);
    end is_full;
```

Our integers and booleans are like those of any other language; details are in the appendix on data types. The use of '=' in *q.size = 0* actually indicates a call of 'int$equal (q.size, 0)'. This use of syntactic sugar allows us to extend symbols such as '=', '+', '-', '*', '/', etc., to abstract types.

Full information on these sugars can also be found in the appendixes.

Now let us write the insert operation.

```
insert = proc (var q: cvt, const val: int) returns ();
    if q.size = 100 then error end if;
    var index: int := (q.first + q.size) // 100;
    q.q[index] := val;
    q.size := q.size + 1;
    end insert;
```

The '//' is a sugar for *type*$mod, that is, the modulo (or remainder) operation of the type. Notice the use of sugared array and record component replacement operations. The next chapter will present a mechanism for signalling and handling exceptions, but for now we will write *error* to indicate that appropriate code has been omitted.

The remove operation should now be easy for the reader:

```
remove = proc (var q: cvt) returns (member: int);
    if q.size = 0 then error end if;
    member := q.q[q.first];
    q.first := (q.first + 1) // 100;
    q.size := q.size - 1;
    end remove;
```

Finally, here are example calls of the operations: (The symbol '~' is prefix not, a sugar for *type*$not (*expr*)).

```
var q: queue := queue$create();

if ~ queue$is_full (q) then queue$insert (q, foo); end if;

if ~ queue$is_empty (q) then bar := queue$remove (q); end if;

var s: int := queue$size (q);
```

This data type (queue) just happens not to use any selectors. However, there will be several examples of selector definitions in later chapters.

2.8. Conclusions

This chapter has dealt with the fundamental semantics of ASBAL, a language intended to preserve as many of the abstraction features of CLU as is possible under the constraint of a stack-oriented semantics and implementation. We started with the notions of variables and objects. We then went into the semantics and the syntax for procedure call and return. Aliasing was discussed, and rules formed to prevent its occurrence. A satisfactory solution to the problem of uninitialized variables was presented, and an implementation outlined. Next, the mechanism of assignment was explained, followed by a discussion of component selection. After discussing implementation, we presented an example to illustrate these concepts. The groundwork has been laid for presenting more advanced features of ASBAL. The next chapter will introduce two new features: iterators and exception handling.

3. Two Extensions

In this chapter we extend ASBAL by the addition of two new features – iterators and exception handling. Iterators introduce a new kind of abstraction, and are implemented by a new kind of module. On the other hand, exception handling just completes the previously seen modules; it changes them from partial functions to total ones by allowing them to specify and deal with exceptional cases. We will present each feature as it is in CLU, and then modify it as necessary for ASBAL.

3.1. Iterators

A major goal of abstraction in programming is to move the programmer away from details and into working at a high conceptual level. Procedures provide functional (or procedural) abstraction, and clusters provide data abstraction. Another useful kind of abstraction has been identified, the *control abstraction* [Liskov77, Wulf76b]. The only sort of control abstraction we will offer is a generalization of looping called an *iterator*, based on the iterators of CLU. A loop has three basic parts:

(1) generation of the sequence of data items to be operated on,

(2) operating on the data, and

(3) testing for completion.

Iterators provide a modular way of generating the sequence of objects to be operated on. In CLU, an iterator generates a sequence of objects that are passed to the body of a loop. The crucial point is that an iterator generates the sequence of items incrementally: one object at a time. This will be easier to understand by following through an example.

Let us say we have an abstract type *binary_tree*. Further suppose that many of our programs that use *binary_tree*'s need to examine all the leaves of a tree in left to right order. If we are given operations to fetch the left and right subtrees of a tree, we can look at the leaves in the desired order by keeping a stack of trees. A loop to do this might look like (in CLU)

```
t: tree := tree of interest;
st: treestack := treestack$create ();
more: bool := true
while more do
   if t is a leaf
         then
            loop body
            if treestack$empty (st)
                  then more := false;
                  else t := treestack$pop (st);
                  end;
         else
            treestack$push (st, right subtree of t);
            t := left subtree of t;
         end;
   end;
```

Thus, the stack of trees is used to remember the subtrees the leaves of which have not been generated. Writing this code out for every loop is somewhat prone to errors and relies on many details. If the type *binary_tree* offered an iterator called *leaves*, we could write the above loop this way:

```
for l: leaf in binary_tree$leaves (t) do
   loop body
   end;
```

The variable *l* is called a *loop variable*, and is local to the for statement.

The for loop is more to the point, and depends on less detail than does the while loop. In short, iterators provide better abstraction. Iterators can also be more efficient than loops written out, because they can be operations of a cluster and thus have access to the representation of objects of the type. The definition of the iterator *leaves* might look like this:

```
leaves = iter (b: binary_tree) yields (leaf);
   if b is a leaf
         then yield (b);
         else
            for l: leaf in binary_tree$leaves (left subtree of b) do
                  yield (l);
                  end;
            for l: leaf in binary_tree$leaves (right subtree of b) do
                  yield (l);
                  end;
         end;
   end leaves;
```

The recursive iterator makes our intent more obvious, and shows a symmetry to the generation algorithm that was obscured in the while loop. There is a possibility that the recursive version is less efficient than the iterative version, but it is not immediately obvious, and may depend upon implementation details.

Let us turn to the semantics underlying CLU iterators. The basic actions involved in using an iterator in a for loop are:

(1) the for loop *calls* the iterator;

(2) the iterator *yields* objects to the body of the for loop;

(3) the loop body is executed with the loop variables set to the objects yielded by the iterator;

(4) the iterator is *resumed*, whereupon it either continues to yield items, or

(5) the iterator *returns*, terminating the loop.

Note that the loop body is executed once per yield, as the example would seem to imply. Also, the iterator is always resumed just after its last yield statement, with all local variables intact. Thus iterators are a form of coroutine. General coroutines require one stack per coroutine to run, but iterators are sufficiently restricted that they can be implemented with a single stack. (Iterators run in a single stack because of the particular pattern of resuming they use.) Let us detail the transformations made to the stack for each of the basic actions listed above.

(1) Iterator call – the arguments are passed and a new frame is set up for the iterator, just as in a procedure call;

(2) Yield – the loop variables, which are created at this point unless they are of fixed size, are set to the objects being yielded, the iterator's resume address and frame pointer are pushed on the stack, and the for loop body is entered with the environment reset to that of the iterator's caller; (notice that this is similar to a call even though yielding is semantically the same as returning);

(3) Loop body – the loop body executes normally, pushing any temporary information on the top of the stack, beyond the iterator's frame;

(4) Resume – the stack is popped back down to the iterator's frame, and execution of the iterator begins again at its resume address, with its environment restored;

(5) Iterator return – the iterator returns to its caller; execution continues after the for loop.

Thus, a yield is a kind of call, and a resume is a kind of return; both are a special case of coroutine resumption.

As the example demonstrates, iterators may contain for loops, even ones that call the iterator recursively. This is useful for walking recursive data structures. Although we did not show it, it should be clear that for loops can be nested with no difficulty.

Another feature we did not mention is that a for loop can be terminated in other ways than by the iterator returning. The loop body can execute a special statement called break, which terminates both the body and the iterator, continuing execution after the for loop. The body may also execute a return statement, which terminates the body, the iterator, and the routine with the for loop, all at once.

3.1.1. Implementing Iterators for ASBAL

The description above has been of iterators as they appear in CLU; here we will see what form iterators take in ASBAL. First of all, our call mechanism can be trivially extended to include calling iterators. Iterator returns are also trivial, being the same as procedure returns. Yielding is a little more complicated. Semantically a yield is like a return. However, it cannot be implemented as a return in ASBAL because returning in ASBAL always create new objects;

an iterator should be able to generate a sequence of existing objects, such as the elements of an array. This suggests that a yield statement should give a list of expressions, which will be evaluated to objects, as is done for selections:

```
yield (exp);
yield (exp1, exp2, ..., expn);
```

We also need a yields clause for iterator headers (the rest of the header can take the same form as procedure headers), which defines the order of the items and their types. It is useful for the iterator to control whether the objects it yields will be const or var in the loop body, so the yields clause includes that information as well. Here are some examples:

```
i = iter (const a, b: int) yields (const int);
```

or

```
i = iter (const f: foo, var b: bar) yields (var foobar, const char);
```

or even

```
i = iter (const count: int) yields ();
```

Now, let us consider the form of the for loop itself. The general form may as well follow CLU's. The only part that needs to be different from CLU is the declaration of the loop variables; the declaration will state whether the loop will use the yielded objects as const's or var's. Here are some examples:

```
for const x: int in ... do ... end for;
for var x,y: int in ... do ... end for;
for const x,y: int, var z: bool in ... do ... end for;
```

The forms of the break and return statements are:

```
break
```

and

```
return
```

3.1.2. Summary

We introduced the notion of an iterator as it appears in CLU, and we looked at the semantics of CLU iterators to help us design iterators for ASBAL. We then designed a form for iterator definitions and for loops in ASBAL. There will be more examples of iterator definition and use at the end of the chapter.

3.2. Exception Handling

In our earlier discussion of procedures we omitted one important aspect of procedural abstractions. A procedure, iterator, or selector might notify its caller of an unusual or error condition. We unify the terms *unusual* and *error* in the term *exception* (or *exceptional*), after Goodenough [Goodenough75]. In this section, we first examine CLU's exception handling mechanism,[1] and then proceed to modify it for ASBAL, in much the same spirit as we did with iterators in the previous section.

3.2.1. Exception Handling in CLU

Any procedure or iterator (we say *routine* for short) in CLU may *signal* exceptional conditions to its caller. The CLU viewpoint on the meaning of such *signals* is this: a module signals to indicate that it cannot perform its duty as a good abstraction. This might be due to an inconsistent state of an object, because of bad arguments, because of a hardware failure, or because of a system limitation (e.g., out of memory). Of course, it may less odiously indicate an unusual but predictable situation, such as *end_of_file*. The simplest semantic view of what a procedure does when it signals an exception is that it returns a different and distinguishable kind of object to its caller. Each exception the procedure might want to signal can be given a different name, and a list of objects can be sent along with the signal to further describe the condition.

The procedure's caller has the option of *handling* exceptions signalled by the procedure. If the caller has a handler for the exception, then it is executed, and execution continues after the statement to which the handler was attached. If the caller does not have a handler for the exception, then the caller signals a special exception called *failure*, sending along the string

---

1. We note that this aspect of CLU has been subject to change, so do not consider what we say here to be definitive about CLU. However, the essence of the mechanism is expected to remain the same. We trust that any further work with ASBAL would adopt any improvements made by the designers of CLU.

"uncaught exception: *name_of_original_exception*" [1]

Here is the format of a statement with a handler block attached:

```
statement;
   except
        when exception_name1: handler1;
        when exception_name2: handler2;
        ...
        when exception_namen: handlern;
        end;
```

A handler block handles only exceptions arising from invocations in the statement to which it is attached. Handler blocks may be nested, since a statement with a handler block is considered to be a statement. If more than one handler is available for an exception when handler blocks are nested, the innermost one takes precedence.

This exception handling mechanism is different from that of PL/1 [IBM70] in several ways:

(1) Handlers are textually associated with blocks of code, rather than being enabled by something like an on statement.

(2) Executing a signal statement *always* exits the current procedure, and it may not be resumed.

(3) The entire call stack is not searched for active handlers; rather, a procedure must be prepared to handle all signals that might come from any procedures it calls directly.[2]

Having handlers statically associated with blocks of code was chosen over dynamic mechanisms because it is cheaper and easier to implement, less confusing, and sufficient in all cases. Signalling always exits the current procedure because this is the only thing consistent with CLU's viewpoint that a signal indicates an inability to continue to perform as asked.

---

1. To help in debugging, if a procedure does not handle *failure*, then it signals *failure*, sending the same string as it received, instead of "uncaught exception: failure"

2. It is safe not to handle an exception only when it is certain that the invocation in question will not raise that exception. For example, in 'if z~=0 then x := y/z end if;' it is reasonable not to handle the exception for division by zero.

Thus a procedure is saying "I give up!" when it signals.

Let us go through an example. Suppose we have a type *queue* with an operation called *next* which returns the member at the front of the queue and removes it at the same time. Clearly, *queue$next* cannot work when applied to an empty queue.[1] Let us say *queue$next* can signal an exception *empty*. The definition of *queue$next* might look like (in CLU)

```
next = proc (q: queue) returns (element) signals (empty);
   if q is empty then signal empty; end;
   fix up q;
   return (old head of q);
   end next;
```

Thus, we see that a signals clause is required in the procedure header. Here are some examples of such clauses:

```
signals (foo, bar (int))
signals (bletch (int, int, bool))
```

The first one states that the procedure can signal two exceptions: *foo*, with no objects, and *bar* with an integer. (Factoring is disallowed here because it leads to ambiguity.) To send objects along with a signal, they are listed as in the yield statement:

```
signal bar (7);
signal foo (i + 5, z * 2, x > 5);
```

A call of *queue$next* and handling of the exception *empty* might look like this (again in CLU):

```
begin
   ...
   x := queue$next (q);
   ...
   end;
   except
         when foo: ...;
         when empty: ...;
         when bar: ...;
         end;
```

1. We are not considering parallel processing situations where such a request might hang until another process puts an item into the queue.

If a signal sends objects, the handler declares variables by which to name these objects.[1] The following example shows how this is done.

```
begin
   ...
end;
except
      when oh_foo (x,y: int, z: bool): body of handler;
      ...
      end;
```

In CLU, the semantics of sending objects with a signal is the same as that of returning objects.

If an iterator signals, the for loop that invoked it is terminated, and a handler executed just as if a procedure invocation had signalled. Naturally, the locus of the signal is the call of the iterator at the top of the for statement. For example:

```
for ... in iterate (x) do
   ...
end;
except
      when iterate_signal (stuff): handler;
      ...
      end;
```

If a signal statement is executed in the for loop body, the body, the iterator, and the routine containing the for loop are terminated all at once. Notice that this is different from the body's catching an exception, as in

```
for ... do
   ...
   statement;
         except
            when oh_foo: ...;
            ...
            end;
   ...
   end;
```

If *oh_foo* is signalled by some routine invoked in *statement*, then the handler will be executed

---

1. Actually, the handler may choose to ignore the objects entirely; the ambitious reader can peruse the syntax in the appendix.

and the execution of the body will continue. Assuming we have shown all handlers, if *oh_bar* is signalled by some routine called in *statement*, the body and the iterator will be exited, and a more global handler executed (if there is one).

3.2.2. Exception Handling in ASBAL

To transfer CLU's exception handling features to ASBAL, we need forms and semantics for signals clauses of routines' headers, signal statements, and handlers. Signalling is basically like returning, but the items sent along with the signal will probably not be handled the same way as return variables. For one thing, it is wasteful to allocate space for objects that might only be signalled once in a while. Another point is that these objects are always the initial values of some new variables and constants: those declared for the handler of the signal. The best approach to sending the objects appears to be to leave them on the top of the stack.

Unfortunately, the space at the top of the stack overlaps with the variables of the signalling routine. The objects will have to be computed first and then copied to that area. Unlike the case of returning, we will probably be willing to pay the price of copying items down onto the top of the caller's frame when they are to be signalled, since exceptions are generally rare compared to returns. This leads us to a signal statement like CLU's, except that ours always creates new objects, just as our returns do. Thus we write:

```
signal foo (10, b(3));
signal bar;
```

The signals clause in procedure and iterator headers gives a list of types, with no names, just as in CLU:

```
signals (foo, bar (int, array[bool]))
signals (bletch (int, int, bool))
```

Once the calling routine has the signalled objects at the top of the stack, the transfer to the handler is semantically a jump, but objects are sent to plug into the handler's variables. The handler's variable list will take the same form as that of a procedure header's argument list part. For example:

```
...
except when foo(const i: int, b: bool, var c: char); ...
    ...
end;
```

Thus the sequence of actions for a signal in ASBAL is as follows:

(1) the expressions in the signal statement (if any) are computed, leaving objects in temporary variables of the signalling procedure;

(2) a run-time system routine examines a data base detailing the handlers of the calling routine;

(3) it proceeds to adjust the stack appropriately and copies down the signalled objects (using bit-copies); then

(4) it makes the handler's variables contain the objects and transfers to the handler.[1]

This is a bit complex, but the run-time system routine is written only once so it is not too painful. The information that must be kept around is not all that bad either. Basically it consists of which range of addresses in the procedure corresponds to which handler block, a list of the exceptions handled, and the addresses for each handler block. If the information is ordered correctly, the run-time routine need only find the first handler in a linear search of a table. The copying of objects does not lead to problems because space either already exists at fixed offset slots for the objects, or space at a fixed offset was saved for a pointer. The objects that really go on top of the stack can be put there in any order since they are referred to through pointers, and popped off all at once because they all have the same scope. Thus, there are no overwhelming implementation problems for our proposed exception handling mechanism, though it does of course incur some overhead.

---

1. This last adjustment may be done as part of the bit-copy of the previous step (i.e. copying into space previously allocated for the handler variables), or may be the setting of a pointer in a fixed offset slot to point to the signalled object. (Thus the pointer provides the address of the handler variable.)

3.2.3. Summary

We have examined CLU's exception handling mechanism in detail. Based on this examination, we designed parts of ASBAL to perform the same function: the structured notification and handling of exceptions. Fortunately, few changes were needed in the mechanism borrowed from CLU, and little additional mechanism was required. Again we feel we have been successful in transferring a good feature from CLU to ASBAL.

3.3. An Example - Sorted Bags of Strings

This section presents another data type definition: a sorted bag of strings. This data type might be used for computing the frequency of occurrence of different words in a sample of text, and printing them out in alphabetical order. (Our presentation is based on the example in [Liskov77]). Here is a description of the operations:

create: proctype() returns(bag);
(create a new empty bag)

insert: proctype(var bag, const string[;20]) signals (full)
(insert the string into the bag; signal full if there is no more room)

count: proctype(const bag) returns(int)
(total number of items in the bag, counting repetitions)

size: proctype(const bag) returns(int)
(total number of distinct items in the bag, i.e., not counting repetitions)

increasing: itertype(const bag) yields(const string[;20], int)
(generate each distinct string in the bag, with its repetition count, in alphabetical order)

The type string in ASBAL is a sequence of characters. Of course, string variables must put a limit on the maximum size string they can store. That is the reason for the parameter '20' to the string types above. (The ';' in 'string[;20]' will be explained in the next chapter.) A string is different from an array of characters in that its contents cannot be changed, i.e., strings are immutable. Strings are whole values even though their individual characters can be accessed. The usual operations on strings, such as substring and index, are provided in ASBAL. A full

list of string operations is in Appendix II.

3.3.1. The Representation

The representation we will use for bags is a binary tree. Each node will contain a string with a count of the number of times that string has been inserted in the bag. All nodes to left of a given node contain strings which alphabetically precede the string associated with the given node. Of course we need to keep a count of the number of items in the entire tree in order to compute size and count efficiently. This "top level rep" is then something like this:

```
record [count: int,
        size:  int,
        t:     tree]
```

We will maintain the tree in an array, using array indexes as "pointers" to the subtrees in the nodes. (We must put a limit on the number of distinct items in a bag. We will use 500 in this example.) This adds stuff to the rep type. (To be really clean about it we would define the tree part as another type, but we desired to keep the example short.) The result is:

```
rep = record [count: int,
              size:  int,
              root:  mbranch,
              tree:  an];

an = array [node; 1, 500];

node = record [s:      string[;20],
               left:   mbranch,
               right:  mbranch,
               number: int];

mbranch = oneof [empty:  null,
                 branch: int];
```

The type *mbranch* is short for "maybe branch": it is either empty, or designates a branch, by an index into the array. A oneof type is a discriminated union, somewhat like the variant records of Pascal [Wirth71]. A oneof object is a *tag* (one of the field names) along with an object of the corresponding type. (There are operations that convert an object of some type to a oneof object with an appropriate tag. They and the tagcase statement will be presented below.)

Allocating space for a oneof variable is easy: just allocate the maximum of the sizes of the various possible types in its fields, plus room for the tag.

3.3.2. The Operations

Let us start with the create operation for bags. We must set all the counts to 0, and initialize the array.

```
create = proc () returns (b: cvt);
   none: mbranch := mbranch.make_empty (null);
   n: node := node${s:      "",
                   left:    none,
                   right:   none
                   number: 0};
   b := rep${count: 0,
             size:  0,
             root:  none,
             tree:  an$fill (n, 1, 500)};
   end create;
```

The "" means the empty (or null) string. The *create* operation shows how to make a oneof value from a non-tagged value of the right type: use the 'make_tag' operation, in this case the *make_empty* operation. (This operation calls the *copy* operation of the type.)

Here is the insert operation on bags:

```
insert = proc (var b: cvt, const s: string[:20]) signals (full);
   b.root := insert1 (b, s, b.root) except when full: signal full; end;
   end insert;

insert1 = proc (var b: rep, const s: string[:20], root: mbranch) returns (m: mbranch) signals (full
   tagcase root in
      tag empty:
         m := add_node (b, s);
```

```
            tag branch (const i: int):
               m := root;
               with var n == b.tree(i) do
                     if s = n.s
                        then
                              n.number := n.number + 1;
                              b.count := b.count + 1;
                     elseif s < n.s
                        then n.left := insert1 (b, s, n.left);
                        else n.right := insert1 (b, s, n.right);
                        end if;
               end with;
            end tagcase;
            except when full: signal full; end;
         end insert1;

      add_node = proc (var b: rep, const s: string[,20]) returns (br: mbranch) signals (full);
         if b.size = 500 then signal full; end if;
         b.size := b.size + 1;
         b.count := b.count + 1;
         none: mbranch := mbranch$make_empty (nil);
         b.tree(b.size) := node${s:      s,
                                number: 1,
                                left:   none,
                                right:  none};
         br := mbranch$make_branch (b.size);
         end add_node;
```

This operation illustrates the use of internal procedures (that is, procedures not exported by a cluster); it also demonstrates how to use the tagcase statement to discriminate with a oneof object. Each case starts with 'tag *tag*' and allows a name to be given to the object so that the name has the discriminated type. This is the *only* way an object in a oneof can be mutated. We also see a real use of exception handling and signalling, although not a very fancy one.

The count and size operations are easy to write:

```
count = proc (b: cvt) returns (c: int);
    c := b.count;
    end count;

size = proc (b: cvt) returns (s: int);
    s := b.size;
    end size;
```

The last operation to write is the iterator *increasing*:

```
increasing = iter (const b: cvt) yields (const string, int);
    for const s: string, i: int in increasing1 (b, b.root) do
        yield (s, i);
        end for;
    end increasing;

increasing1 = iter (const b: rep, br: mbranch) yields (const string, int);
    tagcase br in
        tag empty:
        tag branch (i: int):
            with const node == b.tree(i) do
                for const s: string, j: int in increasing1 (b, node.left) do
                    yield (s, j);
                    end for;
                yield (node.s, node.number);
                for const s: string, j: int in increasing1 (b, node.right) do
                    yield (s, j);
                    end for;
            end with;
        end tagcase;
    end increasing1;
```

Again we see a recursive internal operation and use of the tagcase statement. At the top level our entire type definition looks like this:

```
bag = cluster is create, insert, count, size, increasing;
    rep = ... ;
    ...
    create = ... ;
    insert = ... ;
    insert1 = ... ;
    add_node = ... ;
    count = ... ;
    size = ... ;
    increasing = ... ;
    increasing1 = ... ;
    end bag;
```

Here are some example uses of the bag abstraction (the '/' means division):

```
b: bag := bag$create ();

bag$insert (b, "a string"); except when full: signal nh_fog; end;

avg: int := bag$count (b) / bag$size (b);

n: int := bag$count (b);
for const s: string, i: int in bag$increasing (b) do
    print (s, i, i/n);
    end for;
```

3.4. Summary

This chapter has presented iterators and exception handling for ASBAL. These two features were borrowed with little change from CLU, and the transfer to ASBAL was quite successful. All the central features of ASBAL have now been presented, coming mainly from CLU with alterations to accomodate our object interpretation of variables. The next two chapters consider two additions to the language. The first is parameterization of abstractions. We will explore adding CLU's parameter mechanism to ASBAL, and will end up augmenting it with an original feature for handling types with objects of dynamically varying size. The following chapter investigates adding limited list-processing capabilities to the language. The features designed allow the construction of general graph structures without garbage collection - in the stack.

4. Parameters

This chapter presents the ASBAL mechanism for parameterizing abstractions. We begin with an examination of parameters in CLU. We then borrow and extend CLU's mechanism, modifying it to suit our needs. The major extension made is for parameters relating to the sizes of objects in ASBAL. We have several goals in extending CLU's mechanism for ASBAL:

(1) to make programs as independent of the sizes of their data objects as possible, and to allow sizes to be determined at run-time;
(2) to relieve the programmer of the burden of keeping track of the sizes of variables, and to transfer this burden to the compiler and run-time system; but,
(3) to allow the programmer ultimate control over the sizes of variables.

After presenting our parameter mechanism, we give an extended example using it. We close the chapter with a discussion of possible implementation techniques.

4.1. Parameters in CLU

Here we discuss the parameter mechanism used in CLU. We start with the simplest and most strongly motivated case – parameters to clusters. We present a full example of a parameterized cluster, and then move on to parameterizing other abstractions.

4.1.1. Parameters to Type Definitions

Let us say we have written a cluster to implement queues of integers. A while later we find a need for queues of strings, so we write a new cluster to implement them, borrowing from the previous cluster. Some more time passes, and we find we need queues of customers for a simulation program, so we again adapt the queue-of-integers cluster. This copying and modification could go on forever. What is worse, if some subtle bug is found in the original cluster, a lot of effort is necessary to find and correct all the other clusters that copied its code.

One might imagine using a fancy text editor or macro processor to help in this

correction and updating process. However, we can do much better if we use the idea of an abstraction generator: a parameterized module providing one abstraction per set of parameters. For example, we would like to write a definition of queues using a dummy name for the type, and to allow any type to be filled in later to get the kind of queue required. This groups the information for all classes of queues together, so updates affect all versions, etc. A parameterized type definition is the implementation of an abstraction generator called a *type generator*, since the abstractions generated are types. It will be easier to explain the semantics with an example (in CLU):

```
queue = cluster [t: type] is create, enq, deq, empty;

    rep = array[t];

    create = proc () returns (cvt);
        return (rep$new());
        end create;

    enq = proc (q: cvt, x: t);
        rep$addh (q, x);
        end enq;

    deq = proc (q: cvt) returns (t) signals (empty);
        if rep$size (q) = 0
          then signal empty;
          else return (rep$remt(q));
          end;
        end deq;

    empty = proc (q: cvt) returns (bool);
        return (rep$size(q) = 0);
        end empty;

    end queue;
```

The first thing to notice is the '[t: type]' after cluster, which signifies that *queue* takes a single parameter, called *t*, and that the actual parameter[1] must be a type. Thus, for every type (*foo*, say) there exists a queue of that type (written *queue[foo]*). Even *queue[queue[queue[int]]]* is

1. The term *actual* is in contrast to *formal*. These terms indicate the usual distinction between a template (*formal*), and an instance of it (*actual*).

legal because *queue*[int] is a type, so it is a legal parameter to *queue*, etc. The representation is chosen to be array[*t*] - this demonstrates that *t* is interpreted as if it were an actual type specification[1] inside the cluster definition. The *create* operation simply returns an empty array (representing an empty queue). The *enq* operation adds a new element to the high end of the array. Notice that *t* by itself is a valid type specification in the header of the *enq* operation. It is also legal to declare variables to be of type *t* inside the cluster; we mention this to drive home the point that *t* really is taken to be a type specification within the definition of *queue*. The *deq* operation is symmetrical to *enq*, except that it may signal empty, indicating that its caller tried to remove an element from an empty queue. The *empty* operation is just a test to see if a queue has no members.

4.1.2. Restrictions

In order to demonstrate further features, we will add some new operations to the *queue* cluster. One nice operation to have is *copy*. We would like *copy* to call *t*$*copy* on each element of the queue. Of course, this means that we can only copy *queue*[*t*] if *t* has a *copy* operation (which it need not have). For this reason *restrictions* were added to CLU. A restriction defines a set of types possessing certain operations with particular functionalities. Restrictions are used to limit the legal actual type parameters, and they insure that each actual type parameter has the specified operations. Let us look at *queue*$*copy* for an example:

```
copy = proc (q: cvt) returns (cvt);
       where t has copy: proctype(t) returns(t) end;
   return (rep$copy(q));
   end copy;
```

The call of array[*t*]$*copy* (implicit in rep$*copy*) results in calls of *t*$*copy*; since array[*t*]$*copy* requires a *copy* operation of *t*, we must require that operation of our caller. Restrictions complicate type checking, but are necessary. The where clause can also require a parameter to have several operations, and can put restrictions on any number of type parameters. The keywords proctype, itertype, and oneof are used to declare procedure, iterator, and selector

---

1. A type specification is the syntactic description of a type.

types. (The keywords proc, iter, and selector are not used for this purpose because syntactic ambiguities result.)

The above example puts a restriction on a single *queue* operation. If *t* does not have a *copy* operation then all the other operations of *queue[t]* can be used – just not *queue[t]$copy*. In some cases it is desirable to put a restriction on all the operations. For example, consider extending our sorted bag abstraction of the previous chapter to a type generator. That is, we define a type generator *sorted_bag*, which can be instantiated to produce sorted bags of any ordered type. In terms of operations, the type parameter is required to provide a less than (*lt*) and an equal (*equal*) operation.[1] Such a restriction is stated like this:

```
sorted_bag = cluster [t: type] is ... ;
        where t has lt, equal: proctype(t,t) returns(bool) end;
    ...
    end sorted_bag;
```

We can still put further restrictions on the type parameter within individual operations if needed. Thus, a *copy* operation for the sorted bag cluster would require *t* to have a *copy* operation.

4.1.3. Parameters to Procedures and Iterators

Just as clusters can be parameterized, so can procedures and iterators. Consider a bubble sort routine that takes an array of any appropriate type and sorts it. The same reasoning that lead to cluster parameters is effective here. Here is the procedure header for the sort routine:

---

1. We would really like to say that *lt* and *equal* order the objects of type *t*, but all we can require in a restriction is the current functionality. Naturally, the fact that *lt* orders the objects would be included in the specifications of the sorted bag abstraction, but we do not expect a compiler to check such specifications, and so do not include them in the source text.

```
sort = proc [t: type] (a: at);
        where t has equal, lt: proctype(t,t) returns(bool) end;
    at = array[t];
    ...
    end sort;
```

Operations of a type may be parameterized just as regular routines can; this leads to the following general form for operation specifications:

type_name[parameters to type]$operation_name[parameters to operation]

4.1.4. Other Kinds of Parameters

In CLU, most compile-time constants are allowed as parameters. This includes integers, characters, strings, reals, booleans, and nulls. Not all of these are very useful (there is only one value of type null, so null is useless as a parameter type). Every distinct set of parameters to a parameterized abstraction results in a distinct abstraction. This means that *queue*[int] is different from *queue*[bool], etc. Also, if we are given the definition

```
foo = cluster [x, y: int] ... ;
```

then *foo*[1,2] is different from *foo*[2,2]. In like fashion, different sets of parameters to procedures and iterators produce different procedures and iterators.

There is a goal that type checking be possible at compile-time, which requires instantiation to be possible at compile-time. Therefore, parameters may not be computed at run-time. However, it turns out that even if all parameters are compile-time knowable, instantiation is not always possible at compile-time. This difficulty will be discussed in the section on implementation. Still, run-time computed expressions are not allowed as parameters.

4.2. Parameters for ASBAL

ASBAL can borrow all of CLU's parameter mechanism with no significant changes. However, even though that mechanism works fine, it is not convenient for what will be the most widespread use of parameters in ASBAL: sizes. To handle sizes reasonably we must allow size parameters to be computed at run-time. This extension can be made without too much trouble, but it is not sufficient. Using CLU's mechanism for sizes will still be inconvenient

because every size must be specified explicitly, and each set of size parameters will result in a distinct type. This results in a distinct set of cluster operations for each size (although most of the physical code for the operations can be shared among all instances of the type generator). The major difficulty is that binary (and higher order) operations on objects of different sizes become hard to express, because a cluster may convert only objects of its own specific type to and from the representation. Therefore, there is no way to access the representations of objects of different sizes simultaneously, because objects of different sizes are of different types.

With the proliferation of parameters, expressions become quite complicated. Consider strings as an example. We could not write

```
s := s || t;
```

(The '||' is a sugar for *concat*.) We would be forced to say

```
s := string[100]$copy(string[100]$concat[50](s, t));
```

to get the types to match if *s* had size 100 and *t* had size 50. Just imagine how obscure this statement would look if written out like the one above:

```
s := s || string$substr(t, i, j) || string$rest(u, k);
```

Of course it is possible to extend the notation for infix operators (e.g., ||[100,50]), but the information still obscures the computation.

Having each set of size parameters define a different type (or procedure, etc.) separates types too finely. First of all, it conflicts with abstraction. The objects of many types come in a variety of sizes, in many cases infinite. *Variables* have fixed sizes (because they correspond to storage allocated in the stack) -- *objects* are conceptually of unbounded size. For example, there are strings of any length greater than or equal to zero. Size is not part of the conceptual type of objects, but size information must somehow be provided for allocating storage for variables. If we require every abstraction to be bounded, we are putting an artificial restriction on the abstractions just to make the implementation work out. One way to resolve this conflict is to consider objects to be unbounded, and variables to be imperfect models of the objects they contain. This leads to attributing size bounds only to variables. The effect is that variables cannot hold all objects of their type, but only the ones that will fit in the variable.

In sum, size will be declared only for variables. We find that the most convenient way to state the size information is as part of the type specifications (typespecs) for variables. Our

task is to design convenient syntactic forms for expressing size information where it is appropriate, and to allow such information to be omitted where it is not necessary. The exact technique is to introduce a new class of parameters to types, *size parameters*. These parameters are distinguished from CLU-style parameters (which we call *regular parameters*) by being listed after a ';' in the parameter list. Size parameters are used only with types; routines take only regular parameters. Also, size parameters are always integers; no other types seem useful enough to justify the additional mechanism their incorporation would require.

Two examples of size parameters have already been used in previous chapters. Array takes two size parameters, indicating the minimum lower bound and maximum upper bound of objects storable in an array variable; string takes one size parameter, indicating the maximum length object a string variable can hold. Arrays and strings are the only basic types of varying size; all other types of varying size incorporate them in their representation, although possibly through many levels of data abstraction.[1] The implementations of both arrays and strings insure that objects too large for a variable of their type to hold are not assigned to the variable. Attempts to make such illegal assignments cause an exception, *failure*("*variable overflow*"), to be signalled. Furthermore, the implementation of arrays insures that the objects in array variables are not grown beyond the limits of their containing variables; if such an attempt is made, the variable overflow exception is signalled. To make such exceptions avoidable, we will provide a mechanism for querying the size parameters of a variable. This mechanism can be used to check sizes before assignments or growing operations.[2]

4.3. The Size Parameter Mechanism

Having introduced some of the basic concepts and features of size parameters, we now go into detail about their use. This is more easily done by going through the syntactic forms used for specifying size in typespecs, and the restrictions imposed on which forms may be used with typespecs in different positions.

---

1. That arrays and strings are the only sources of objects of different sizes is similar to the fact that all mutation is accomplished via records and arrays.

2. Of course, one can just attempt the operation and then handle the exception, but it is often better style to prevent the exception's occurrence.

4.3.1. Size Specifiers

A *size specifier* (sizespec) is the syntactic form representing a size parameter. There are three forms of sizespecs. First we have the *exact sizespec*, which is an expression evaluating to an integer. In some situations the expression is further restricted (e.g., to be compile-time known), but it can usually be any run-time-computable expression.

The next form of sizespec is '*', and it is called a **-sizespec*. A star is used as a sizespec to indicate that any value is allowed for that size parameter, or that the size is immaterial. For example, a routine may take a string as an argument without needing to know or restrict the size of string variable in which the string object happens to be stored. In fact, we expect the size of arguments to be irrelevant to most routines.

The other form of sizespec is '?*id*', where *id* is an identifier.[1] These sizespecs are called *?-sizespecs*. The ?-sizespec form is equivalent to '*', except that a ?-sizespec permits the size of a particular variable to be queried. For example, a procedure *p* may be written to take one argument, an array. To be flexible, *p* will accept an array of any size, but it must know the size so as not to cause an overflow in growing the array. The code for *p* might look like

```
p = proc (var a: array[int]; ?low, ?high));
    ...
    if x > a?high then ...
    ...
    end p;
```

The expression *a?high* evaluates to the second size parameter of the actual argument to *p* at run-time. (The result of *a?high* is not necessarily the same as *array[int]$high(a)*; the first is the size of the variable, and the second is the current high bound of the array object in the variable.)

---

1. This notation was suggested by the use of '?' in Alphard [Wulf76a].

4.3.2. The Kinds of Typespecs

There are three forms of typespecs in ASBAL. The first form is called the *variable typespec* (v-typespec) because it is used mainly in variable declarations. All the sizespecs of a v-typespec must be exact sizespecs, so that the actual space required for a variable can be computed and allocated. We will detail all places where each form of typespec is used below. Here are examples of v-typespecs:

```
string[;15]
string[;u(x)+v(x)]
array[int; 1, 100]
array[int; 1, 10*j+5]
array[int; f(x), 3]
array[int; foo(x, y), bar(y, z)+2]
```

The second form of typespec allows exact or *-sizespecs to be used, and is called a *v*-typespec* (for variable or * typespec) for short. It is used where any size is allowed or size is irrelevant, but where querying is not allowed. V-typespecs are a subset of v*-typespecs; here are some v*-typespecs that are not v-typespecs:

```
string[;*]
array[int; *, 10]
array[int; *, v(x)]
array[int; 1, *]
array[int; u(x), *]
array[int; *, *]
```

We allow an abbreviation for typespecs all of whose sizespecs are '*': the size parameter part of the parameter list may be omitted, including the ';'. Furthermore, if such omission results in '[]', the brackets can be dropped as well. Hence

```
array[int; *, *] and string[;*]
        become
array[int] and string
```

respectively.

The third form of typespec is the most general: any of the three sizespecs may be used in it. This form is called the *v*?-typespec* (for variable, *, or ?id typespec). V*-typespecs are a subset of v*?-typespecs (and hence v-typespecs are also a subset of v*?-typespecs); here are

some v*?-typespecs that are not v*-typespecs:

```
string[;?len]
array[int; 1, ?high]
array[int; ?low, *]
array[int; ?low, ?high]
```

(There are many more combinations of legal sizespecs in v*?-typespecs for arrays.)

4.3.3. How Type Specifications are Used

Now we discuss which form of typespec is used in each syntactic position, and the meaning attached to it in that position. We will do this separately for different groups of syntactic positions.

4.3.3.1. Arguments to Routines

The typespecs for arguments to routines are v*?-typespecs, so that routines can handle objects of any size conveniently. If a size parameter of an argument type is specified exactly (and in this case only a compile-time constant[1] is allowed), then that size parameter must match that of the actual argument exactly. In the case of *- and ?-sizespecs, any size parameter is acceptable. The use of a ?-sizespec allows that size parameter to be queried. Let us consider an example:

```
p = proc (var a, b: array[int; 1, ?high]);
    ...
    end p;
```

In this case, both *a* and *b* must be arrays of integers with a lower bound of 1.[2] Their upper bounds are not restricted, and need not be the same. The actual upper bounds can be obtained via the expressions *a*?*high* and *b*?*high*.[3] As another example use of argument typespecs, consider the following program which adds the elements of one array to the end of the another:

---

1. However, side-effect free expressions involving parameters to routines, and using only built-in operations, are considered to be compile-time constants.
2. These bounds are the bounds of the *variables*, not the current bounds of the array *objects*.
3. '?' should be thought of as a special binary operator, similar to '!' in selections.

```
p = proc[t:type](var a1: at, const a2: at) signals(overflow());
        where t has copy: proctype(const t) returns(t);
    at = array[t; *, ?high];
    if (a1?high - at$high(a1)) < at$size(a2)
        then signal overflow;
        end if;
    for const x: t in at$elements(a2) do
        at$addh(a1, x);
        end for;
    end p;
```

The test in the if statement is: "Does *a1* have enough room for a copy of each element of *a2*?" The *elements* iterator for arrays produces each element in the array from the lowest to the highest.

4.3.3.2. Return Variables

Arguments are the most obvious and strongly motivated use for size parameters because size parameters in arguments allow single procedures to handle objects of any size conveniently. However, there are also some situations where flexibility in the size of objects returned by a procedure is helpful. For this reason we allow the size parameters of return variables to be determined dynamically; specifically, these size parameters may be computed from the arguments to the procedure being called. For comprehensibility of ASBAL programs, we require that any size computation for return variables not mutate arguments of the procedure being called. This is done by permitting only const uses of the arguments in these size expressions.

Consider a procedure that appends the contents of two arrays together to form a new array. If it is known that the new array will never be enlarged, it is reasonable to create the smallest possible array that will contain the desired result, so as to avoid any wasted storage. Here is the skeleton of such a procedure:

```
q = proc (const a1, a2: aint) returns (a3: array[int;1,aint$size(a1)+aint$size(a2)]);
    aint = array[int];
    ...
    end q;
```

Determining the size of return variables on the fly has some complications, however.

Recall that return variables are really specifications for variables passed in by the caller. If the variable passed in is a temporary, then there is no problem because the mechanisms presented in Chapter 2 allow for determination of size temporaries after computation of arguments to the invocation "creating" the temporaries. In fact, that mechanism was designed with flexible return variables in mind. On the other hand, if the variable passed in as the return variable is not a temporary we may have a conflict of size. A check must be done, often at run-time, to compare the size of the variable being passed in with the size declared in the procedure header.

At this juncture we have an option: we may require that the sizes match exactly, or just that the variable passed in is at least as big as the one we would get from the return variable specification. We have chosen to be flexible and allow any variable of sufficient size. We delay discussion of the basis for this decision until the entire size parameter mechanism has been presented.

Two questions remain: what do we do if the size of a return variable fails the run-time check outlined above? Our solution to this problem is to have the invocation being attempted signal *failure("variable overflow")*. The other question is: what do we do if a return variable size computation signals? To this problem we have a solution similar to the one above - signal *failure("size expression signalled")*.

Because of the run-time checks often required, flexible return variables may be expensive. However, we believe that the common uses of flexible return variables will be handled at compile-time. The reason why compile-time checks will often suffice is that most types taking size parameters have sizes which vary; if a return variable of such a type were constructed using the minimal amount memory, it could not be grown thereafter. Therefore, we believe it will be more common for the user to specify the size of the variable to be returned by passing an argument (or perhaps a parameter) to the procedure, rather than having the procedure compute the size itself. We believe that the expressions used to convey the size information may be comparable at compile-time even if they are symbolic; that is, it may be possible to perform the checks even if the size is a parameter or an argument of the procedure making the call. Here is an example:

```
p = proc (const n: int, ... ) ... ;
    ...
    var x: foo[;n];
    ...
    x := q (n, ... );
    ...
    end p;

q = proc (const i: int, ... ) returns (a: foo[;i]);
    ...
    end q;
```

We grant that it may not be at all easy to design a compiler "smart" enough to perform this sort of optimization - we are merely pointing out that the optimization may be possible in many cases.

4.3.3.3. Declarations

There are two sorts of declarations: those with initialization and those without. A declaration without initialization must use a v-typespec so that storage can be allocated for the variable being declared. Any expression evaluating to an integer is allowed for computing the size parameters.

Declarations with initialization are more complicated, because we have the opportunity to reduce storage requirements: we can permit the variable to be the exact size returned by the procedure being invoked to initialize the variable. Thus we allow *- and ?-sizespecs in the typespecs for declarations with initialization. Any parameter specified by a *- or ?-sizespec takes the value computed by the invocation for its return variable; any exact sizespecs in the declaration with initialization are kept as is. Therefore, the normal check necessary for flexible return variables in assignment may have to be performed in declarations with initialization as well; the check can be omitted if all sizespecs are *- or ?-sizespecs. Constant definitions follow the same scheme as declarations with initialization.

Here are some examples of declarations and constant definitions:

```
var n: int := 100;
var a: array[int; 1, n];
var b: array[int] := array[int]$fill (0, 1, 50);
var c: array[int; 1, ?high] := foo();
const d: array[int] := b;
const e: array[int; 1, 100] := c;
```

The low and high bounds of *a* will be 1 and 100; those of *b*, 1 and 50. The low bound of the array returned by *foo* must be one, but the high bound may be anything, and can be queried by writing *c?high*. The bounds of *d* will be 1 and 50, just like *b*. The definition of *e* will fail unless *c* has at most 100 elements.

4.3.3.4. Representation Types

The typespec for the representation type (rep type) of a cluster must be a v-typespec so variables of the type being defined can be allocated. Clearly all size parameters in the rep typespec must be determinable given all parameters to the abstract type. However, arbitrary expressions are allowed in computing the size parameters for the rep type from the parameters of the abstract type. As with return variables, these expressions must be side-effect free, because they may be evaluated to compute the size of a return variable. Also, if an exception is signalled when a rep's size expression is evaluated, *failure*("*size expression signalled*") is signalled to the creator of the variable.

The header for a cluster with size parameters takes this form:

$$id_0 = \text{cluster [} \textit{regular\_parameter\_list} \text{ ; } id_1, \ldots, id_n \text{] is } \ldots;$$

The id_i for $i > 0$ are the names of the size parameters. These names are only used in defining the rep type and other equates; unlike regular parameters, they are not available to the cluster operations as specific values. Furthermore, types and expression using these names are not available to the operations. This is because the values associated with the abstract size parameter names are not per cluster instantiation, but rather per object; hence it does not make sense to use those names in cluster operations.

Let us introduce an example we will use throughout our discussion of size parameters and rep types. Assume for the moment that ASBAL does not have strings, and we need to implement them using arrays of characters. Here is a skeleton of part of the string cluster:

```
string = cluster[;len] is ... ;
   rep = array[char; 1, len];
   size = proc (const s: cvt) returns (n: int);
        n := rep$size(s);
        end size;
   ...
   end string;
```

Notice that the *size* operation returns the size of the *object*, not the size of the variable.

Now we come to the question of what *s*- and ?-sizespecs mean when written in typespecs for the abstract type. For example, what does *string*[*s*] or *string*[?*i*] mean? They merely mean that those abstract size parameters are not being specified, and in the case of ?-sizespecs, that those abstract size parameters may be queried, e.g. *x*?*i* if *x* were declared as *string*[?*i*]. In every case where the size of the rep must be known, all abstract size parameters will be available, so the rep size parameters can be computed and space allocated.

The only potential confusion left is the meaning of the typespecs rep and cvt. Rep takes the size parameters of the abstract type; the meaning is "the rep typespec obtained from giving the abstract type those size parameters". As in CLU, cvt is just a shorthand for the abstract or rep typespec at the interface of a routine, with a conversion applied at the appropriate time: down for incoming objects, and up for outgoing ones. Therefore, cvt takes the same size parameters as the abstract and rep typespecs do. Notice that neither rep nor cvt requires statement of regular type parameters; the regular type parameters are implicit in the instantiation of the cluster. The conversions up and down have the same semantics and implementation as in CLU - they cause little or no run-time action, but are used merely to change the compiler's "point of view" on the type of an object.

To illustrate the use of cvt, consider the procedure below as part of our example string cluster:

```
concat = proc (const s1, s2: cvt) returns (s3: cvt[;rep$size(s1)+rep$size(s2)]);
   ...
   end concat;
```

Notice that the arguments (*s1* and *s2*) have been down'ed before the computation of the size of *s3*. If we had occasion to create a temporary variable of type rep in the string cluster, we might write

```
var x: rep[;n] ... ;
```

which would be equivalent to

```
var x: array[char; 1, n];
```

except that the latter cannot be up'ed. It cannot be up'ed because doing so involves inversion of arbitrary functions in the general case; this is so because there are no restrictions on the way in which the rep type depends on the abstract type's size parameters. However, any array[char] can be assigned to a rep variable (provided the sizes match; hence, it is always possible to up a copy of an object if the types match, ignoring size parameters.

If the sizes of the string variables input to the *concat* operation were needed in the operations procedure body, the header could have started as

```
concat = proc (const s1, s2: cvt[;length]) ... ;
```

The lengths of *s1* and *s2* could then be obtained by *s1length* and *s2length*, respectively.

4.3.3.5. Other Positions in Routine Headers

There are a few other positions in routine headers that require typespecs. First, there are the types of parameters to routines; however, these are all simple typespecs such as int, char, bool, and type, which have no size parameters, so there is no problem. The typespecs for objects yielded by an iterator are vs-typespecs: a size is explicitly given and enforced, or any size is allowed, but there is no use for ?-sizespecs here. The same arguments hold for the types of objects signalled by routines, and the of-type in selector headers. Here are some examples of clauses from routine headers:

```
... signals (foo(array[int;1,*]), bar(string)) ...
... yields (string, array[char;1,2]) ...
... of string ...
bletch = proc[x: int] ...
ralph = iter[i: int] ...
edgar = selector[flag: bool] ...
```

4.3.3.6. Types of Routines

A situation slightly different from routine headers is the expression of typespecs for the arguments, etc., in the typespecs of routines (i.e., proctype's, itertype's, and seltype's). The typespecs for routine types should allow full type checking, so typespecs for arguments, returns, yields, etc., must all be given. The argument typespecs are v*-typespecs, there being no use for ?-sizespecs in that position. (A routine accepts either a particular size, or any size.) Likewise, return variables are given v*-typespecs, but computed sizes are given as *'s, not expressions: only compile-time expressions are allowed so that as much type checking can be done at compile-time as possible. Thus the type of string$concat is written as

```
proctype (const string, string) returns (string)
```

which is short for

```
proctype (const string[*], string[*]) returns (string[*])
```

Yields, signals, and of- typespecs are all handled just like return variable typespecs. That takes care of all the special items in typespecs of routines. Here are some more example routine typespecs:

```
proctype(var array[bool;1,*], const string[;10])
itertype(const array[int]) yields (string[;15])
seltype() of string from string
```

4.3.3.7. Actual Type Parameters

Now we come to the writing of typespecs for actual type parameters of abstractions, e.g., the *t* in array[*t*]. These are always v-typespecs (with one exception), so that variables of the type can be declared. The exception is the type parameter to ptr. The type ptr has to do with pointers, which are discussed in the next chapter, however, let us explain here how ptr is different. The type generator ptr is used for typed pointers, and it takes as a parameter the type of object pointed to. Since the size of a pointer is independent of the size of the object pointed to, *- and ?-sizespecs are allowed in typespecs used as parameters to ptr. After reading Chapter 5 it should be clear why this will work.

Here are some examples of typespecs used as parameters:

```
array[array[int;1,100]]
record[a, b: string[;20]]
ptr[a, array[int;1,?high]]
ptr[a, record[a, b: string]]
```

Please ignore the first parameter to ptr for now; the second parameter is the one discussed above.

4.3.3.8. Operation Names

There is one last position where typespecs are needed: in the names of cluster operations. In this position all kinds of typespecs are allowed, since size parameters are completely irrelevant. However, it is common to write the short forms of v*-typespecs in cluster operations, omitting the size parameter part completely. This gives programs a nicer appearance, but is not essential. It is also common to use a short name that is equated to a v*?-typespec; for example, *at* for array[t]. All of these are legal names for the *concat* operation on strings:

```
string[;20]$concat
string[;*]$concat
string[;?len]$concat
string$concat
```

(This operation also has an infix form: '||'.) In clusters, rep is also legal as a type for forming operation names.

4.4. An Example Cluster - Sequences

The header for the sequence cluster is

```
seq = cluster [t: type; n] is null, addh, addl, concat, remh, reml, trim,
                              first, last, fetch, elements, copy, equal, length;
        where t has copy: proctype (const t) returns (t) end;
```

Sequences have many of the same operations as arrays, except that sequences are not mutable (their state cannot be changed). It is easiest to explain how sequences work by presenting the operations a few at a time. But first, the representation:

```
rep = array[t; 1, n];
```

Thus, sequences will be modelled by arrays. This is convenient because they are similar in many respects.

```
null = proc () returns (s: cvt[,0]);
    s := rep$create (1);
    end null;
```

The array create operation returns an empty array; its argument specifies the low bound of the array object returned. Notice that it is probably not useful write

```
var x: seq[t] := seq[t]$null();
```

because all *x* could ever hold is the empty sequence. (All other sequences are too big to fit in *x*.)

```
addh = proc (const s: cvt, e: t) returns (new: cvt[,rep$size(s)+1]);
    new := rep$create (1);
    for const x: t in rep$elements (s) do
        rep$addh (new, x);
        end for;
    rep$addh (new, e);
    end addh;
```

The *addh* operation returns a new sequence with one more element at the end than the one passed in, therefore the size of the returned object is one bigger than the actual size of the argument sequence. (Notice that this is not necessarily the same as 'n + 1'.) The *elements* operation is an iterator that generates the elements of an array in order from the first to the last. The *addl* and *concat* operations are similar to *addh*.

```
addl = proc (const s: cvt, e: t) returns (new: cvt[,rep$size(s)+1]);
    new := rep$create (1);
    rep$addh (new, e);
    for const x: t in rep$elements (s) do
        rep$addh (new, x);
        end for;
    end addl;
```

```
concat = proc (const r, s: cvt) returns (new: cvt[;rep$size(r)+rep$size(s)]);
    new := rep$create (1);
    for const x: t in rep$elements (r) do
        rep$addh (new, x);
        end for;
    for const x: t in rep$elements (s) do
        rep$addh (new, x);
        end for;
    end concat;
```

Now we present the operations that produce shorter sequences from their inputs: *remh*, *reml*, and *trim*.

```
remh = proc (const s: cvt) returns (new: cvt[;max(0,rep$size(s)-1)]) signals (empty);
    n: int := rep$size(s);
    if n < 1
        then signal empty;
        else
            new := rep$create (1);
            index: int := 1;
            while index < n do
                rep$addh (new, s[index]);
                end while;
        end if;
    end remh;

reml = proc (const s: cvt) returns (new: cvt[;max(0,rep$size(s)-1)]) signals (empty)
    n: int := rep$size(s);
    if n < 1
        then signal empty;
        else
            new := rep$create (1);
            index: int := 2;
            while index <= n do
                rep$addh (new, s[index]);
                end while;
        end if;
    end reml;
```

(*Max* is used to prevent a '-1' from messing things up when an empty sequence is passed to *reml*.) *Trim* is given bounds between which elements in the argument sequence are to be retained. *Trim* returns whatever portion of the argument overlaps with the range between the bounds.

```
trim = proc (s: cvt, low, high: int) returns (new: cvt[,max(0,high-low+1)]);
    start: int := max (1, low);
    n: int := rep$size (s);
    end: int := min (n, high);
    new := rep$create (1);
    for const j: int in int$from_to_by (start, end, 1) do
        rep$addh (new, s[j]);
        end for;
    end trim;
```

The *from_to_by* iterator generates the integers from its first argument through to its second argument incrementing by the third argument; it is like an Algol for loop.

Here are the selection operations: *first*, *last*, *fetch*, and *elements*.

```
first = selector () of t from s: cvt signals (empty);
    if rep$size (s) = 0
        then signal empty;
        else select s[1];
        end if;
    end first;

last = selector () of t from s: cvt signals (empty);
    n: int := rep$size (s);
    if n = 0
        then signal empty;
        else select s[n];
        end if;
    end last;

fetch = proc (i: int) of t from s: cvt signals (range);
    if (i < 1) | (i > rep$size(s))
        then signal range;
        else select s[i];
        end if;
    end fetch;
```

The vertical bar is a sugar for the or operation, in this case 'bool$or'.

```
elements = iter (const s: cvt) yields (const t);
    for const e: t in rep$elements(s) do
        yield (e);
        end for;
    end elements;
```

Notice the use of the array iterator *elements* to implement our own iterator. It would be nice to

be able to assign sequences, so we define a *copy* operation.

```
copy = proc (const s: cvt) returns (new: cvt[rep$size(s)]);
   new := s;
   end copy;
```

The *copy* operation will often be this simple, but there are a few types where more must be done. We also provide another very useful operation, *equal*. We note that *equal* needs an extra restriction on *t*.

```
equal = proc (const r, s: cvt) returns (eq: bool)
        where t has equal: proctype(const t, t) returns(bool) end;
   eq := (r = s);
   return;
   end equal;
```

Notice that we use the underlying array *equal* operation, which calls the *equal* operation of *t* to compare the arrays element by element. Now we write the *length* operation.

```
length = proc (const s: cvt) returns (l: int);
   l := rep$size(s);
   end length;
```

It will be helpful to see some example uses of sequences.

First we define a few types:

```
si100 = seq[int; 100];
si_ = seq[int; *];
silen = seq[int; ?len];
```

Now some declarations:

```
a: si100;
b: si100 := si100$null ();
c: si_ := b;
d: silen := c;
if d?len = 0 then ...
```

First, *a* is uninitialized, and has room for sequences up to 100 integers long. The next variable, *b*, is the same size, but has been assigned the null sequence. Since the size of *c* was determined dynamically, it can hold only the null sequence. (This is not very useful – it illustrates why size should normally be specified in declarations.) The same is true of *d*, however its size can be queried by using *d?len* as shown in the if statement. Here are a few more examples:

```
a := a || d;
b := si_$addh (a, 5)
if si_$last (b) = 5 then ...
var j: int := 0;
for const i: int in si_$elements(b) do
   j := j + i;
   end for;
```

Notice that the first line calls the *si100$concat* operation.

We have defined a complete type generator for sequences. This example is atypical in that it has no mutating operations. We chose this over a mutable type because it demonstrates more of the parameter mechanism, since it returns more things, and tends to allocate the minimum storage possible. (Allocating the minimum for mutable types is not always desirable, since they may need to grow later. Furthermore, even if the objects are immutable, larger ones may be assigned to a variable later. Of course the style of use is up to the programmer.)

4.5. Implementation

Here we discuss how to implement ASBAL's parameter mechanism. We first explore techniques for the regular parameters; these methods are borrowed directly from CLU. We then consider the additions necessary for size parameters.

4.5.1. Regular Parameters

The most straightforward idea is to pass parameters as extra arguments in calls. This works fairly well, except when procedures and iterators are passed around as objects. When an instance of a parameterized procedure or iterator is passed around, its parameters must be stored in the object, since they are not available when it is called. Likewise, an operation of a parameterized type must carry the parameters of the type around.

This difficulty suggests what we call the *macro* implementation of parameters. This implementation actually substitutes the actual parameters in and comes up with separate procedures (iterators, selectors, clusters) for each distinct set of parameters. This would seem to be inefficient in terms of memory use, but can be good in some situations. Its main advantages are simplicity, and the ability to do better optimization of code once the substitutions have been

made. Of course, the entire module does not have to be duplicated. One can have a small parameter dependent section which contains the information relevant to parameters, and a pointer to a large parameter independent section. This plan is somewhat like that of using linkage sections to hold per process data in an operating system, with large sections of sharable pure code.

There is still one problem however. It is possible to write programs which generate an unbounded number of different parameters at run-time.[1] Here is a simple procedure (in CLU) that has that property:

```
nasty_a = proc [t: type] (n: int) returns (prototypeinfo);
    at = array[t];
    if n = 0
        then return (nasty_b[t]);
        else return (nasty_a[at] (n - 1));
        end;
    end nasty;

nasty_b = proc [t: type] (n: int);
    ...
    end nasty;
```

It should be easy to see that this generates *t*, array[*t*], ..., array^n[*t*]. Since new types can be generated at run-time, some kind of run-time data structure must be maintained for types. Maintaining the data structure is not too hard, but if it is kept in a fixed-size table then it can potentially overflow. However, the sort of recursion shown above does not appear to be very useful. Therefore we avoid the problem by ruling it out in ASBAL. (Recursions in size parameters are harmless, since different size parameters do not result in different abstractions.)

There is a similar problem with restrictions. Infinitely recursive restrictions can be written. We must deal with them in ASBAL. A compiler must check for recursive restrictions, and prevent the infinite recursion. For details see [Wulf77].

---

1. Gries and Gehani appear to have been the first to identify this problem [Gries77], although there is a footnote (number 19 in [Wulf76a]), a paper from the Algol group, that could be referring to the same difficulty. Our examples are based on the Gries and Gehani article, however.

4.5.2. Implementing Size Parameters

Now we turn to the question of implementing ASBAL's size parameters. First of all, they are not true parameters to the type, and appear only as dummies, or in positions to allow allocation of memory for variables. The basic technique for handling size parameters is to store the size information in the variables. This method leads to a nice implementation of *x?y*: just fetching a component at a fixed offset from the beginning of *x*, very similar to records. Because the sizes are stored with the variables there are no problems of allocating space for size parameters dynamically - the space has already been reserved in each variable. (The next section will discuss storage formats for variable size objects in more detail.) In the case of abstract data types defined by users, the underlying sizes of arrays and strings must be kept for the use of procedures receiving the components as arguments, etc. However, the abstract size parameters must also be kept, for querying and for the size checks required in passing pre-existing variables as return variables (see the next section and Section 4.3.3.2).

4.6. Analysis of Costs of Size Parameters

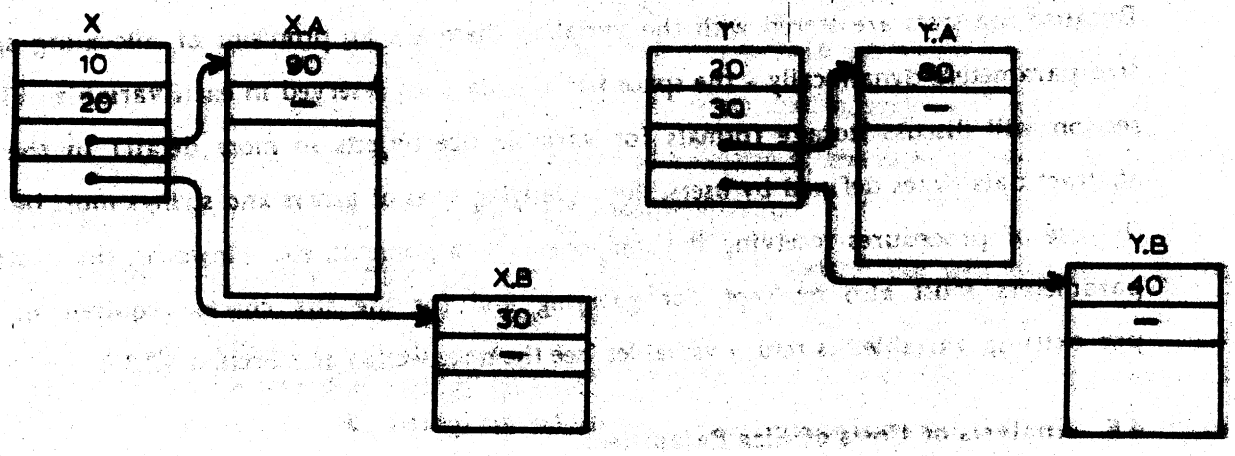
There are two major costs associated with size parameters: storage overhead and processor time, and both are somewhat dependent on the actual storage representation used. Therefore, let us consider the storage efficiency of possible representations, and the extra processor time required by size parameters. Part (b) of Figure 4. shows the most general storage format, one using pointers. This format is simple to use since items are always at compile-time known offsets within substructures, although considerable indexing and indirection may be required to access a deeply nested item. More efficient forms such as the linear format of part (c) of the figure are possible in many cases. Such a linear format saves memory and cuts access time because the pointers do not have to be stored or followed. However, the linear representation is not sufficient for all cases. It is better to adopt a general representation using pointers - we believe that a single storage format should be used throughout the system. Having multiple formats in the system would be bad for the following reasons:

Figure 4. Size Comparison

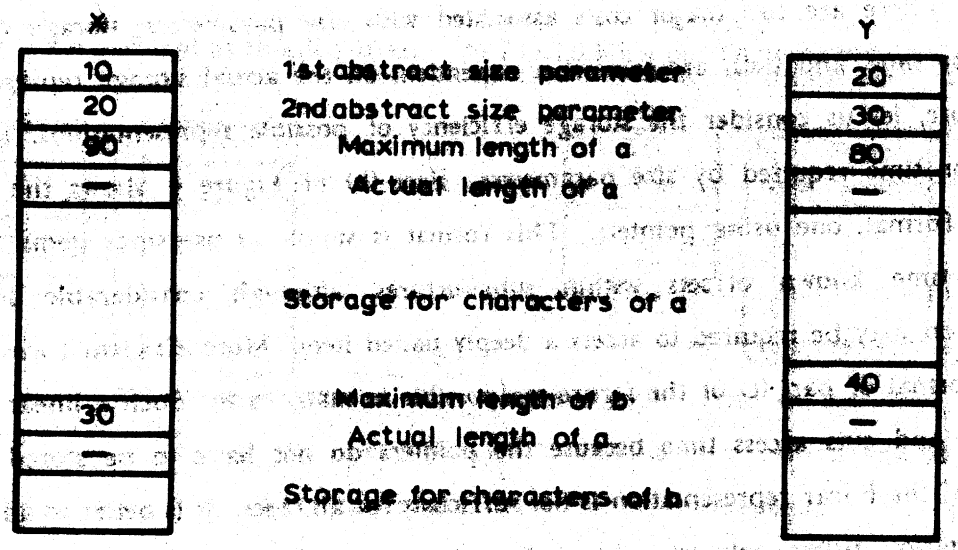
a) Let this be part of the *foo* cluster:

```
foo = cluster [size1, size2] is ... ;
    rep = record[a: string[:100-size1],
                 b: string[:size2+101]];
    ...
    end foo;
```

b) Let *x* be declared as foo[:10,20], and *y* as foo[:20,30]; a pointer representation produces:



c) An optimized representation for *x* and *y* is linear:



(1) code generation would be made more complicated;

(2) if multiple copies of modules were made, each handling one storage format, modularity would be threatened;

(3) if, on the other hand, modules could handle any storage format, the code would be larger, or interpretive execution would slow processing;

(4) the entire run-time system would be more complex, while the payoff might be small.

Thus, the optimization to a format more compact than the pointer format may not be worth the complications it introduces.

Given that we will use the pointer storage format, let us examine the cost of size parameters in detail. First, let us see how much storage overhead is introduced by having size parameters. The storage overhead for size parameters consists of one integer per abstract size parameter per object of an abstract data type, plus one pointer for each array or string component. It is hard to assess just how much impact this overhead will have, because it entirely depends on how often variable size objects are used, and whether the arrays and strings in them tend to be large or small. Dope vectors have been accepted in many languages, and size parameters are only a generalization of dope vectors. We believe that the storage overhead for size parameters is acceptable. Besides, this overhead is unavoidable because size parameters can be determined at run-time. Therefore, we suggest that storage overhead is less of a problem than processor time.

In examining processing overhead, we consider the bit-copy operation first. For fixed size types a bit-copy can be accomplished with a block move, provided pointers to components are represented as relative offsets, and all components are packed together linearly. (Both provisos are possible, and the offsets can be determined at compile-time.) With care, the components of an object of a variable size type can be packed together in a similar way, although the offsets will generally be computed at run-time, and the order of the parts accessed through pointers (i.e., offsets) may not always be the same. The time when care must be exercised is in the initial construction of the object, because the components will never be moved later. The only information that should be stored at a fixed offset in the stack frame for variables of a variable size type is a pointer to the object itself in the dynamic part of the

frame; in that way a single, uniform storage format is achieved.

Run-time size checks comprise the remaining overhead. In Section 4.3 we made a decision regarding how closely return variables size specifications must match the sizes of pre-existing variables; namely, we decided to allow any pre-existing variable to be used so long as each of its arrays and strings were at least as large as those specified for the return variable. Thus if x were a pre-existing variable in Figure 4, and [illegible] were a specification for a return variable intended to be x, the size match would fail. It would not fail because the abstract size parameters ((10, 20) and (20, 30)) are different, but because the b component of x is too small. In general, this size comparison requires a tree-walk of the abstract type tree for the return variable specification and the pre-existing variable; that is, array and string sizes must be compared pairwise.[1] For example, for x and y in Figure 4, the sizes of the a components would be compared, and the sizes of the b components.

Requiring size parameters to match exactly would reduce the processor time spent doing size checks, because if the sizes match exactly at the abstract level, all lower levels match, too, so they need not be checked. We opted for flexibility rather than efficiency largely just to investigate a new idea more deeply; we should see whether programmers feel they need the extra flexibility before we eliminate it.

In sum, the major costs of size parameters are storage overhead and run-time size checks. The storage overhead is not exorbitant, and is of a sort people have come to accept. On the other hand, run-time checks are much less acceptable. Requiring sizes to match exactly on assignments reduces the overhead significantly, but it is not known whether the flexibility of non-exact size parameter matching is required.

It appears that optimizing run-time size checks will be difficult, though we can always hope some simple techniques will be developed for the cases most commonly encountered in practice. Optimizations of storage representations may often be possible, but will be undesirable since such optimizations will tend to expand code, reduce modularity, increase execution time, and generally increase complexity of the run-time system.

---

1. Note that only one pair of strings in an array of strings need be compared; this generalizes to arrays of any type.

4.7. Summary

The flexibility gained with size parameters can be expensive. However, size parameters are really just a generalization of the bounds of arrays, and many of the same implementation techniques apply. Notice that if size parameters are required to match exactly in assignments, we have a scheme very close to those where size is part of the type. However, we have avoided several difficulties associated with having size be part of the type of objects:

(1) We do not have different operations for objects of different sizes;

(2) and thus we have prevented an explosion of parameters to operations.

(3) We know explicitly which parameters must be compile-time known, and which may be computed at run-time;

(4) and because types (as opposed to sizes) must be compile-time known, we avoid having run-time type objects (i.e., objects of type type) at run-time, although we do require run-time size information;

(5) and, again because types are compile-time known, we can perform all the difficult type checking at compile-time.

Although it is a matter of opinion, we feel that separating size information results in a cleaner notion of type and helps to separate abstract concerns from implementation details. Overall, we are certain of the usefulness of regular parameters, and believe that size parameters are also helpful in programming.

5. Areas and Pointers

In this chapter we present a mechanism for dynamic storage allocation. It allows programs to build general graph-like data structures without requiring garbage collection or much run-time overhead. Furthermore, the use of dangling references can be prevented at compile-time. Our presentation begins with *areas*, the objects that perform storage allocation. The discussion of areas is followed by a description of *pointers*, the objects used to name objects allocated in areas. We then present details of using areas and pointers; it is here that the techniques used to prevent dangling references are developed.

After presenting the area and pointer mechanisms, we discuss the impact of the mechanisms on aliasing, comment on the copy problem, and present a variety of methods that might be used to implement areas. Lastly we have three examples to illustrate the use of the mechanisms.

5.1. Areas

An area, in its simplest implementation,[1] is a block of storage in a stack frame, somewhat like an array. The idea is that the area parcels out this block dynamically, on request. Areas are based on the collections of Euclid [Lampson77], but there are several important differences. The major difference is that a collection allocates objects of a single type, whereas objects of different types can coexist in the same area. Thus an area bounds only the total amount of storage used rather than bounding the number of objects of each type separately, as collections would. This can lead to better storage utilization.

The simplest allocation method is to allocate objects linearly from one end of the area to the other. No reclamation is done; because areas are in the stack, the space for an entire area can be reclaimed when the frame it is in is released.[2] When the size of a requested allocation is larger than the remaining space, the allocation operation will fail. This allocation technique brings out the similarity between areas and arrays: arrays can grow dynamically using the *addh*

1. We will discuss more sophisticated implementation schemes for areas later in this chapter. However, the general properties of areas will hold true for any implementation.
2. Again, we will outline implementation schemes that do more (e.g., reclamation) later.

or *addl* operations; areas allocate new components dynamically in like fashion. The similarity ends there, however, because arrays are homogeneous aggregates and areas are heterogeneous.

Pointers are used to access objects allocated in areas. Following a pointer is not unlike indexing an array, but a pointer's type includes the area in which the object pointed to resides, and the type of the object, for safety. Thus the type generator ptr (for pointer) takes two parameters: an area and a type; ptr[*a*,*t*] means a pointer to an object of type *t* in the area *a*. (The type generator ptr and the use of areas as parameters will be discussed in more detail below.) The allocation of objects in areas is performed by the operation ptr[*a*,*t*]$*alloc*. It takes one argument, an object that is copied to produce the newly allocated object. The new object is created using *t*$*copy*. The type of ptr[*a*,*t*]$*alloc* is

```
proctype (const t) returns (ptr[a,t])
```

where *a* is an area and *t* is a type. *Alloc* signals *failure*("*area out of memory*") when there is not enough memory left in the area to allocate an object of the requested type. If *a* is an area, then

```
var p: ptr[a,int] := ptr[a,int]$alloc(5);
```

is a legal declaration with initialization. Its effect is to allocate an integer in the area *a*, and set the pointer variable *p* to point to that newly allocated integer. In this case the new integer is *5*.

Corresponding to *alloc*, there is a selector, *deref*, used to access objects allocated in areas; the type of ptr[*a*,*t*]$*deref* is

```
seltype () of t from ptr[a,t] signals (bad_pointer)
```

where *a* is an area, and *t* is a type. *Deref* signals *bad_pointer* when given a null pointer to follow. (The null pointer will be discussed below.) An unsugared use of *deref* is

```
ptr[a,int]$deref(p)
```

The standard selection sugar allows this to be written as

```
p.deref
```

However, there is a special sugar for *deref* which is more convenient than either of the previous forms:

```
p↑
```

There is no *free* operation to release previously allocated storage. *Free* would be unsafe, or if safe, prohibitively expensive. Having *free* and requiring safety would amount to requiring objects to be reference counted, and still one could not truly free cyclic structures without first

breaking the cycles. We feel that reference counting is too expensive to justify requiring it for all areas. However, particular areas can do reference counting with some assistance from the compiler; there would still be no explicit *free* operation, but setting a pointer to nilptr might cause the object previously referred to by the pointer to be freed.

5.2. Pointers

For each area *a* and each type *t* there is a pointer type ptr[*a*,*t*]. The objects of that pointer type are pointers to objects of type *t* in area *a*. There are five operations of the type ptr[*a*,*t*]; in addition to *alloc* and *deref*, which were previously introduced), we have:

(3) equal: proctype(const ptr[*a*,*t*], ptr[*a*,*t*]) returns (bool) – returns true if and only if the two pointers point to the same object (i.e., the same location in the same area);

(4) copy: proctype(const ptr[*a*,*t*]) returns (ptr[*a*,*t*]) – copies its argument (the pointer, *not* the object pointed to); ptr[*a*,*t*]$equal(*p*, ptr[*a*,*t*]$copy(*p*)) will return true;

(5) null: proctype () returns (ptr[*a*,*t*]) – always returns the null pointer, a pointer which points to no object. (Remember that following the null pointer fails and signals an exception.)

There is a sugar for ptr[*a*,*t*]$null(); it is nilptr. Notice that nilptr carries no designation of pointer type – the correct type can be obtained from context except in the case of nilptr†, which will always signal *bad_pointer* anyway. The *only* sources of pointers are *alloc* and nilptr.

5.3. Using Pointers and Areas

Up to this point we have described some features of areas and pointers, but have omitted several crucial points. A goal in the design of the area mechanism is safety. In particular, we desire to prevent dangling references with only compile-time checks. Prevention of dangling references depends equally on several different parts of the design: it is the synthesis of these parts that achieves our goal of safety, and not the individual parts.

The technique used to prevent dangling references is basically the following. We use the syntactic scope of each area object to define a dynamic, semantic scope of the area object at

run-time, i.e., we arrange things such that the area is only nameable where it will exist when the program is run. We also arrange for any object that might contain (or try to construct) references to objects in an area, to have the area's name as part of its type. This "trick" allows standard type checking to prevent dangling references at compile-time. Thus, we use the standard type checking restrictions of the language to get as much of the checking as we can.

5.3.1. Area Creation

Area is a type, and areas are objects of the type area. The only operation of the type area is area$*new*, which is used to create new areas. This operation takes two arguments: a string (describing what sort of area management scheme is to be used,[1] e.g., "simple" or "ref_counted"), and an integer (describing the size of the area to be created, e.g., in words, or bytes, or some other standard unit such as the size of an int). Thus the type of area$*new* is

proctype(const string, int) returns (area) signals (bad_arguments(string))

The exact meaning of both of the arguments is system dependent: the number and kinds of area management schemes, and their names are determined by the language implementation; the unit storage size is determined by the implementation, and the meaning of the size argument may depend on the area management scheme chosen, as well. Of course area$*new* may signal if its arguments are improper (e.g., the size is negative).

Although area is a type, we do not allow variables of type area; in fact, only two things can be done with areas: they may be created, and they may be used as actual parameters. Area variables are a bad idea because area assignment is dangerous - area assignment could result in dangling references to the area written over by the assignment.

Since there are no area variables, a special statement is used to create new areas, the new statement. For example,

```
new a: area = area$new ("simple", 500);
```

creates a new area *a*, of the "simple" variety and of size 500 units. The new statement is intended to parallel constant definitions, and the scope of the area introduced in a new statement is the same as the scope an identifier in a constant definition would have in the same

1. See Section 5.6, which is about implementing areas, for several area management schemes.

position. However, the right-hand side of the := in a new statement must be a call of area$new. Furthermore, the new statement is the only construct that may call area$new.

5.3.2. Pointers and Areas in Reps

It is desirable to allow pointers in the representations of abstract data types. Having pointers in reps permits dynamic data structures to be built and referred to, one of the goals of the area mechanism. However, any type using pointers in its representation is relying on the area which contains the objects pointed to. Let us make explicit the notion of a type relying on an area: a type is said to *depend on* an area if that area might possibly be accessed via objects of the type. Thus the types array[ptr(a;)] and pair[ptr(a;), ptr(b;)] both depend on area *a*, and the second type depends on area *b* as well.

We mentioned our method of preventing dangling references above; we make types depending on an area unwritable outside the scope of the area. We make the types unwritable by requiring any type depending on an area to take that area as a parameter. Thus, areas are not global and dependence on them must be explicitly stated. For example, a type *list* that allocates its elements in area *a* must take *a* as a parameter. Recall that, once created, the only way an area may be used is as a parameter. We see that this also is allowed so that area dependent types (and other abstractions) can be appropriately associated with the areas on which they depend.

An example type definition using pointers in its representation is given below:

```
binary_tree = cluster [a: area, t: type] is ...
    ...
    rep = ptr[a, node];
    node = record [left, right: rep];
    ...
end binary_tree;
```

Notice that the type *node* is recursive. CLU does not allow such directly recursive definitions, but we find them useful, and hence allow them. Two types are defined to be equal if and only

if their (possibly infinite) specifications are the same.[1] A procedure that used a binary tree as a temporary data structure might look like this:

```
foo = proc ...
    ...
    begin
        const a: area = area$new("simple", n);
        bintree = binary_tree[a, int];
        ... bintree$... ...
        ...
        end;
    end foo;
```

The other thing to notice about *binary_tree* is that it takes *a* as a parameter; it must do so to use *a* in its representation.

Why are no other uses of areas other than as parameters allowed? As was argued before, area variables are dangerous because assignment of areas is an uncontrollable source of dangling references. Other uses of areas, such as storing them in data structures, or passing them as arguments, tend to destroy the static scoping required so that the compile-time checks to prevent dangling references will work. Besides, since such dynamic positions may not be used as parameters, and ptr takes the area pointed into as a parameter, these dynamic uses of areas would not be helpful: the usefulness of areas depends on pointer types, and if in some context the type of pointers into an area cannot be expressed, nothing can be done with the area. In sum, there is no way for dangling references to arise from data structures because the type of the data structure depending on an area cannot be expressed anywhere the area does not exist.

5.3.3. Closing the Loopholes

As demonstrated above, dangling references cannot arise from data structures. However, there are more possible sources of dangling references. For example, the procedure ptr[*a*,*t*]$*alloc* is clearly bound to the area *a*, and we would not like that procedure to be usable

---

1. This rule is the same as that used in Algol 68. See [Wijngaarden77] for an algorithm for checking type (mode) equivalence.

where *a* does not exist. One might think, "The area *a* is named in writing out ptr[a,i]$*alloc*, so there is no danger." However, there *is* a potential danger. Consider the following procedure definition:

```
foo = proc[a: area](const i: int) returns (int);
  ...
  end foo;
```

The assignment statement below is presumably legal:

```
p := foo[a];
```

where *a* is an area, and the type of *p* is

```
proctype(const int) returns (int)
```

Therefore, we can write

```
var p: proctype(const int) returns (int);
begin
  const a: area = establish("simple", 100);
  p := foo[a];
  end;
p(5);
```

The code pictured above may follow a dangling reference to the area *a*; that reference is hidden in the procedure object assigned to *p*. We say that any routine that might access an area *depends on* that area; if a routine creates an area it does not depend on that area. Just as with other objects, we would like the type of routine objects to reflect any dependence on areas. Most routines' types do refer to the areas they use. For example, the type of ptr[a,i]$*alloc* is

```
proctype(const i) returns(ptr[a,i])
```

and that of ptr[a,i]$*deref* is

```
seltype() of i from ptr[a,i] signals(bad_pointer)
```

and both types refer to *a*. To prevent dangling references of the sort exhibited by *foo* above, we prohibit routines from taking an area as a parameter without having that area as part of their type. Thus the procedure *foo* above is illegal. Since a routine must take an area as a parameter to be able to access it, this guarantees that the type of any routine that might access an area names that area.

We are not ruling out anything useful by prohibiting routines like *foo*. If the type of a routine does not refer to any areas, then no objects depending on the areas can be passed

through the routine's interface. And if no objects depending on an area are passed through a routine's interface, then there is no point in the routine's taking the area as a parameter in the first place: if the area is to be used only locally, the routine may as well create an area for its own private use.

Another loophole is the use of area as an actual parameter in a position requiring a type. For example, if an abstraction has a type parameter *t*, it may declare variables of type *t*, arrays of type array[*t*], etc. Previous restrictions we have made prohibit the use of areas as variables and their storage in data structures. Therefore, we must make an additional restriction that area may not be used as an actual type parameter.

5.3.4. Summary

Here are the restrictions we made to prevent dangling references:

(1) areas, once created, may only be used as actual parameters;

(2) if a routine takes an area as a parameter, then that area must appear in the type of the routine.

(3) area may not be used as an actual type parameter.

In addition, pointers may not be used as parameters, though pointer *types* may be. Thus array[ptr[*a*,*t*]] is a legal type, but *bar*[*p*] where *p* is of type ptr[*a*,*t*] is not. (It is not clear what meaning can be attached to pointers as parameters anyway.)

With the restrictions stated above there is no possible way of following a dangling reference in ASBAL. However, it is possible to create a dangling reference that can never be followed. Consider this fragment of code:

```
new a: area = area$new("simple", 100);
...
begin
    new b: area = area$new("simple", 100);
    ptype = ptr[b, int];
    qtype = ptr[a, ptype];
    var p: ptype := ptr[b,int]$alloc(5);
    var q: qtype := ptr[a,ptype]$alloc(p);
    ...
    end;
...
```

After the begin block is exited, the pointer constructed in area *a* by the initialization of *q* will remain. That pointer will be dangling because it points into area *b*, which has been destroyed. However that pointer can never be accessed because its type (namely ptr[a, ptr[b, int]]) cannot be written outside the begin block (the scope of *b*). Even if the begin block were in a loop, there is no way to "remember" such a dangling pointer and use it again when it would be invalid.

5.4. Pointers and Aliasing

With the addition of pointers, we arrive at a universe of objects very similar to CLU's. We gain many advantages: sharing, the ability to build general data structures, etc.; but we gain the same disadvantages present in CLU. For one thing, sharing is a double-edged sword. We must accept that a dereferenced pointer may overlap with almost anything of the same type. It may overlap with an argument, with a selection, or with any other dereferenced pointer of the same type. What kind of aliasing rule should we have in this situation? Our approach is based on Euclid [Lampson77]. Aliasing rules are used to prevent *unexpected* sharing, not all sharing. The sharing possible when pointers are dereferenced is considered to be expected, so no checks are made necessary at a dereferencing. However, var arguments still cannot overlap, so if two *dereferenced* pointers are passed as arguments there may be a run-time check; but no checks are necessary if the pointers themselves are passed. Note that only (dereferenced) pointers of the same type need be checked; no others can possibly overlap. The sharing possible through pointers is not quite as bad as the sharing potentially possible in selections: an object in an area is never destroyed by having another object written over it; objects in areas

may only be mutated. (This is because *p*↑ is a selection and cannot be assigned to.)

Another problem, which was mentioned when selections and aliasing were first discussed in Chapter 2, is that having an object as a const does not guarantee that the object's state will not change. This is because the object may be accessible via another path as a var, for example, by following a chain of pointers. However, even though a const may be mutated under some conditions, there is a simple condition under which it can be guaranteed not to be mutated: the object is not allocated in an area. Tests for aliasing always catch overlapping objects residing in the same variable, so if an object that is physically part of a variable is accessible as a const, then we can be sure it will not be mutated. The aliasing detection checks performed before procedure calls guarantee this. On the other hand, if the object in question is in an area, it might be mutated via pointers other than the one used to access it, but its identity can never change because the implicit variable it resides in can never be assigned to. This is an advantage of dereferencing to objects instead of to variables.

Note that if an object has components that are stored in an area, then its components can always be replaced by replacing the pointers to them; we lose no useful ability through dereferencing pointers to objects instead of to variables. The major disadvantage of sharing in ASBAL is the same as its major disadvantage in CLU: sharing makes verification and proofs about programs difficult, by requiring more complex axioms and proof rules. The complication of proofs resulting from sharing is an as yet unsolved problem common to all languages having pointers or sharing.

5.5. The Copy Problem

When presented with an object to copy that contains pointers, should we copy just the pointers, or the objects pointed to as well? The problem is that some types require copying the objects pointed to, and other types forbid it. As discussed in the second chapter, the only solution to the problem is to have each type provide a *copy* operation, which does the appropriate thing for that type.

In CLU, a *copy* operation will usually copy the objects referred to instead of the references, but both sorts of copying are provided in many cases. For example, CLU has two copy operations for arrays, called *copy* and *copyl*; *copy* does a full, recursive copy while *copyl*

copies only object references. However, references are always implicit in CLU, whereas they are always explicit in ALPHARD. Because we have explicit references, and because our objects can physically contain components, *copy* operations in ALPHARD will usually copy only what is physically contained in an object. For example, the *copy* operation of records and arrays is of this sort. This gives us the following recursive definition of copying for records and arrays: each component is copied using the *copy* operation of its type. Recursive copying of this sort is usually desired, but the implementer of a type has complete freedom in writing the *copy* operation of that type.

The copy problem is closely related to another problem which we call the *equivalence problem*. In general there is a hierarchy of equivalence relations on the objects of a type, and it is not at all obvious which one to use when the question is asked "Are these two objects equal (equivalent)?" For some types the strongest useful equivalence relation is that of identity: two objects are equivalent if and only if they are the same physical object. For other types, equivalence of the states of objects is used; this is often what is meant by an equality test in a program, and is the usual definition of *equal* in ALPHARD. In some cases even looser equivalence relations are useful. At any rate, equivalence testing must be an operation of a type just as copying must be. For example, consider sets represented by a linked list. Two sets are considered equal if and only if they have the same members – the order of the members in the linked list does not matter. Therefore, equality of linked lists is not the appropriate equivalence relation for sets; it is too strong. However, *equal* is generally defined recursively on data structures, so each type should provide it.

5.6. Implementing Areas and Pointers

Our original notion of an area was a block of storage allocated in a stack frame, and that of a pointer was a machine address (or possibly an offset from the beginning of the area). However, many other implementations of areas are possible. Areas could be blocks of memory taken from a storage pool separate from the stack. This implementation requires more run-time support code, but has more flexibility. For example, areas could grab more blocks of storage automatically if their original amount was used up. Fixed size blocks would be allocated, and the blocks used by an area would be returned to a pool of free blocks when the

area was destroyed. Thus, very efficient storage management would be possible. One could even go so far as to copy currently inaccessible areas out to on-line mass storage devices to get an effective increase in address space. An somewhat different approach is to implement a single heap in which all areas allocate their objects. The objects of each area could be chained together to be freed when the area is destroyed.

Somewhat orthogonal to the source of the storage is its management. The simplest scheme has been mentioned before: linear allocation with no reclamation. However, areas could reference count their objects; *alloc* and the pointer copy operation could have code to maintain the counts. (The compiler would have to help in noting pointers that are destroyed, however.) With more sophisticated run-time support, various garbage collection schemes could be implemented.[1] Our goal has been to avoid the *necessity* of garbage collection, but that does not mean that we cannot provide it when asked.

One merit of areas is that they allow many different storage management schemes to coexist, if care is taken. Hence, storage management facilities can be tailored to the programmer's needs in each problem, even within different parts of the same program.

We believe areas are a flexible and potentially efficient alternative to global garbage collection. Pointers can be as efficient as machine addresses, and allocation within areas need not be slow – area routines will most likely be hand coded in assembly language. Arguments to the routines will be in terms of machine addresses, offsets, and numbers rather than types, etc., because they will be called by object code and not directly by users. The ability to tailor storage management to the task is probably the biggest advantage of areas over a global storage management scheme.

---

1. The main difficulty is supplying the information required for tracing. See Bishop [Bishop77] for applicable partial garbage collection techniques. Perhaps these techniques could be combined with Baker's ideas on incremental garbage collection [Baker77], or with the transaction file methods [Deutsch76, Barth77] to provide areas that do local, incremental garbage collection.

5.7. Example One - Queues

We now present the first of our three examples using pointers and areas - queues. Here we essentially re-work the example in Chapter 2, adding appropriate features we have presented since then. Here is the new representation:

```
queue = cluster [a: area, t: type] is ...;

rep = record[first: ptype,
             last: ptype];

ptype = ptr[a, element];

element = record[next: ptype,
                 number: t];
```

We use a linked list as the representation of a queue. The *first* pointer is null if and only if the queue is empty; the *next* pointer of the last element is always null. See Figure 5 for the entire *queue* cluster. Pointers are a lot more convenient than the *qrep* values we used before, because the pointer object itself can be used as a place holder. Therefore we do not have to write the rather verbose *figure* statements as we did when using *qrep*'s. Notice that we have generalized queues of integers to queues of any type.

Figure 5. The Queue Cluster

```
queue = cluster[a: area, t: type] is create, insert, remove, members;
        where t has copy: proctype(const t) returns(t) end;

    rep = record[first, last: ptype];

    ptype = ptr [a, element];

    element = record[next: ptype, member: t];

create = proc () returns (q: cvt);
    q := rep${first, last: nilptr};
    end create;

insert = proc (var q: cvt, const x: t);
    var p: ptype := ptr[a, element]$alloc(element${next: nilptr, member: x});
    if q.first = nilptr
        then q.first := p;
        else q.last↑.next := p;
        end if;
        q.last := p;
    end insert;

remove = proc (var q: cvt) returns (x: t) signals (empty);
    if q.first = nilptr
        then signal empty;
        else
            x := q.first↑.member;
            q.first := q.first↑.next;
        end if;
    end remove;

members = iter (const q: cvt) yields (const t);
    var p: ptype := q.first;
    while p ~= nilptr do
        yield (p↑.member);
        p := p↑.next;
        end while;
    end members;

end queue;
```

5.8. Example Two - Sorted Bags

This example is a re-working of the example in Chapter 3. As with queues, we have improved bags by parameterizing them. The new representation for bags, *t*, is:

```
rep = record[count:  int,
             size:   int,
             root:   pnode];

pnode = ptr[a, node];

node = record[element: t,
              count:   int,
              left:    pnode,
              right:   pnode];
```

This representation is essentially the same as the previous one with the array replaced by an area, and array indexes replaced by pointers. Figure 8 presents the entire cluster. We feel the new implementation is much more elegant than the previous one. In the previous cluster, the array containing the nodes had to be passed in any recursive calls in the previous cluster, whereas here the "passing" of the area is implicit.

Figure 6. The Sorted Bag Cluster

```
bag = cluster[a: area, t: type] is create, insert, count, size, increasing;
      where t has copy:      proctype(const t) returns(t),
                  equal, lt: proctype(const t, t) returns(bool) end;

    rep = record[count:  int,
                 size:   int,
                 root:   pnode];
    pnode = ptr[a, node];
    node = record[element: t,
                  count:  int,
                  left:   pnode,
                  right:  pnode];

    create = proc () returns (b: cvt);
       b := rep${count: 0, size: 0, root: nilptr}
       end create;

    insert = proc (var b: cvt, const x: t);
       b.count := b.count + 1;
       const new_ptr: pnode, allocated: bool = insert1 (b.root, x);
       b.root := new_ptr;
       if allocated then b.size := b.size + 1; end if;
       end insert;

    insert1 = proc (const p: pnode, x: t) returns (q: pnode, allocated:bool);
       if p = nilptr
            then
               q := ptr[a, node]$alloc(node${element: x, count: 1, left, right: nilptr});
               allocated := true;
       elseif p↑.element = x
            then p↑.count := p↑.count + 1;
       elseif p↑.element < x
            then q, allocated := insert1 (p↑.left, x);
            else q, allocated := insert1 (p↑.right, x);
            end if;
       end insert1;
```

Figure 6. (continued)

```
size = proc (const b: cvt) returns (s: int);
    s := b.size;
    end size;

count = proc (const b: cvt) returns (c: int);
    c := b.count;
    end count;

increasing = iter (const b: cvt) yields (const t, int);
    for const e: t, c: int in increasing1 (b.root) do
        yield (e, c);
        end for;
    end increasing;

increasing1 = iter (const p: pnode) yields (const t, int);
    if p = nilptr then return; end if;
    for const e: t, c: int in increasing1(p↑.left) do
        yield (e, c);
        end for;
    yield (p↑.element, p↑.count);
    for const e: t, c: int in increasing1(p↑.right) do
        yield (e, c);
        end for;
    end increasing1;

end bag;
```

5.9. Example Three - Symbol Table

Our last example is a new one. The abstraction is taken from [Wulf76c], and is presented to allow comparison with Alphard. A symbol table performs a mapping from strings (representing identifiers) to attribute objects in block-structured fashion. Here is the cluster header:

```
symtab = cluster[a: area, attr: type] is create, insert, is_defined,
                                         enter_block, leave_block, lookup;
         where attr has copy: proctype(const attr) returns(attr) end;
```

and a description of the operations:

create: proctype () returns (symtab)
creates a new, empty, symbol table

insert: proctype (var symtab, const string, attr) signals (defined)
inserts a new symbol with initial attributes; signals *defined* if the symbol is already defined at this block level

is_defined: proctype (const symtab, string) returns(bool)
returns true if and only if the symbol is defined at this block level

enter_block: proctype (var symtab)
performs whatever housekeeping is necessary for a new block level

leave_block: proctype (var symtab) signals (underflow)
flushes symbols of top level and drops back a level; signals *underflow* if an attempt is made to leave the outermost block level

lookup: selftype (string) of attr from symtab signals (not_present)
selects the attribute object for the symbol (if any); signals *not_present* if there the symbol is not in the table

A hash table will be used to look up the symbols in the table. We will use a linked list for symbols hashing to the same bucket; the hash table will be used to fetch such a list. Each entry in one of these lists will be a pointer to the data structure for one symbol. This data structure consists of the name of the symbol (a string), and the stack of entries made for that symbol. Each block is represented by the list of the symbols defined in it, and the blocks are stored in a stack. An actual statement of the representation should make this more clear:

```
rep = record[level:      int,
             blocks:     blk_stk,
             hash_table: hashtab];

blk_stk = stack[a, block];
block = record[symbols: symlist];
symlist = list[a, p_sym_ent]];

hashtab = array[bucket: bucket];
n = some integer;

bucket = list[a, p_sym_ent];
p_sym_ent = ptr[a, sym_entry];

sym_entry = record[symbol: string,
                   stack:  attr_stk];

attr_stk = stack[a, attr_entry];
attr_entry = record[level:      int,
                    attributes: attr];
```

See Figure 7 for an example of a symbol table representation after some operations have been performed. Here are brief summaries of the operations of *stack[a,t]* and *list*.

Operations of *stack[a,t]*:

(1) create: proctype() returns(stack[a,t])
creates a new, empty stack

(2) push: proctype(var stack[a,t], const t)
pushes a new object on the stack

(3) pop: proctype(var stack[a,t]) returns(t) signals(underflow)
pops the top element off the stack;
signals *underflow* if given an empty stack

(4) top: seltype() of t from stack[a,t] signals(empty)
selects the top element of the stack;
signals *empty* if given an empty stack

(5) empty: proctype(const stack[a,t]) returns(bool)
returns true if and only if the stack is empty

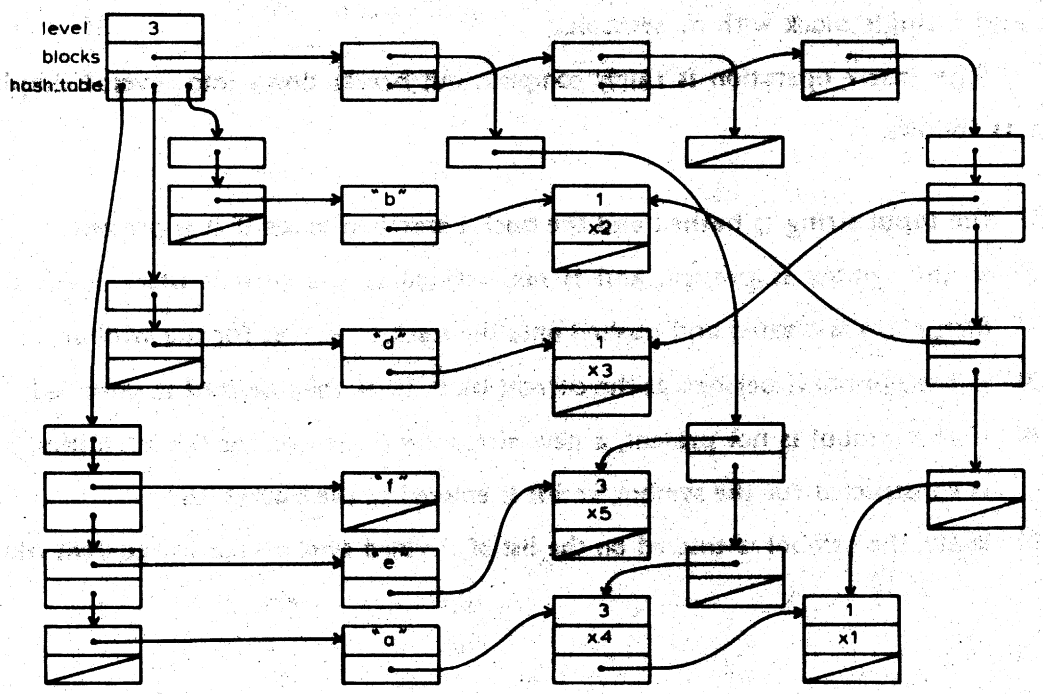
The operations of *list[a,t]* that we use:

Figure 7. A Snapshot of a Symbol Table

Below is a drawing of the representation of a symbol table after the following operations have been performed on it:

create
insert: a, x1
insert: b, x2
insert: d, x3
enter_block
enter_block
insert: a, x4
insert: c, x5
enter_block
insert: f, x6
leave_block

(Assume that *a*, *c*, and *f* hash to the same bucket, and *list* and *stack* are implemented with linked lists.)



(1) create: proctype() returns(list[a,t])
returns a new, empty list

(2) cons: proctype(const t, list[a,t]) returns (list[a,t])
returns a new list consisting of its argument plus a new element at the front of the list;
the new element is made with *t$copy*

(3) members: itertype(list[a,t]) yields(t)
yields the elements of the list in order

Now we present the operations of the *symtab* cluster, one at a time.

```
create = proc () returns (s: cvt);
   s := rep${level:      1,
             blocks:     blk_stk$create(),
             hash_table: hashtable$[bucket]create(), 1, n)};
   blk_stk$push(s.blocks, block${symbols: symlist$create()});
   end create;
```

Thus *create* returns a symbol table at block level 1, the outermost block, with an empty hash table, and a single block with no symbols.

The *insert* operation is fairly complex, but breaks down into several simple cases. It works as follows:

(1) the input string is hashed and the bucket searched to see if it is present;

(2) if the symbol is present, and is not defined at the current block level, then a new *attr_entry* is created and pushed onto the stack of entries for the symbol;

(3) if the symbol is defined at the current block level, then *defined* is signalled;

(4) if the symbol is not present, a new *attr_entry* is created for the attributes, a *sym_entry* is constructed for the symbol, and it is entered in the bucket list;

(5) lastly, the symbol is entered on the list of defined symbols for the current block.

```
insert = proc (var s: cvt, const sym: string, attrib: attr) signals (defined);
    const bkt_num: int = hash(sym);
    var p: p_sym_ent := nilptr;
    for p in bucket$members(s.hash_table[bkt_num]) do
        if p↑.symbol = sym
          then
              if attr_stk$empty(p↑.stack) cor p↑.stack.top.level~=s.level
                then
                    attr_stk$push(p↑.stack,
                                  attr_entry${level:      s.level,
                                              attributes: attrib});
                    break;
                else signal defined;
                end if;
          end if;
        end for;
    if p = nilptr
        then
          p := ptr[a, sym_entry]$alloc(
                  sym_entry${symbol: sym,
                             stack:  attr_stk$create()});
          attr_stk$push(p↑.stack, attr_entry${level:      s.level,
                                              attributes: attrib});
          s.hash_table[bkt_num] := bucket$cons(p, s.hash_table[bkt_num]);
        end if;
    const newblk: symlist = symlist$cons(p, blk_stk$top(s.blocks).symbols);
    blk_stk$top(s.blocks).symbols := block${symbols: newblk};
    end insert;
```

The operator cor (for *conditional or*) evaluates its second argument only if the first argument is false; its value is the logical *or* of its arguments. There is also a cand operator: conditional *and*, and it evaluates its second argument only if the first argument is true. The regular *and* and *or* operators are sugars for calls, and therefore always evaluate *both* arguments. The cor used above prevents our following a null pointer.

The rest of the operations, *is_defined*, *enter_block*, *leave_block*, and *lookup*, are straightforward. Notice that *leave_block* must throw away all symbol definitions for the block being exited. However, it does *not* throw away an empty *sym_stk*; in this sense a symbol, once entered, is never deleted.

```
is_defined = proc (const s: cvt, sym: string) returns (d: bool);
    for const p: p_sym_ent in bucket$members(s.hash_table[hash(sym)]) do
        if p†.symbol = sym
          then
              d := (~attr_stk$empty(p†.stack) cand p†.stack.top.level=s.level);
              return;
          end if;
        end for;
    d := false;
    end is_defined;

enter_block = proc (var s: cvt);
    blk_stk$push(s.blocks, block${symbols: symlist$create()});
    s.level := s.level + 1;
    end enter_block;

leave_block = proc (var s: cvt) signals (underflow);
    if s.level = 1 then signal underflow; end if;
    s.level := s.level - 1;
    for var q: p_sym_ent in symlist$members(blk_stk$pop(s.blocks).symbols) do
        attr_stk$pop(q†.stack);
        end for;
    end leave_block;

lookup = selector (sym: string) of attr from s: cvt signals (not_present);
    for const p: p_sym_ent in bucket$members(s.hash_table[hash(sym)]) do
        if p†.symbol = sym
          then
              if sym_stk$empty(p†.stack)
                then signal not_present;
                else select p†.stack.top.attributes;
              end if;
          end if;
        end for;
    signal not_present;
    end lookup;

end symtab;
```

5.10. Comparison of Area- and Stack-Based Programming

There are considerable differences between designing clusters for objects in the stack and ones for objects to be allocated in areas. Unfortunately the user must plan ahead because abstractions designed for the one storage mode will rarely be convertible to operate in the other. The reason is that stack- and area-based abstractions take different parameters: stack-based abstractions will use size parameters, and area-based abstractions will take at least one area as a parameter, but not usually any size parameters. However, if the situation is examined more closely, it appears that stack- and area-based abstraction will always be different abstractions, so interchangeability of them is not really desirable. For example, stack-based abstractions will always be bounded, whereas area-based types will usually be unbounded. Often just this difference is enough to cause the functionalities of operations to be defined differently for stacks as opposed to areas. Another difference is that arrays will be used to represent lists in stack-based abstractions, but true linked lists with pointers will be used in areas. This matter of bounded vs. unbounded abstractions needs further research.

5.11. Summary

We have presented areas and pointers, features that add dynamic storage allocation and list processing capabilities to ASBAL without requiring garbage collection or great run-time overhead. Our pointers are *safe*: they may never point to garbage. Pointer safety is guaranteed by compile-time checking which prevents following any dangling references. We extended our aliasing detection and parameter mechanisms for areas, and discussed a variety of possible implementations for areas. Lastly, we presented three programming examples; two were new implementations of previous examples, and one was a new cluster. We believe that the concepts behind our areas may be useful in other languages besides ASBAL.

6. Summary and Conclusions

We have designed a programming language incorporating abstract data types that does not require garbage collection. Our approach was to use CLU as a basis and change or extend it as needed. The major changes were to the underlying semantic model of computation. We formulated a new object-oriented semantics that preserves many of the object-oriented concepts of CLU, although not in their ideal form. These changes were necessary to obviate the need for garbage collection.

The major extensions were:

(1) Selectors - a mechanism for accessing objects contained within other objects;

(2) Size parameters - a feature allowing control over the size of variables, but handling size automatically where control is not needed; size parameters are essential where object sizes must be bounded at creation;

(3) Areas and pointers - a mechanism for dynamic storage allocation and manipulation of list structures within the framework of the stack.

Of the three extensions, two are just generalizations of commonly accepted ideas, from their present use to the realm of abstract data types. Selectors generalize record and array component access, and size parameters generalize array bounds and dope vectors.

Areas extend the language in quite a different direction. They are an orthogonal addition to the basic ASBAL presented in Chapters 2 to 4. However, areas were added so easily because the object-oriented semantics of the earlier chapters was designed taking areas into account. For example, if we were to delete areas from ASBAL, the copy problem would not arise. Without the copy problem we might be able to have a system defined copy for all types. In the absence of areas the whole semantic model might be significantly different. The area mechanism was largely inspired by the collections of Euclid [Lampson77].

6.1. Suggestions for Further Research

There are many areas of possible further investigation related to the design of languages supporting abstract data types. First, let us describe a few concerning ASBAL directly. One obvious extension of our work is to implement the language we have designed. An implementation would give direct evidence of whether features we claim are good really are worthwhile and efficient enough in practice. We believe that the mechanisms for returning and size parameters are the most questionable. These two mechanisms were the hardest to design, and are still not completely satisfactory. A redesign of these parts by other researchers might be worthwhile.

A more ambitious undertaking would be extensions to ASBAL for systems programming, although any implementation would need to extend the language some. Here are some systems programming features that might be added to ASBAL.

(1) machine-oriented types (such as words, bytes, bit strings, addresses, etc.);

(2) type checking loopholes to allow certain unsafe operations to be performed;[1]

(3) controlled excursions into assembly language for other unsafe operations, such as control of input/output devices and special hardware (e.g., cache memory), process swapping, and other features for building higher level parallel programming constructs;

(4) user-written storage management packages, possibly in the form of new implementations of the area type.

Another suggestion for further investigation is incorporation of our area and pointer mechanism in other languages. We believe our scheme has merit independent of ASBAL, and

---

1. For example, memory allocation involves the unsafe and type-incorrect conversion of a block of memory words to an arbitrary type. Much work remains to be done on the question of how often and how locally such type-checking loopholes must be used, if at all. See Euclid [Lampson77] for one technique of bypassing type checking. The mechanism presented there is simple but inadequate, because there is not enough control over which programs may use it, and how they use it.

it would be interesting to see if it did have applications elsewhere. Comparison of our area mechanism with its parent, Euclid's collection mechanism, might also be worthwhile. The size parameter scheme could also be used in other language designs, since it is a fairly straightforward generalization of the dope vector concept.

A different line of research is to design a language with the same goals as ASBAL, but using static rather than stack allocation of storage for objects. This might involve a synthesis of the CLU cluster and the operating system concept of type managers. Each type manager would statically possess storage and would allocate and free its own objects within that storage.[1] The user's variables in the stack would contain only typed object references, which would be interpreted only by the appropriate type manager. Hence each type manager would distribute only references to its objects; the objects would be kept in its private storage. Because the references would be interpreted only by the type manager distributing them, they could take any form convenient for the type manager.

A static allocation scheme might work very well, particularly for systems programming languages, and could be simpler than ASBAL. It would also look more like CLU since object references are used, and sharing could easily be allowed. A major problem is freeing the storage used by inaccessible objects. Another difficulty would be parameterizing type managers.[2] Still, we believe the type manager approach is quite promising, and might be developed into a simpler and more practical language than ASBAL.

6.2. Conclusions

We believe we were successful in designing a language with abstract data types that does not require garbage collection. And we believe the language is built on a firm philosophical basis, which leads to a consistent design.

---

1. The allocation of storage to the type managers might not be completely static, but would be simple; the type managers would have complete responsibility for the allocation and management of objects of their type.

2. It might be hard to devise a scheme in which the type manager for queues (say) could deal with queues of any type. Likewise, it might be hard, or inefficient, to use a separate type manager for queues of each type.

ASBAL does not have the elegance of CLU. But we did not expect it would. There appears to be a trade-off between elegance on the one hand and efficiency on the other. CLU's semantics achieve elegance through the use of a simple and powerful semantic model, which unfortunately requires fairly complex run-time support. We have traded away some of that elegance for a more efficient run-time mechanism. However, we have tried not to compromise some more important aspects of CLU. Our prejudice has been toward a completely type-safe language, type-safety being a key to the abstract data type methodology.

If ASBAL does not have CLU's elegance, neither does it have the simplicity of traditional languages, for example Pascal. In fact, CLU is more complex than Pascal (and similar languages), but more because it has a parameter mechanism than because of abstract data types.[1] Yet ASBAL is more complex than CLU. We believe the source of ASBAL's complexity is the constraint of running within a stack, and not using an automatically managed heap. In a sense, we have built ASBAL on an inappropriate foundation, but one forced on us by our requirements.

ASBAL represents a synthesis of ideas from several languages, and several semantic models. We feel the synthesis was profitable, and hope that our work may suggest and encourage more investigation in the area.

---

1. Pascal has been criticized on the grounds that it is *too* simple in this respect: no Pascal program can deal with arrays of any size.

I. Syntax of ASBAL

Here we present the full, context-free syntax of ASBAL. The metalanguage used is an extension of the usual Backus-Naur form. It has several special symbols, and their meaning is as follows:

[ ] - enclose optional parts of a production;

{ } - enclose an item which may be repeated any number of times, including zero;

| - separates choices, thus ' a | b ' generates either ' a ' or ' b ';

( ) - are used to group items and eliminate ambiguity;

=> - used to separate the left-hand side of each production (a nonterminal) from the right-hand side (the string it generates).

Nonterminal symbols are represented by lower case identifiers. All other symbols are terminal (excepting the special metalanguage symbols).

I.1. Formal Syntax

I.1.1. Modules

program => module ; { module ; }

module => cluster | procdef | iterdef | seldef

cluster => id = cluster [fcparms] is ids ; [restrictions ;]
{ equate ; } rep = type ; { equate ; }
clustermodule { clustermodule } end [id] ;

clustermodule => procdef | iterdef | seldef

procdef => id = proc [fparms] fargs [frets] [faigs] ;
[restrictions ;] body end [id] ;

iterdef => id = iter [fparms] fargs [fyids] [faigs] ;
[restrictions ;] body end [id] ;

seldef => id = selector [fparms] ([ids : qtype { , ids : qtype }])
of stype from id : qtype [faigs] ; [restrictions ;] body end [id] ;

I.1.2. Parameters and Restrictions

fparms => [fparmitem { , fparmitem }]

fcparms => [[fparmitem { , fparmitem }] [; ids]]

fparmitem => ids : (int | bool | char | type | area)

The above three productions are for formal parameters (to all definitions except clusters), formal parameters to clusters, and the items in formal parameter lists.

aparms => [[aparm { , aparm }] [; exp { , exp }]]

aparm => exp | qtype

The productions for *aparms* and *aparm* are for actual parameters, which may be expressions, or type specifications (with ?id's and *'s in them).

restrictions => where restriction { and restriction } and [where]

restriction => id has restrict { , restrict }

restrict => ids : (ptype | itype | selftype)

I.1.3. Arguments, Returns, Yields, and Signals

fargs => ([faitem { , faitem }])

faitem => (var | const) ids : qtype { , ids : qtype }

frets => returns ([ids : stype { , ids : stype }])

fylds => yields ([fyldItem { , fyldItem }])

fyldItem => (var | const) stype { , stype }

fsigs => signals (fsigItem { , fsigItem })

fsigItem => id [(stype { , stype })]

The above productions are for formal argument lists, returns lists, yields lists, and signals lists. They are used mainly in module definitions.

I.1.4. Statements

body => { equate ; } { statement ; }

equate => id = exp ;

statement => decl
| assign

```
        | if
        | while
        | for
        | with
        | except
        | begin
        | return
        | yield
        | select
        | signal
        | invoke
        | tagcase
        | break
        | new

decl    => var ids : qtype { , ids : qtype } := exp { , exp }

        | var ids : type { , ids : type }

        | const ids : qtype { , ids : qtype } = exp { , exp }
```

In the first and third productions for *decl*, the number of identifiers on the left must equal the number of expressions on the right.

```
assign  => ids := exp { , exp }
```

There must be either one expression, or as many expressions as there are identifiers.

```
if      => if exp then body { elseif exp then body } [ else body ] end [ if ]

while   => while exp do body end [ while ]

for     => for fordecl { , fordecl } in invoke do body end [ for ]

        | for [ ids ] in invoke do body end [ for ]
```

fordecl -> (var | const) ids : qtype

with -> with (var | const) id := exp do body end [| with]

except -> statement except { whenarm ; } [othersarm ;] end

There must be at least one whenarm or othersarm present.

whenarm -> when ids [fargs] : statement
| when ids (*) : statement

In the first production, all the exceptions named must send exactly the same types of objects in the same order. The second production is used for throwing away the objects sent along, and the number and types of the objects need not be the same for all exceptions listed.

othersarm -> others (const id : qtype) : statement
| others (*) : statement

The first production's object is a string (so the qtype must resolve to string), and is the name of the exception signalled.

begin -> begin body end

return -> return

yield -> yield [([exp { , exp }])]

select -> select exp

signal -> signal id [([exp { , exp }])]

invoke -> exp ([exp { , exp }])

tagcase -> tagcase exp in tagarm { tagarm ; } end [tagcase]

tagarm -> tag ids [((var | const) id : qtype)] : statement
| others : statement

There may be only one others arm per tagcase statement. All tags must be accounted for in each tagcase statement. All tags named on the same arm must be for the same type, and the type of the locally declared identifier must match that type.

break => break

new => new id : area = exp

The expression must be an invocation of areanew.

Several of the above statements are allowed only in particular contexts. The return statement is legal only in procedures and iterators; yield is legal only in iterators; select is legal only in selectors; and break is allowed only in for and while loops.

I.1.5. Expressions

exp => exp bop exp
| uop exp
| (exp)
| literal
| selexp
| invoke
| qtype $ id [sparms]
| up
| down

The last four productions of *exp* need explaining; they are for routines. The special routines up and down are allowed only in clusters and convert between the abstract and rep types.

selexp => id
| exp . id [(exp { , exp })]
| exp [exp]
| exp ↑

These forms are (in order): a variable, selection, array indexing, and pointer following.

bop => ?
| **
| * | / | //
| + | - | ||
| < | <= | = | >= | > | ~< | ~<= | ~= | ~>= | ~>
| & | cand
| | | cor

The operators on each line have the same precedence. The operators in the first line bind most tightly, those in the second less tightly, etc., so the last line binds least tightly. For all operators except **, 'x op y op z' means '(x op y) op z'; 'x ** y ** z' means 'x ** (y ** z)'.

uop => - | ~

literal => intlit | charlit | strlit | boollit | nullit | ptrlit
| qtype $ [exp : exp { , exp }]
| qtype $ [exp .. exp : exp]
| qtype $ { ids : exp { , ids : exp } }

The second and third lines are for the array constructor, which has two forms. The '[exp_0 : exp_1 , exp_2 , ..., exp_n]' form means an array with low bound exp_0, and n elements, exp_1 through exp_n, in increasing order. The '[exp_0 .. exp_1 : exp_2]' form means an array with lower bound exp_0, upper bound $max(exp_0 - 1, exp_1)$, and all elements copies of exp_2. The last production is for record constructors.

boollit => true | false

nullit => nil

ptrlit => nilptr

I.1.6. Types

```
type        => int
             | bool
             | char
             | null
             | area
             | string [ ; exp ]
             | array [ type ; exp , exp ]
             | record [ ids : type { , ids : type } ]
             | oneof [ ids : type { , ids : type } ]
             | id [ eparms ]
             | ptype
             | itype
             | seltype
             | cvt [ ; exp ]
             | ptr [ id , qtype ]

ptype       => proctype fpargs [ fprets ] [ fsigs ]

itype       => itertype fpargs fyids [ fsigs ]

seltype     => seltype [ fpargs ] of stype from qtype [ fsigs ]

fpargs      => ( [ fpargitem { , fpargitem } ] )

fpargitem   => ( var | const ) qtype

fprets      => returns ( [ stype { , stype } ] )

ids         => id { , id }
```

The productions of *type* are for those positions where a v-typespec is required.

I.1.7. Star Types

```
stype       => int
             | bool
             | char
             | null
             | area
             | string [ [ ; sparm ] ]
             | array [ stype [ ; sparm , sparm ] ]
             | record [ ids : stype { , ids : stype } ]
             | oneof [ ids : stype { , ids : stype } ]
             | id [ sparms ]
             | ptype
             | itype
             | selftype
             | cvt [ [ ; sparm } , sparm { ] ]
             | rep [ [ ; sparm } , sparm { ] ]
             | ptr [ id , qtype ]
sparms      => [ [ sparm { , sparm } ] [ ; sparm { , sparm } ] ]
sparm       => exp | *
```

An *stype* is used for variable-type parameters.

I.1.8. Question Mark or Star Types

```
qtype       => int
             | bool
             | char
             | null
             | area
```

| string [[; qparm]]

| array [qtype [; qparm , qparm]]

| record [ids : qtype { , ids : qtype }]

| oneof [ids : qtype { , ids : qtype }]

| id [qparms]

| ptype

| itype

| seltype

| cvt [[; qparm { , qparm }]]

| rep [[; qparm { , qparm }]]

| ptr [id , qtype]

qparms => [[qparm { , qparm }] [; qparm { , qparm }]]

qparm => exp | ? id | *

The nonterminal qtype expands to v*?-typespecs.e's and ?id's.

I.2. Syntactic Sugars

There are quite a few syntactic forms which are 'sugars' for normal invocations. These forms are cataloged below; x, y, and z are expressions, n is an identifier, and T is the syntactic type of x (i.e., its type as declared in a program or determined by the compiler).

Sugar	Expansion
x.n	T$n(x)
x.n := z	T$put_n(x, z)
x[y]	T$elementt(x, y)
x[y] := z	T$store(x, y, z)
x ** y	T$power(x, y)
x * y	T$mul(x, y)
x / y	T$div(x, y)
x // y	T$mod(x, y)
x + y	T$add(x, y)
x - y	T$sub(x, y)
x || y	T$concat(x, y)
x < y	T$lt(x, y)
x <= y	T$le(x, y)
x = y	T$equal(x, y)
x >= y	T$ge(x, y)
x > y	T$gt(x, y)
x ~< y	~(x < y)
x ~<= y	~(x <= y)
x ~= y	~(x = y)
x ~>= y	~(x >= y)
x ~> y	~(x > y)
x & y	T$and(x, y)
x | y	T$or(x, y)
- x	T$minus(x)
~ x	T$not(x)

I.3. Reserved Words

and	char	else	from	iter	of	record	seltype	then	when
area	cluster	elseif	has	itertype	oneof	rep	signal	to	where
array	const	end	if	new	others	return	signals	true	while
begin	cor	except	in	nil	proc	returns	string	type	with
bool	cvt	false	int	niltr	proctype	select	tag	up	yield
break	do	for	is	null	ptr	selector	tagcase	var	yields
cand	down								

I.4. Terminal Symbols

```
id          => alpha { alpha | digit }

alpha       => letter | _

letter      => A | ... | Z | a | ... | z

digit       => 0 | ... | 9

intlit      => digit { digit }

charlit     => ' ( char_rep | " ) '

strlit      => " { char_rep | ' } "

char_rep    => printing | \ special

printing    => any ASCII character such that 37(8) < octal value < 177(8)

special     => '          % represents '
             | "          % represents "
             | \          % represents \
             | n          % represents NL (newline)
             | t          % represents HT (horizontal tab)
             | p          % represents FF (form feed)
             | b          % represents BS (backspace)
             | r          % represents CR (carriage return)
             | v          % represents VT (vertical tab)
             | octal octal octal

octal       => 0 | ... | 7
```

II. Basic Data Types of ASBAL

Here we detail the operations of the basic types of ASBAL. Let us first describe the special notations used. The arguments to operations (the actual objects, not the syntactic expressions) are called 'arg1', 'arg2', etc., or just 'the argument' if there is only one. If an operation signals 'foo', we say that foo occurs. The 'type$' part of operation names is dropped where there is no ambiguity. Arithmetic expressions and comparisons contained in the text are to be computed over the domain of all integers, not just the domain of the type int.

Some definitions involve restrictions. If a definition has a restriction, it is either a standard where clause, or of the form

where each T_i has oper_decl$_i$

which is an abbreviation for

where T_1 has oper_decl$_1$, ..., T_n has oper_decl$_n$

Several definitions will involve tuples. A tuple is written $\langle a_1, ..., a_n \rangle$. The a_i are called the components of the tuple, and a_i is called the ith component. A tuple with n components is called an n-tuple. We also define the following operations on tuples:

$Size(\langle a_1, ..., a_n \rangle) = n$

$A = B$ iff $(Size(A) = Size(B)) \wedge (\forall i | 1 \le i \le n)[a_i = b_i]$

$\langle a, ..., b \rangle \,||\, \langle c, ..., d \rangle = \langle a, ..., b, c, ..., d \rangle$

$Front(\langle a, ..., b, c \rangle) = \langle a, ..., b \rangle$

$Tail(\langle a, b, ..., c \rangle) = \langle b, ..., c \rangle$

$Tail^0(A) = A$ and $Tail^{i+1}(A) = Tail(Tail^i(A))$

$Occurs(A, B, i) = (\exists C, D)[(B = C \,||\, A \,||\, D) \wedge (Size(C) = i - 1)]$

Lastly, we say tuple A occurs at index I in tuple B if Occurs(A, B, I) holds.

II.1. Nulls

There is only one, immutable object of type null, denoted by nil.

equal: proctype(const null, null) returns (bool)

Always returns true.

copy: proctype(const null) returns (null)

The obvious copy.

II.2. Booleans

There are two, immutable objects of type bool, denoted by true and false. They represent the logical truth values.

and: proctype(const bool, bool) returns (bool)
or: proctype(const bool, bool) returns (bool)
not: proctype(const bool) returns (bool)

The standard logical functions.

equal: proctype(const bool, bool) returns (bool)

Equal returns true iff its arguments are the same.

copy: proctype(const bool) returns (bool)

Copy simply copies its argument.

II.3. Integers

Objects of the type int are immutable and represent a subrange of the mathematical integers. The subrange (which may differ with each implementation) is a closed interval [Int_Min, Int_Max], where Int_Min $\leq -2^{15}+1$ and Int_Max $\geq 2^{15}-1$. An overflow exception is signalled by an operation if the result would lie outside this interval.

add: proctype(const int, int) returns (int) signals (overflow)
sub: proctype(const int, int) returns (int) signals (overflow)
mul: proctype(const int, int) returns (int) signals (overflow)

The standard integer operations.

minus: proctype(const int) returns (int) signals (overflow)

Minus returns the negative of its argument.

div: proctype(const int, int) returns (int) signals (zero_divide, overflow)

Div computes the quotient of arg1 and arg2, i.e., the integer q such that $(\exists r | 0 \leq r < |arg2|)$ [arg1 = q * arg2 + r]. Zero_divide occurs if arg2 = 0.

power: proctype(const int, int) returns (int) signals (negative_exponent, overflow)

This computes arg1 raised to the arg2 power. Power(0, 0) = 1. Negative_exponent occurs if arg2 < 0.

mod: proctype(const int, int) returns (int) signals (zero_divide, overflow)

This computes the integer remainder of dividing arg1 by arg2; i.e., the result is arg1 - arg2*div(arg1, arg2). Zero_divide occurs if arg2 = 0.

from_to_by: itertype(const int, int, int) yields (const int) signals (zero_step)

This iterator yields, in succession, arg1, arg1 + arg3, arg1 + 2 * arg3, etc., until the next value to be yielded, x, satisfies (x > arg2 ∧ arg3 > 0) ∨ (x < arg2 ∧ arg3 < 0). Zero_step occurs if arg3 = 0.

lt: proctype(const int, int) returns (bool)
le: proctype(const int, int) returns (bool)
equal: proctype(const int, int) returns (bool)
ge: proctype(const int, int) returns (bool)
gt: proctype(const int, int) returns (bool)

The standard ordering relations.

copy: proctype(const int) returns (int)

The obvious copy operation.

II.4. Characters

The objects of type char are immutable, and represent characters. Every implementation is assumed to provide at least 128 characters, but no more than 512. Character are numbered from 0 to some Char_Top, and the numbering defines the ordering for the character type. The first 128 characters are the ASCII characters in their standard order.

i2c: proctype(const int) returns (char) signals (illegal_char)

I2c returns the character numbered arg1 in the numbering of characters. Illegal_char occurs iff the argument is not in the range [0, Char_Top].

c2i: **proctype(const char) returns (int)**

Returns the number corresponding to its argument.

lt: **proctype(const char, char) returns (bool)**
le: **proctype(const char, char) returns (bool)**
equal: **proctype(const char, char) returns (bool)**
ge: **proctype(const char, char) returns (bool)**
gt: **proctype(const char, char) returns (bool)**

The ordering relations consistent with the numbering of characters.

copy: **proctype(const char) returns (char)**

The obvious copy.

II.5. Strings

Strings are immutable objects. Each string represents a tuple of characters. The ith character of the string is the ith component of the tuple. The size of a string must be a legal integer; if it is not, then a failure exception is signalled. Furthermore, a variable declared string[n] must be able to store strings whose size does not exceed n, and may possibly store larger strings.

size: **proctype(const string) returns (int)**

Returns the size of the tuple representing its argument.

indexs: **proctype(const string, string) returns (int)**

The operation returns the least index at which arg2 occurs in arg1. (Notice that this means 1 is returned if arg1 is the 0-tuple.) If arg2 does not occur in arg1, then 0 is returned.

indexc: **proctype(const string, char) returns (int)**

Indexc returns the least index at which the 1-tuple <arg2> occurs in arg1. If <arg2> does not occur in arg1, then 0 is returned.

c2s: **proctype(const char) returns (string)**

Returns the string represented by the 1-tuple <arg1>.

concat: **proctype(const string, string) returns (string)**

Concat returns the string for which arg1 || arg2 is the representation.

append: **proctype(const string, char) returns (string)**

This operation returns the string represented by arg1 || <arg2>.

fetch: **proctype(const string, int) returns (char) signals (bounds)**

Fetch returns the $arg2^{th}$ character of arg1. Bounds occurs if (arg2 < 1) ∨ (arg2 > size(arg1)).

substr: **proctype(const string, int, int) returns (string) signals (bounds, negative_size)**

Substr returns the string represented by the tuple of size min(arg3, size(arg1) - arg2 + 1) which occurs at index arg2 in arg1. Bounds occurs if (arg2 < 1) ∨ (arg2 > size(arg1) + 1). Negative_size occurs if arg3 < 0.

rest: **proctype(const string, int) returns (string) signals (bounds)**

Equivalent to substr(arg1, arg2, size(arg1)), i.e., the result is $Tail^{arg2-1}(arg1)$.

s2ac: **proctype(const string) returns (array[char])**

This operation creates a new array, the elements of which are the characters of the argument. The low bound of the array is 1, and the size of the array is size(arg1). The ith element of the array is the ith character of the string.

ac2s: **proctype(const array[char]) returns (string)**

Ac2s is the inverse of s2ac. The result is the string with characters in the same order as in its argument. Thus the ith character of the result is the $(i + low(arg1) - 1)^{th}$ element of the argument.

chars: **itertype (const string) yields (const char)**

This iterator yields, in order, each character of its argument.

```
lt:     proctype(const string, string) returns (bool)
le:     proctype(const string, string) returns (bool)
equal:  proctype(const string, string) returns (bool)
ge:     proctype(const string, string) returns (bool)
gt:     proctype(const string, string) returns (bool)
```

These use the usual lexicographic ordering based on the ordering for characters. The lt operation is equivalent to the following procedure:

```
lt = proc(const x, y: string) returns (less: bool);
   const size_x, size_y: int = string$size(x), string$size(y);
   var min: int;
   if size_x <= size_y
      then min := size_x;
      else min := size_y;
      end if;
   for const i: int in int$from_to_by (1, min, 1) do
      if x[i] < y[i]
         then less := true; return;
         end if;
      end for;
   less := (size_x < size_y);
   end lt;
```

```
copy:   proctype(const string) returns (string)
```

The obvious copy.

II.6. Arrays

The array type generator defines an infinite class of types. For every type T there is a type array[T]. Array objects are mutable. The state of an array object consists of:

1. an integer Low, called the low bound, and
2. a tuple Elts of objects of type T, called the elements.

We also define Size = Size(Elts), and High = Low + Size - 1. We want to think of the components of Elts as being numbered from Low, so we define the array_index of the ith component to be (i - Low + 1). Each array object is of bounded size, in two ways. First, its Size, Low, and High must all be legal integers. Secondly, Low and High are bounded by the size of the variable containing the array object. Any attempts to violate these restrictions result in a failure exception: failure("illegal_array") in the first case, and failure("variable overflow") in the other. A variable (or object component) of type array[T; l, h] must be able to contain array objects with

Low ≥ l and High ≤ h; it may be able to contain larger arrays. If an array is assigned to a variable, grown with addh or addl, or shifted with set_low, such that the limits of the variable would be exceeded, then failure("variable overflow") is signalled, as mentioned above.

For an array A, we should write Low_A, etc., to refer to the state of that object, but subscripts will be dropped where the association is clear.

Note that for all array operations, if an exception occurs, (other than failure), then the states of the arguments are unchanged from those at the time of invocation.

We use the abbreviation AT for the type array[T].

create: proctype(const int) returns (AT)

This returns an array for which Low is arg1 and Elts is the 0-tuple.

new: proctype() returns (AT)

Equivalent to create(1).

low: proctype(const AT) returns (int)
high: proctype(const AT) returns (int)
size: proctype(const AT) returns (int)

These operations return Low, High, and Size, respectively.

set_low: proctype(var AT, const int)

Set_low makes Low equal to arg2. This operation may involve physically shifting the elements of the array in storage. However, block move instructions available on many machines can be used to help implement set_low.

trim: proctype(var AT, const int, int) signals (bounds, negative_size)

This operation makes Low equal to arg2, and makes Elts equal to the tuple of size min(arg3, High' - arg2 + 1) which occurs in Elts' at index arg2 - Low' + 1.[1] That is, every element with array_index less than arg2, or greater than or equal to arg2 + arg3, is removed. Bounds occurs if (arg2 < Low') ∨ (arg2 > High' + 1). Negative_size occurs if arg3 < 0.

---

1. Elts', Low', etc., refer to the state prior to invoking the operation.

fill: proctype(const int, int, T) returns (AT) signals (negative_size)
where T has copy: proctype(const T) returns (T)

Fill returns an array for which Low is arg1, and Elts is a (max(0, arg2))-tuple in which every component is a copy of arg3. It calls T$copy max(0, arg2) times to get the elements, in order. Negative_size occurs if arg2 < 0.

fetch: seltype(int) of T from AT signals (bounds)

Fetch selects the element of arg1 with array_index arg2. Bounds occurs if (arg2 < Low) v (arg2 > High).

bottom: seltype() of T from AT signals (bounds)
top: seltype() of T from AT signals (bounds)

These operations select the elements with array_index's Low and High, respectively. Bounds occurs if Size = 0.

store: proctype(var AT, const int, T) signals (bounds)
where T has copy: proctype(const T) returns (T)

Store makes Elts a new tuple which differs from the old in that arg3 is the element with array_index arg2. T$copy is used to copy the argument. Bounds occurs if (arg2 < Low) v (arg2 > High).

addh: proctype(var AT, const T)
where T has copy: proctype(const T) returns (T)

This operation makes Elts the new tuple Elts' * <arg2>. T$copy is used to create the new component.

addl: proctype(var AT, const T)
where T has copy: proctype(const T) returns (T)

This operation makes Low equal to Low' - 1, and Elts the tuple <arg2> * Elts'. T$copy is used to create the new component. (Decrementing Low keeps the array_index's of the previous elements the same.)

remh: proctype(var AT) returns (T) signals (bounds)

permuted.) Records are mutable objects. The state of a record of type record[Id_1: T_1, ..., Id_n: T_n] is an n-tuple. The ith component of the tuple is of type T_i. The ith component is also called the Id_i-component. We abbreviate record[Id_1: T_1, ..., Id_n: T_n] by RT below.

create: proctype(const T_1, ..., T_n) returns (RT)
where each T_i has copy: proctype(const T_i) returns (T_i)

This operation returns a new record with the tuple <arg1, ..., argN> as its state. It uses $T_i$$copy to copy arg_i. Create is not available to the user, but its use is implicit in the record constructor.

Id_i: seltype() of T_i from RT

This operation selects the Id_i-component of its argument. There is an Id_i operation for each Id_i.

Put_Id_i: proctype(var RT, const T_i)
where T_i has copy: proctype(const T_i) returns (T_i)

This operation makes the state of arg1 a new tuple which differes from the old in that its Id_i-component is a copy of arg2 made using $T_i$$copy. There is a put_Id_i operation for each Id_i.

equal: proctype(const RT, RT) returns (bool)
where each T_i has equal: proctype(const T_i, T_i) returns (bool)

This operation compares the tuples of arg1 and arg2 component by component using $T_i$$equal for the Id_i-component. If all the comparisons return return true, the result is true; otherwise the result is false.

copy: proctype(const RT) returns (RT)
where each T_i has copy: proctype(const T_i) returns (t_i)

This operation returns a record whose state is a copy of the state of the argument. This copy is made using $T_i$$copy for the Id_i-component.

II.8. Oneofs

The oneof type generator defines an infinite class of types. For every tuple of id/type pairs $<(Id_1, T_1), ..., (Id_n, T_n)>$, where all the id's are distinct and in lexicographic order, there is a type oneof[Id_1: T_1, ..., Id_n: T_n]. (The user may write this type with the pairs permuted.) Oneof objects are immutable. Each oneof object is represented by a pair (Id_i, X), where X is of type T_i. The Id_i part of the pair is called the tag, and X is called the value. We abbreviate oneof[Id_1: T_1, ..., Id_n: T_n] to OT below.

make_Id_i: proctype(const T_i) returns (OT)
where T_i has copy: proctype(const T_i) returns (T_i)

This operation returns the oneof object for the pair (Id_i, arg1), using $T_i$$copy. There is a make_Id_i operation for each Id_i.

is_Id_i: proctype(const OT) returns (bool)

This operation returns true iff the tag of arg1 is Id_i. Its use is implicit in the tagcase statement. There is an is_Id_i operation for each Id_i.

value_Id_i: seltype() of T_i from OT signals (wrong_tag)

If the argument has tag Id_i, this selects the value part of the argument. Wrong_tag occurs if the tag is not Id_i. This operation is used implicitly by the tagcase statement. There is a value_Id_i for each Id_i.

equal: proctype(const OT, OT) returns (bool)
where each T_i has equal: proctype(const T_i, T_i) returns (bool)

If the tags of the arguments are different, then false is returned. If the tags are both Id_i, then the result is $T_i$$equal applied to the value parts of the arguments.

copy: proctype(const OT) returns (OT)
where each T_i has copy: proctype(const T_i) returns (T_i)

This operations returns a oneof object with the same tag as its argument, and a value part a copy of the value part of the argument. If the tag is Id_i, then the copy is made using $T_i$$copy.

II.9. Pointers

The pointer type generator defines an infinite class of types. For each area A, and each type T, ptr[A, T] is a type. The representation of pointers is not defined explicitly, but implicitly through the behavior of pointer objects. Pointer objects are immutable. We abbreviate ptr[A, T] to PT below.

nilptr: **proctype() returns (PT)**

This operation returns a pointer that points to no object. Therefore, it is equal only to other null pointers of the same type, and cannot be dereferenced.

alloc: **proctype(const T) returns (PT) signals (failure(string))**
where T has copy: proctype(const T) returns (T)

This operation creates a copy of arg1 in area A, returning a pointer to the newly created object. The copy is made using T$copy. Failure occurs if the area cannot contain the new object; the string signalled is "area out of memory".

deref: **seltype() of T from PT signals (bad_pointer)**

This operation "follows" a pointer to the object pointed at. Bad_pointer occurs if the null pointer is dereferenced.

equal: **proctype(const PT, PT) returns (bool)**

This operation returns false unless arg1 and arg2 point to the same object, or arg1 and arg2 are both null pointers.

copy: **proctype(const PT) returns (PT)**

This operation returns a pointer equal to its argument. That is, the result points to the same object as the argument.

II.10. Areas

An area object is used for the dynamic allocation of other objects.

new: **proctype(const string, int) returns (area) signals (bad_arguments)**

This operation returns a new area. Arg1 is used to describe what sort of area management scheme is desired, and arg2 is for size. The meaning of both arguments is implementation dependent.

II.11. Procedures, Iterators, and Selectors

For each procedure, iterator, and selector type there are three operations: create, copy, and equal. Create is not available to the user; its use is implicit in the compiler and run-time system. Copy presumably does not copy the object code of a routine, but merely a descriptor. Equal does not mean "computationally equivalent". Two routines derived from the same module with the same parameters (if any) are considered equal; for this purpose different operations of a cluster are considered to be different modules.

References

Baker77. Baker, Henry G., Jr., "List Processing in Real Time on a Serial Computer", MIT AI Lab. Working Paper 139, February 2, 1977.

Barth77. Barth, Jeffrey M., "Shifting Garbage Collection Overhead to Compile Time", pp. 513-518, CACM, Vol. 20, No. 7, July 1977.

Bishop77. Bishop, Peter B., "Computer Systems with a Very Large Address Space and Garbage Collection", Mass. Inst. of Tech., Lab. for Comp. Sci., TR-178, May 1977.

Branquart70. Branquart, P., and Lewi, J., "A Scheme of Storage Allocation and Garbage Collection for Algol 68", in Algol 68 Implementation, Proc. IFIP Working Conf. on Algol 68 Impl., Munich, July 1970.

Dahl66. Dahl, O. J., and Nygaard, K., "Simula - an Algol-based simulation language", pp. 671-678, CACM, Vol. 9, No. 9, September 1966.

Deutsch76. Deutsch, L. Peter, and Bobrow, Daniel G., "An Efficient, Incremental, Automatic Garbage Collector", pp. 522-526, CACM, Vol. 19, No. 9, Sept. 1976.

Geschke75. Geschke, Charles M., and Mitchell, James G., "On the Problem of Uniform References to Data Structures", pp. 207-217, IEEE Trans. on Soft. Eng., Vol. SE-1, No. 2, June, 1975.

Goodenough75. Goodenough, John B., "Exception Handling: Issues and a Proposed Notation", publication 550-1, Sof-Tech, Inc., Waltham, MA, April 1975.

Gries77. Gries, David, and Gehani, Narain, "Some Ideas on Data Types in High-Level Languages", pp. 414-420, CACM, Vol. 20, No. 6, June 1977.

Guttag75. Guttag, John V., "The Specification and Application to Programming of Abstract Data Types", Computer Systems Research Group TR CSRG-59, Univ. of Toronto, Sept. 1975.

IBM70. IBM, PL/1 Language Specifications, IBM Order No. GY33-6003-2, 1970.

Lampson77. Lampson, B. W., et. al., Report on the programming language Euclid, ACM SIGPLAN Notices, Vol. 12, No. 2, February, 1977.

Liskov77. Liskov, B., et. al., "Abstraction Mechanisms in CLU", pp. 564-576, CACM, Vol. 20, No. 8, August 1977.

Liskov75. Liskov, B., and Zilles, S., "Specification Techniques for Abstract Data Types", pp. 7-19, IEEE Trans. on Soft. Eng., Vol SE-1, No. 1, March 1975.

Ross69. Ross, D. T., "Uniform Referents: An Essential Property for a Software Engineering Language", pp. 91-101, Software Engineering, J. T. Tou, ed., vol. 1, Academic Press, New York.

Scheifler77. Scheifler, Robert W., "Type Parameters and Infinite Recursion", CLU Design Note No. 65, MIT Lab. for Comp. Sci., Sept. 16, 1977.

Steele75. Steele, Guy L., Jr., "Multiprocessing Compactifying Garbage Collection", pp. 495-508, CACM, Vol. 18, No. 9, Sept. 1975. (Steele Corrigendum, p. 354, CACM, Vol. 19, No. 6, June 1976.)

Wijngaarden77. van Wijngaarden, A., et. al., "Revised Report on the Algorithmic Language Algol 68", ACM Sigplan Notices, Vol. 12, No. 5, May 1977, pp. 1-70.

Wirth71. Wirth, N., "The Programming Language Pascal", Acta Informatica 1, pp. 35-63, 1971.

Wulf76a. Wulf, W. A., London, Ralph L., Shaw, Mary, "Abstraction and Verification in Alphard: Introduction to Language and Methodology", Carnegie-Mellon Technical Report, June 14, 1976.

Wulf76b. Wulf, W. A., London, Ralph L., Shaw, Mary, "Abstraction and Verification in Alphard: Iteration and Generators", Carnegie-Mellon Technical Report, August 20, 1976.

Wulf76c. Wulf, W. A., London, Ralph L., Shaw, Mary, "Abstraction and Verification in Alphard: A Symbol Table Example", Carnegie-Mellon Technical Report, December 29, 1976.

CS-TR Scanning Project
Document Control Form

Date : 10/26/95

Report # LCS-TR-190

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 154 (160-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:
☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-154) UNITED TITLE PAGE, 2-154	
(155-160) SCANCONTROL, COVER, SPINE, TRGTS (3)	

Scanning Agent Signoff:
Date Received: 10/26/95 Date Scanned: 11/17/95 Date Returned: 11/22/95

Scanning Agent Signature: Michael W. Cooper

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 11/17/1995

M.I.T. Libraries
Document Services