I hereby state that this thesis contains only my own original work, except where acknowledgement has been made to the published work of others.

DATA STRUCTURE REPRESENTATION AND TRANSFORMATION

Christopher William Johnson

Thesis submitted for the degree of Doctor of Philosophy

Australian National University

April 1982
I hereby state that this thesis contains only my own original work, except where acknowledgement has been made to the published work of others.

Christopher W. Johnson
ABSTRACT

This work investigates the derivation of representations of abstract data types by means of manually directed transformation of their definitions in a program. The central problem addressed is the form of language facilities needed to provide more than one representation for an abstract type within any particular program. An algorithmic programming language with data encapsulation features covering a broad spectrum of data spaces is described. This Broad Spectrum Language combines the forms needed for an appropriate data space for defining abstract data structures and a machine-oriented data space for their implementation. It contains declarative facilities for the control of shared bindings (dynamic aliasing) and nonexistent bindings at the high level end of the spectrum, and for references and overlaid storage at the other. A "bundle" declarative facility in BSL provides the language framework to maintain assignment compatibility between different representations as they are developed from one abstract definition. The bundle treats assignment for structured data objects as a non-primitive operation, and the interfaces of the operations that implement assignment between different representations of an abstract type are immutable under the applied transformations. A small library of transformations manipulates the program constructs corresponding to data structures and bindings. The transformations are described as procedures in an algorithmic language which is
designed to operate upon a phrase structured parsing of programs in strongly structured languages. The text of the program to be transformed is manipulated in terms of an Extended BNF description of the syntax of its language. One minor group of transformations is automatically triggered by conditions found in the form of the program; the more substantial transformations are initiated manually. Their application in series constitutes a specification for the mechanical derivation of an implementation of a program.
I wish to thank my supervisors, Dr R.B. Stanton and Dr B.P. Molinari (both of the Department of Computer Science, Australian National University), for their guidance in the years I was pursuing this research. Dr Stanton's wide-ranging ideas provided much of the basis of my training in the science of Computer Science, and his encouragement spurred me out of the many ruts and quagmires that beset the path of the inexperienced. Dr Molinari gave time to the reading of my thesis and his constructive suggestions were very valuable.

The Commonwealth of Australia Post Graduate Research Awards and the Australian National University funded this research. The Australian National University Computer Services Centre provided the computer facilities used in the research work.

Colleagues at the Software Technology Centre of the University of York, U.K., helped with suggestions on the presentation of the thesis. The Centre provided facilities for its production.

Bev Johnstone, for her beautiful typing and perseverance with the recalcitrant form of word processor I asked her to use, Megan Cook, for additional typing and for proof reading and editing, Margaret Browne, for proof reading and helpful comments, and Maxine Esau and Dave Hawking, for their help in production, have my grateful thanks.
## CONTENTS

1 Introduction
   1.1 Data structures in programming languages 1
   1.2 The problem addressed by this thesis 4
   1.3 The form of the adopted solution 7
   1.4 Thesis organization 9

2 Literature survey and critique
   2.1 Introduction 11
   2.2 Literature survey 14
   2.3 Critique 34
   2.4 Conclusions: transform a Broad Spectrum Language 45

3 The Broad Spectrum Language
   3.1 Introduction 50
   3.2 Koala - a referenceless language 60
   3.3 Maclan - a machine-oriented language 115

4 Representation by transformation
   4.1 Introduction 135
   4.2 The major transformations 137
   4.3 Transformations in action 142
   4.4 Catalogue of transformations 153
   4.5 Formal description and implementation 180
   4.6 Discussion: transformations and the machine end of BSL 197

5 Multiple representations
   5.1 Introduction 206
   5.2 Type-splitting 209
   5.3 Assignment compatibility between different representations 229

6 Two representations of the bounded stack
   6.1 Bounded stack definition 248
   6.2 Bundle form 250
   6.3 Linked list 253
   6.4 Vector with top 278
   6.5 Alternative transformation paths 290
   6.6 Discussion 293

7 Conclusion
   7.1 Summary 299
   7.2 Conclusions 303
   7.3 Future work 311
   7.4 Conclusion 316

Appendices
   1 Syntax description 317
   2 BSL syntax 319
   3 Xlan description of BSL 325

Bibliography 329
1 INTRODUCTION

1.1 Data structures in programming languages

The form, content and direction of programming languages is a central concern of Computer Science. The history of the design of programming languages has resulted from the struggle between the two nearly opposing forces of increasing abstraction and maintaining efficiency of implementation since the early days of the electronic computer. In the past decade awareness of the "software crisis" has added the third force of a concern with program reliability, under the name of Software Engineering, to modify these pulls with pressure away from increasing language complexity.

Definition facilities for data structures in particular must meet requirements of providing both powerful abstractions and efficient realizations. The relative importance of the two is undecided ("Efficiency per se is not terribly important [but] inefficiency becomes an excuse to build a poorly structured program" [Wulf 1972]; "a well-designed [language] feature ... does not imply any unexpected, hidden inefficiencies" [Wirth 1974]). A compromise has generally been effected with a single facility defining both the abstract form and operations of data and the form of its implementation: reference variables, fixed-length strings, limitations on FLEX arrays in Algol 68 are examples.

The more powerful constructors of abstract data types make the same compromise. Interest in data structure facilities has come to concentrate on the module (or capsule, or cluster) as a means of
abstract definition. This construct hides information: the details of the definition of the abstraction are isolated from the remainder of the program, which may access the data only through a number of narrow interfaces to the capsule. The procedures and functions of the interface are the operations that characterize the abstract data type defined by the capsule.

In more or less elaborate form the capsule has been made a major part of the design of many recent languages [CLU; Alphard; Model; Modula; Ada] and itself extended and modified [Mitchell and Wegbreit 1977]. Abstractive power comes from allowing newly defined modules to be used equally with primitive types in the construction of further modules. But the compromise remains: efficiency has been served by maintaining the programmer's awareness of the real machine's limitations in the data space the language supports.

Concerns with reliability of programs are represented in the formulation of various programming methodologies, providing a range of informal to semi-formal ways of disciplining the structure of a program to display the programmer's intentions most clearly. They have also increased interest in the possibility of providing formal proofs of a program's correctness. Both of these have influenced language design: the methodologies of information hiding [Parnas 1972] and stepwise refinement [Dijkstra 1972; Wirth 1971] lead directly to the capsule concept, and supporting the requirements of provable programs is a major aim in Alphard and recent work by Dijkstra [1976]. Other methodologies such as program transformation are used informally without language support to develop efficient representations of given
programs in parallel with proofs that the resulting program is correct [Dijkstra 1972; Pritchard 1979].

Work on data definition facilities is concerned with providing reasonably efficient and naturally expressive forms for constructing abstract data definitions [Ledgard and Taylor 1977]. One approach is to define more abstract primitive types and constructors, and to apply automatic or semi-automatic methods to derive an efficient representation [SETL, Low 1974], or to optimize programs automatically, using conventional data structures [Kibler et al. 1977]. Another is to separate the concerns of abstraction and representation into distinct elements of the language, gaining differing degrees of what has been termed "data independence" or "representation independence" [Mealy 1967], so that the choice of representation is not fully determined by the statement of the abstraction. Thus different work has emphasized abstract definition by algebraic means, making program proofs easier, and implementation in procedural terms [Alphard]; raising the level of the data space that the procedural abstract language manipulates further from the machine data space [SAIL]; separating the definition and implementation of data types into distinct textual modules, or into separately compiled modules [Modula 2; Ada]; automatically performing transformations of an abstract program definition to an optimized program representation [Kibler et al. 1977]; automatically selecting efficient representations of abstract data types used in a program [Low 1974].
1.2 The problem addressed by this thesis

This thesis addresses the attainment of data representation independence in the context of programming language facilities and programming methodology. It is concerned with the questions of what facilities are appropriate to the definition of data abstractions without considering their representation, and how constructive methods of developing efficient representations from a given abstraction can be provided to complement them.

Separating the definition of data structure abstractions from their representations is a problem that can only be approached in a bounded context. The problem is here bounded by relating it to programming languages and methodologies, so that both "abstraction" and "representation" are terms with referents in conventional programming languages. Thus a "representation" is a description of entities in the data space in which "efficient representations" must lie, that of the conventional real computing machine. Specific designs of machine data spaces need not be taken in detail: the data space of a "conventional von Neumann machine" can be described by a programming language that is sufficiently "machine-oriented" to have uniform linear-addressable storage as its data space. The abstractions of interest are exemplified by the data space of a language for defining and manipulating data abstractions which is independent of the machine at least to the extent of not being constrained to be close to its natural forms and constructions. Suitable forms of data description may be algebraic or procedural, but to be comparable with conventional languages we require the language to have a model of typed data, and
contain the notions of separation of program from data, data type definitions, variables, assignment, and procedures.

The gap between abstraction and representation must be bridgeable, for the independence is not to be taken to mean absolute independence. In terms of practical programming, there must be a constructive method of deriving chosen representations from an initial abstraction.

The problem of data independence, then, is to create an appropriate descriptive language for defining abstract data types, and to systematically and constructively relate the data types in particular programs of this language to equivalents in a machine-oriented language. Diagrammatically, a programming system supporting data independence can be pictured as follows.

```
abstract program
(machine-free language)
representation
specifications
acceptor/translator
concrete
program
(machine language)
```

The relationship between an abstract data type definition and the realization or representation of that type is one aspect of the general problem of implementing a program. The only requirement of the implementation of a program for some virtual machine is that the computations evoked by the two programs (the abstract and its implementation) be the same: any such equivalent program for the real machine is then an implementation of the abstract. Lacking as we do a General Theory of Programs, which would (if it existed) allow the detection or construction of equivalent programs for any machines, there is no more
general solution available which would subsume the constructive bridging problem for data types. Some constraint on the classes of programs that are considered is necessary; following Dijkstra [1972] the programs that may be compared between different virtual machines are those that have the same structure. Each component of the abstract program has an easily identifiable component corresponding to it in the concrete program, and in particular data type definitions correspond.

The form of an acceptable solution to the problem is influenced by some additional concerns. One is that the process of deriving representations from abstractions should produce reliably correct representations: the programmer should not have to be concerned with the mechanics of the construction of machine-level programs which can reasonably be systematized and possibly automated in an acceptor/translator.

The constraint accepted on equivalence should not be so strong as to inhibit the possibility of more than one representation being allowed for a given abstraction in a particular program. For data independence the form of the abstract definition should not also specify the representation, but further, an abstraction should be allowed to be broad enough so that representing each of its instances efficiently may require more than one form of representation. That is, the notion of "type" should not have to be preserved in the same form across the representation gap. Likewise, "identity" and "value" may be variable, so that the possibility of representing sharing by copying exists.
The diagrammatic description above does not prescribe how closely the means of specifying the representation should be integrated with the abstract program text, or whether it should be at all. One side-benefit of data representation independence is that there is a possibility of delaying the binding of particular representations to abstractions beyond the time of writing the program, or of compiling it. Delaying beyond compilation time may mean binding either at link time, as with separate compilation, or at execution time, as with interpreted representations. Delaying the binding to beyond the time of writing the program may be achieved by feeding representation specifications separately to the acceptor/translator.

1.3 The form of the adopted solution

The approach taken to this problem is to work within a single programming language with a "broad spectrum", that is one with a set of common concepts that may be applied in programs which utilize either the machine-oriented data spaces needed to study the development of representations or the abstract data space or a mixture of both.

The common features at both ends of the spectrum are the means of defining and applying data types, all the control mechanisms of the language, and a notion of binding. The machine-oriented data space has relatively fixed binding, reference variables, arrays of fixed length; the abstract space has more dynamic binding, recursive types, and flexible structures.
The language acceptor/translator is driven by the programmer’s specifications concerning a program’s representation, as active instructions specifying the translations to be performed. Each representation specifier is implemented as a transformation of the program, a complete ordered series of transformations producing a cascade of successively modified programs to bridge the gap between an abstract program and one of its implementations at opposite ends of the language spectrum. The structure of the programming system can therefore be pictured in more detail as:

abstract program

representation step 1
(transformation)

translator/acceptor
(transformer)

intermediate program

representation step 2

transformer

concrete program

The Broad Spectrum Language was developed from its abstract end (Koala) first, while its machine-oriented end (Maclan) developed more slowly in parallel with the transformations. Thus it was not until
the research work was well advanced that it could be seen how close Maclan's data space is to Pascal's. Nevertheless it was decided to proceed with Maclan rather than Pascal because it has essential features not available in Pascal, in particular data type encapsulation, generic parameter types for procedures, assignment and equality as non-primitive operations, and arrays whose size is fixed only on their creation. In addition it was apparent that a single spectrum could not include both Pascal and Koala without the latter losing some of its important properties. To assist the reader there is a comparison between Maclan and Pascal in section 3.3.5.

1.4 Thesis organization

Chapter 2 surveys some of the literature on data abstraction facilities. Examples are presented to show that constructing representations by the commonly assumed method of stepwise refinement is inadequate. Chapters 3 and 4 present the various parts of the programming system outlined above. Chapter 3 describes a language "BSL" with a broad data space spectrum. One end of the spectrum is a language for defining data abstractions that we term a "referenceless" language, with the name "Koala" (section 3.2). The other end, with a machine-oriented data space, is described in section 3.3 as a language called "Maclan".

Chapter 4 presents a library of transformations for deriving Maclan programs from Koala programs, section 4.5 providing a description of a language "Xlan" for defining these transformations in a procedural form suited to driving a translator. Chapter 5 discusses some
aspects of the close relationship between the programming language and
the transformation system. Chapter 6 develops an example from
abstraction through to two representations in detail, while Chapter 7
provides a summary of the work and some conclusions and more wide-
ranging commentary on the transformational solution to the problem of
independence of data representation.
2 LITERATURE SURVEY AND CRITIQUE

2.1 Introduction

2.2 Literature survey

2.2.1 High-level languages and representation

2.2.2 Introductions to data structuring

2.2.3 Compilation and optimization

2.2.4 Abstract data types and structured programming

2.2.4.1 Structured programming

2.2.4.2 Hoare and derivative languages

2.2.4.3 Earley and VERS

2.2.4.4 ELI

2.2.4.5 Abstraction vs references

2.2.4.6 Data abstraction languages

2.3 Critique

2.3.1 Representation and the capsule languages

2.3.1.1 Alternative representations of a stack

2.3.1.2 A less refined example - the set

2.4 Conclusions: transform a Broad Spectrum Language

2.4.1 Representation and abstract types

2.4.2 Stepwise refinement and transformation

The problem of data structure representation in programming languages rests on how we describe concrete representations of data structures, how we describe a particular data structure in the abstract, and how we relate the two in programs. This relationship must bridge the "representation gap" between abstract and concrete, without filling that gap so firmly as to destroy the "data abstraction" between [Nealy 1967] of the two.

Our knowledge of data structure representation resides largely in a stock of practical techniques, which are generally applied to programs. Normally, and in a few computer systems, in this chapter we shall consider the forms by which this knowledge is expressed, in both

null
2.1 Introduction

The study of data structure representation is part of the problem of implementing high-level language programs. The general solution to program implementation involves considering all low-level programs that have equivalent effect to the high-level program. In this work, to make this solution space manageable we factor programs into data and control components, and (following Dijkstra [1972, p. 23]) require that the control components of high-level and low-level programs "establish the same net effect" and have the same sequencing at many levels of comparison. We extend this "same sequencing" requirement to data components, and require a clear correspondence between specific abstract and concrete data items in the programs, again at many levels of comparison. We use the term "closely equivalent" to describe programs meeting these requirements.

The problem of data structure representation in programming languages rests on how we describe concrete representations of data structures, how we describe a particular data structure in the abstract, and how we relate the two in programs. This relationship must bridge the "representation gap" between abstract and concrete, without filling that gap so firmly as to destroy the "data independence" [Mealy 1967] of the two.

Our knowledge of data structure representation resides largely in a set of practical techniques, which are generally applied to programs manually, and in a few automatic systems. In this chapter we shall consider the forms by which this knowledge is expressed, in both non-
language and language-related areas. We later make use of a few of the techniques themselves as examples in the system for deriving representations by transformation. We are therefore looking for descriptive techniques which are abstract and can be directly applied to programs. As for a programming language description of concrete data structures, we need look no further than the records, arrays and pointers used widely in languages such as PL/I and Pascal: we shall not survey these.

The programming language description of a data structure abstraction is the abstract data type. We are concerned only with algorithmic descriptions of abstract types and their characteristic operations, because algorithmic descriptions remain the most used and intuitively understood method we have for specifying complex programs for all purposes. There were several language designs concerned with data abstraction in the late 1970s, in particular the "capsule languages" CLU, Alphard and Model. They share a similar approach to data, very similar levels of algorithmic abstraction and similar relations between abstraction and representation. These notions are found in aspects of the "data type"; in particular the languages’ data independence rests on the properties of their types, expressible as answers to the questions: "How readily can we replace one representation of an abstraction by another?", and "How readily can distinct representations be used for different instances of one abstraction within a particular program?"
2.2 Literature survey

2.2.1 High-level languages and representation

The problem of the representation gap between high-level data structures and the machine is addressed largely informally in the literature. The areas called "data structuring" and "run time structures" present many specific solutions to parts of the problem, and these solutions have some generality. Others are embodied in the representation decisions frozen into compilers and interpreters for particular languages and machines.

The gap between the high-level language and the machine avoids attention because, in general, a deliberate language design decision has been made to minimize it. In Algol 60, for example, the primitive types are modelled by well-defined machine primitives, the array by a descriptor and block of contiguous storage. The addition of the pointer or reference in Simula 67, Algol 68 or Pascal accentuates this: the semantics are designed to make a stored machine address the simple implementation. That is, the design of language data features is conditioned by the class of "conventional" machines; as Low says, "the data structures ... some programming languages [e.g. PL/I, Algol 68] allow are very detailed and maintain a close tie with their implementation" [Low 1974, p. 1]. Wirth [1974] even makes this an important criterion in the design of a good programming language feature, "that it does not imply any unexpected, hidden inefficiencies of implementation" [Wirth 1974, p. 390]. A good language "must ... give [the programmer] an idea of the complexity and effectiveness of the
features it offers ... if the use of the same feature under only slightly different circumstances yields widely different factors of economy, then the language clearly lacks this desirable property" [ibid]. While this is perhaps an extreme view, it is a clear statement of one of the roles of high-level programming language constructs. As well as encapsulating higher level abstract concepts they stand directly for groups of machine-level constructions. We strongly disagree with this view: structures suited to building machines are not necessarily closely related to structures for solving problems. This equivalence of high- and low-level data is too close to permit any degree of data independence.

2.2.2 Introductions to data structuring

The kind of representational knowledge we have about data structures is contained in many introductory text books on "Data Structuring". In these the abstract data structures are generally described by their means of implementation: Knuth [1973], which is a standard reference, is a good example. The representation of an abstract structure is described by means of snapshot diagrams of a particular instance on a conventional linear-coordinate word-byte machine. Such diagrams use blocks to represent contiguous pieces of storage (equivalent to records and arrays) and arrows for the values of stored addresses (reference variables) connecting the blocks.

For example, consider linked allocation for sparse matrices (based on diagram 10 p. 299, and fig. 14 p. 300). "The representation ... consists of circularly linked lists for each row and column. Each
node of the matrix contains three words and five fields ... There are special list head nodes, BASEROW[i] and BASECOL[j], for every row and column." A word in the machine Knuth uses has a sign (shown as the leftmost field) and five bytes, each capable of having 64 values: the representation of a node by a block of three contiguous words is shown below.

In the diagram of the whole structure nodes take this form:

```
+--+-+--+-+--+-+--+-+
| ROW | UP |
+-----+-----+
| COL | LEFT |
+-----+-----+
| VAL |
+-----+
```

The representation of the matrix

```
( 50 10 )
(  0  0 )
(  0 -20 )
```

is shown as follows, where a row value of -1 indicates a column list header and vice versa; fields left blank are irrelevant.
The remainder of the description is a program text defining the operations applicable to the structure by showing how they are implemented. Other texts use programs in more declarative languages to define the storage blocks more carefully and make the code more readable, but the principles remain the same; the use of Pascal, Algol 68, or PL/I data facilities provides a description of common machine features, without inventing a new machine or describing a specific one. "The
use of pictorial representation is a sign of the immaturity of the computing profession" [Dijkstra 1977] and these descriptions of data structuring remain very primitive. The reader is expected to generalize from the examples given: no principles of representation (other than, perhaps, "pointers are the basic means of representing data structures") are drawn out other than informally, and the relation between alternative representations for the one abstraction is only implicit.

A great deal of representational knowledge is incorporated in such work on data structuring, and in similar studies of representations in non-uniform storage media. The question this prompts is what principles of representation can be extracted from these descriptions, and how can these principles be expressed?

2.2.3 Compilation and optimization

Optimizing compilers embody knowledge of representation. A compiler for a language with strongly machine-conditioned data structures (e.g. the widely-distributed Pascal P4 compiler) need incorporate only a limited understanding of data representation; the language features are chosen to have a direct machine-code equivalent, and the compiler implements this mapping directly.

An optimizing compiler for such languages, however, or a practical compiler for a very high level language (i.e. one such as SETL in which language primitives are not necessarily those well-suited to real machines) must incorporate more knowledge of alternative
representations for the one high-level construct. Recent work decomposing the code generation process in a compiler looks at selecting representations for abstract data structures guided by their time and space costs [Low 1974], and the interaction between control structure and array data structures in systematizations of general code "optimization" [Allen and Cocke 1972; Standish et al. 1976; Loveman 1977].

Low [1974, 1978; Low and Rovner 1976] describes an automatic system for choosing efficient representations of abstract data structures. It is provided with a library of machine-level, partial implementations for the abstract set and associative triplet structures of SAIL. For sets the representations provided in the library include linked list, AVL tree, bit-array, hash table, and sorted variable-length array. The system first partitions the set of variables of a particular program by the existence of any assignment and value comparison operations. A group of feasible representations is then chosen from the library for each subset; a partial representation is feasible if it implements all of the operations applied to members of the subset.

The library includes formulae for the time and space cost of each representation, and a hill-climbing heuristic procedure is used to achieve an optimum choice of representations for all the data structures in a program. The formulae are applied to data on frequency of use of operations, expected values and size of data instances, etc. obtained from the programmer and from trial executions of the program.
Low's work demonstrates an automated ability to analyze a program with abstract data structures and no type declarations, and to make an intelligent choice of representation for the program based on that analysis.* The representations are however given to the system in their machine-acceptable form: Low has improved the ability of a language acceptor to select "sensible" representations, but has not advanced our understanding of those representations.

Low's method chooses a representation from a number of alternatives: optimization refines a choice once made. Allen and Cocke [1972] provide "a categorization of most of the transformations which [were] currently reasonably well understood" for machine independent code optimization. The transformations are of program code quite close to the machine level and are presented informally. The aim in these optimizations is to save execution time; they concentrate on the manipulation of loops to reduce the amount of control variable handling, the elimination of dead code, single evaluation of common subexpressions, etc.

The later work of Loveman [1977] and of Standish's group [Standish et al. 1976; Kibler et al. 1977] includes most of these optimizations in a stronger framework of source-to-source program transformation. Loveman adds more transformations, again mainly concerned with loops, which further modify uses of the whole or parts of arrays in particular. These optimizations concentrate on control structure (looping, conditionals), the movement of expression evaluations (out

* Schwartz [1975] discusses a similar analysis without the sophisticated selection methods.
of loops, into compile time), and the interaction of the index­
controlling loop control structure with the index-accessed array data
structure. A transformation is specified as an action driven by a
pattern match on a parsed program and a supplementary predicate, with
a further "win-predicate" to determine whether there is some gain in
applying the transformation in a particular case. The action consists
of substituting for patterns, making assertions, and suggesting the
names of other transformations to try. For example, a transformation
to substitute values for manifest simple expressions takes the form:

```
custom_computation:
  IN "x op y" WHERE constant(x) AND constant(y)
  THEN "x op y" => value(x) op value(y)
  TRY custom_computation, constant_assert.
```

In applying transformations to related loop control variables and
array index expressions, Loveman considers the representation of a
two-dimensional array as a one-dimensional array, "assuming row major
storage". However this representational knowledge is not regarded in
the same way as the "optimizing" transformations. Other representa-
tions that might be expected here (e.g. a choice between flattening
two-dimensional arrays by row or by column, or representing them as an
array of pointers to rows or columns) are not mentioned.

Standish et al. [1976] also formalizes transformations controlled
by pattern-matching combined with (possibly procedural) "enabling con-
ditions". The transformations are applied to a program text in
interaction with the programmer. In addition to Loveman's transforma-
tions the representation of abstract data structures (sets) is con-
sidered briefly as well as procedure expansion and generation. A
typical transformation to unfold a conditional expression is

\[
\text{PROVIDED is commutative (X, Y):}
\]
\[
X \langle \text{operator} \rangle (\text{IF Y THEN Z ELSE W})
\]
\[
\Rightarrow (\text{IF Y THEN X \langle \text{operator} \rangle Z ELSE X \langle \text{operator} \rangle W})
\]

The enabling condition is the expression between "PROVIDED" and ":". The double-tailed arrow separates the two halves of the transformation and specifies its direction. The "\langle \text{operator} \rangle" is a syntactic category; W, X, Y, and Z are "program fragments"; each is substituted for uniformly throughout the transformation.

The representation of a set as a list appears as an assertion ("Represent (Y) by a (List)") and a number of transformations whose "enabling conditions" are related: these conditions depend on the representation assertion. For example:

\[
\text{IF (the Representation of (Y) is a (List))}
\]
\[
\text{THEN Assert (the Representation of (Y) is (Finitely Enumerable)).}
\]

The transformations enabled by these conditions map each of the (abstract) elementary operations of a set into pieces of code which use operations on a list. This group of transformations includes one which depends on the representation being "finitely enumerable", for instance, and others which depend on the representation being a list, others on it being an array. Operations to determine membership of a set, and to iterate through all members, have the following transformations:
These transformations thus capture some elements of data structure representation, and do so in a more factored form than Low's library. The detail given is insufficient to indicate the extent of this work; however the representation of data structures at a lower level of abstraction (that of two-dimensional arrays, for instance) is apparently not pursued. Kibler et al. [1977] for example considers in detail transformations on a program which manipulates a two-dimensional array, but no transformation changes its dimensionality.

The concepts and notations employed here are obviously relevant to describing the relation between abstract data and its representation, but the ranges of abstractions and representations are small.
2.2.4 Abstract data types and structured programming

2.2.4.1 Structured programming

"Structured programming" (in the sense of describing programs as a hierarchy of successively refined abstractions) is currently an important paradigm for our understanding of programs. It was initially formalized for the control structure of programs, using the existing control/abstraction feature of the "closed subroutine" or procedure [Dijkstra 1972; Wirth 1971a]. A program is a definition of a mechanism which evokes a computation: we can "use our understanding of a program to make assertions about the ensuing computations", and "the ease and reliability with which we can [do this] depends critically upon the simplicity of the relation between the two"; in particular we must "shorten the conceptual gap between the static program text (spread out in 'text space') and the corresponding computations (evolving in time)" [Dijkstra 1972, p. 16]. Dijkstra gives little attention to data, however, and Wirth refines the abstract data structure in a single step.

The "data abstraction" languages deliberately extend the principles of structured programming to the definition of data structures, converging on the FORM (Alphard), CLUSTER (CLU), and SPACE (Model). Following Horning [1976] we use the term "capsule" for this class of data definitions (and hence refer to "capsule languages") throughout this work. The capsule has the primary concern of giving a functional description of an abstract data structure, disciplined by the principles of stepwise refinement and information hiding [Parnas 1972].
same stepwise refinement links the abstraction to its representation in the form of primitive data types. We now describe these and related data structure definitions which are less integrated with structured programming; in the next major section we will consider how well the structured programming paradigm copes with our questions of abstraction and representation [section 2.3].

2.2.4.2 Hoare and derivative languages

Hoare [1968; 1972] presents a theory of data structuring that is strongly dependent on a concept of type similar to that of mathematics, and is incorporated in the design of Algol W, Algol 68 and Pascal. A type is a set of values, and a variable or expression has a type depending on the values it can have. Data structures are abstractions described through types. Simple data types are (optionally ordered) enumerated sets of symbols, or arithmetic, or a subrange of these.

Non-elementary data types are constructed from elementary or structured types by a small number of methods (cartesian product, discriminated union, array, powerset and sequence); their properties depend only on their method of construction. The description of each data type constructor is by its abstract form (in a declarative notation combining Algol 60-like keywords and set concepts), the manipulations that apply to instances of the constructed type (using Algol 60-like statements and expressions), and the axioms of the interaction of these manipulations and values (in formal logic). These few constructors cover many examples of data structuring. The axioms
summarize the theory, specifying invariant properties of constructed data types, and the effects of manipulations on them.

Hoare discusses the representation of each data type on a conventional words-bytes-bits machine using a description generally in English words with block and arrow pictures of parts of computer storage. Two embellishments of the constructive type definitions affect the representation of types including them: the concepts of recursion and sparseness. Recursive data structures need no distinguishing mark in their declaration, but require a pointer in their representation. A sparse array or powerset type is marked with "SPARSE" in its declaration. This makes no difference to its abstract properties, but it is expected that few of its range of possible values will be used, and therefore a SPARSE ARRAY is more efficiently "represented as if it had been declared" as a SEQUENCE of key and information cartesian products, for instance.*

In Hoare's work and in related languages a data type is a set of values with a behaviour completely determined by the construction of the type. There is no power to define an abstract data type characterized by a set of user-defined operations.

The incorporation of the Hoare abstract type constructors in the second Algol generation of languages is affected by machine efficiency considerations, and the distinctions between abstraction and represen-

* Pascal similarly applies the representation modifier "PACKED" to arrays, records, sets, and file types, but provides only an intuitive (and implementation-dependent) description of its effect [Jensen and Wirth 1975].
tation are lost. The languages include a reference type constructor to distinguish dynamic and recursive structures from others, by their declarations and operations; and the relatively inefficient "sequence" is omitted or implemented as a sequential access external file and therefore operationally restricted.

2.2.4.3 Earley and VERS

Earley [1971] points out the distinction between the semantics of data structures and their implementation in describing the VERS language data structuring facilities. One abstract type is provided: the V-graph (a directed graph with uniquely named or numbered edges from any node). Its sole method of articulation is the "access path", a sequence of edge labels. Specific data structures are described as instances of V-graphs and operations on them. Classes of V-graphs are described transformationally, not declaratively; there is no other means of data abstraction. The implementation facility consists of adding a specific machine word-bit layout to a V-graph using a block of contiguous storage for each node, arcs being references, with the ability to implement the primitive V-graph operations for a specific data item as composite operations on another set of V-graphs. The representation provides a direct mapping from V-graphs onto machine data structures; "lower level V-graphs ... more closely (or exactly) reflect the structure of the data in the machine" and "at the lowest level each node, link, and atom corresponds exactly to some allocatable resource in the machine". There is thus a single language which incorporates both an informal data abstraction facility, where the
data structures are not conditioned by the machine, and a limited machine data specification facility, used only for representation and not intended for general programming.

2.2.4.4 ELL

ELL [Wegbreit 1974; Holloway et al. 1974] seeks to provide both a very efficient representation for data and a natural notation for its use. The notation is provided by making ELL an extensible language: in particular some user-defined behaviour of modes (i.e. data types) can be specified by declaration. An "extended mode" definition consists of an instance of another mode chosen to implement it, and a set of routine definitions operating on that implementation. These routines refer to specific aspects of the behaviour of a mode: type conversion, assignment, selection, printing, and generation. Syntactically conventional program constructs which require such properties of their operands (a mixed mode assignment statement, for instance) are evaluated by calling the corresponding routine. The built-in modes and mode constructors include ones which map directly and efficiently to machine structures (modes such as integer, pointer and Boolean, a "vector" mode constructor to give a fixed-length array). Mealy [1974] gives examples of ELL being used to define representations of abstract structures.

An extended mode definition can be used to capture one level of abstraction while constructing an abstract data type by stepwise refinement: the operations and behaviour of the type can be implemented as details that can be ignored by modes at higher levels of
abstraction. The behaviour that can be easily re-defined is restricted to the few language-defined properties, other operations being somewhat clumsier. The gap between high-level constructs and the machine is closed in ELL by data structures which correspond directly to machine structures: the distinction between the refinement of a data abstraction (i.e. the definition of its logical properties) and its representation can be made only implicitly by judicious use of levels of abstraction, and is not a notion inherent to or enforceable by the language.

The representation of these structures is discussed, using Mesa [Geschke and Mitchell 1975] takes some similar notions of type behaviour (assignment, selection) and allows their re-definition: the emphasis here is strongly on having multiple syntactic constructs to evoke such procedures. The aim is to achieve a separation between abstract data types and representation, but again this is not enforceable, and the explicit use of references is required for more notions than in ELL (to pass parameters other than by value, for example).

2.2.4.5 Abstraction vs. references

Hoare [1975a] points out how Algol 68, PL/I and Simula 67 rely on the reference or pointer as both a mixture of (unidentified) logical abstract notions and a direct description of representation. He suggests a mixed bag of reasons for the undesirability of references in high-level languages, some of which are analogous with arguments against the GOTO: the reference is both logically difficult to cope with in programs, and may not be as efficient in a programmer’s hands as its direct mapping to a stored machine address would suggest.
Hoare develops a notation for one class of abstract data structure, a strict tree-like recursive data type. This uses a form of cartesian product and discriminated union to define trees of quite rich structure recursively. The operations on a tree are constrained to preserve its properties: they are unable to introduce cycles, for instance. This is achieved by forbidding selective updating of a tree, constructing values by functional forms and traversing them by recursive procedures, always using value-copying semantics.

The representation of these structures is discussed, using storage block and pointer arrow diagrams, machine addresses and tag bits. The selective updating and sharing of nodes that are not permitted in the abstract are then introduced to optimize the representation.

Hoare sees automatic optimization, however, as divorcing the programmer from the representation of the program with consequent loss of efficiency. He suggests that a notation might be found to encourage the programmer to pick out cases susceptible to optimization, or that the programmer guide the optimization process at some intermediate but low language level. Until such ideas can be implemented the programmer should use the abstractions to design the program reliably and then translate manually to some low-level language with pointers, using "systematic correctness-preserving techniques".
2.2.4.6 Data abstraction languages

The capsule is a data abstraction definition mechanism which intentionally meets the requirements of structured programming. It is provided in very similar forms by the FORM of Alphard [Wulf et al. 1976], the CLUSTER of CLU [Liskov and Zilles 1974; Liskov 1976] and the SPACE of Model [Johnson and Morris 1976]. The three are compared in Morris [1976]. Alphard in addition incorporates assertions defining a form's properties. The capsule formalizes and generalizes the Simula 67, ELL and Mealy [1967] concept of an abstract data type as being a state space and a set of operations over it. The state space (called "REP" in CLU, "DEF" or "REP" in Model, "REPRESENTATION" in Alphard) appears as a local environment containing instances of other types; the operations are functions, procedures and operators in this environment. The internal detail of the capsule, in particular the nature of the state space, is hidden from the environment containing the capsule; only explicitly listed operations are exported from the capsule.

Capsules are also used to describe classes of data types, by what Berry et al. [1976] refer to as "constructor capsules": a constructor capsule is one with parameters, among which types are allowed. An instance of a constructor capsule is used as a data type. A "stack" constructor can be defined, for example, with a parameter stating the type of its elements; the type "stack(integer)" is one of its instances. Primitive type capsules, constructor capsules and constructors such as integer, array, record and union are the elements from which all capsules ultimately derive.
Stepwise refinement is made possible in capsules by the independence of the constituent data types that describe any capsule. Just as the constituent statements of a structured statement can be refined in ignorance of each other and of their containing statement, so can the types applied to construct a particular capsule be themselves refined independently, as capsule definitions. For instance, although a type applied as index type to an array constructor may be constrained to possess certain abstract properties (as expressed well in Model in particular), the refinement of base and index types is unaffected by their application in this constructor, or by the use made of the resulting type.

Abstraction and representation are thus joined; but these languages make no distinction between the logical refinement of an abstract data type and the specification of its representation. They assume that the mechanisms that are appropriate for building a hierarchy of well-behaved abstractions are also suited to specifying their efficient representation, and that a "good" refinement can also be a "good" representation. They also assume that the primitives can be chosen to combine sufficient programmer control of machine representation with behaviour clean enough to avoid violating the principles of encapsulation (e.g. a non-local reference variable could export access to an object otherwise hidden within a capsule). The Alphard workers, for instance, are simply "hopeful" that the protection and encapsulation mechanisms, and restriction of references to a single level, are sufficient to allow references (which are an "undesirable construct") to be used safely [Shaw 1976]. Berry et al. [1976] argue that refer-
ences are inappropriate as an abstraction: for efficiency's sake they should be available in the language, but for the sake of good structuring should be restricted to use within the bounds of single capsules at the lowest level of refinement. Berry proposes that a rich set of "dangerous" but efficient "basic constructors" (like reference, array, record, union, set, subrange, flexible array) be provided to allow the programmer access to "well-known implementation techniques", i.e. to control the machine representation closely. Constructor capsules, including these basic constructors, are to be available only for defining the implementation (i.e. state space) of other capsules: only type capsules may be manipulated directly by the program in general. The manipulation of the dangerous constructor capsules is thus allowed only within a narrow program scope, so localizing their use that the program can be proved despite their presence.

Mitchell and Wegbreit [1977] describe a version of capsules which is distinguished by a generic definition facility for constructor capsules ("schemes"). A scheme is a capsule, but a set of schemes of the same name and similar parameter specifications can provide alternative implementations for the same abstraction. A data type instantiates a scheme, selecting one implementation on the basis of properties of the types and values of actual parameters. A "needs clause" provides this selection mechanism. It may choose a representation directly (using the value of a parameter for instance), or indirectly, by depending on concealed attributes of the actual parameters. The only definition of the abstraction is that of its implementations. A scheme set is the constructor of a single abstract data type, but its various implemen-
tations are not type compatible with each other despite this common abstraction.

2.3 Critique

We have surveyed this work for the light it throws on the relation between the specification of an abstraction and its representation mentioned above. Firstly, the capsule languages provide a clear and usable formalization of data type abstractions. They are able to define formally data types and classes of data types that correspond well with our intuitive notions of data abstraction. Hoare's earlier work isolated certain data abstractions useful in programming, and exemplified the method of characterizing those cases by their behavioural properties that ELL and the capsule languages generalize.

As for making a distinction between an abstraction and its representation, however, the capsule languages are inadequate, as we shall see below.

Although Hoare distinguishes between the behaviour of data types (i.e. the definition of the abstraction) and their implementation, the derived capsule languages do not generalize this distinction. Alphard does capture the semantics of a capsule in logical assertions which are distinct from the state space and algorithms of the implementation: however we are concerned with the algorithmic, and not assertive, treatment of abstract data definition.

In all the capsule languages the notions of abstract data type and of representation are both depicted by the data type as defined by
a capsule; and the representational and logical properties of a cap­
sule are displayed by a stepwise refinement of capsules. That is, the
machine representation of a capsule is the expected representation of
the primitive types and constructors (array, record, ref, etc.) that
define its semantics, and the programmer controls program implementa-
tion by choosing particular refinements of equivalent abstractions.
The philosophy of the capsule languages is to provide alternative
representations by extra-language features: by re-writing the refine-
ment of a capsule, or choosing previously written capsule implementa-
tions from a library. The library capsule may be textually incor-
porated in the program, or added in the link-edit phase of compilation
[Shaw 1976; Liskov and Zilles 1974].

Berry's separation of capsules into "constructor" and "type" cap-
sules does not make a distinction between semantic definition and
implementation: the primitive constructors like array and pointer have
both functions. Ada and Modula 2 make the distinction only in a tex-
tual sense, with no underlying strength. Mitchell and Wegbreit [1977]
distinguish to the extent of making many representations of one
abstraction possible within a typing system, but allow no abstract
definition distinct from this collection of representations.

Earley does make this distinction, with a very limited range of
alternative representations: an abstraction can be refined in V-
graphs which have well-defined semantics, and the implementation of
these V-graphs specified in the storage layout of the record
representing each of their nodes. Substantial differences in
representation (e.g. linked list vs. array) are not captured thus and
require explicit reprogramming of the abstraction's refinement in V-graph terms.

2.3.1 Representation and the capsule languages

In considering the representation of abstract data types in the capsule languages, we add a question of expressive capability to those on data independence:

1. How readily can an abstraction be specified distinctly from a representation of it?

2. Can different representations of one abstraction be mixed in one program?

3. Are there representations that are difficult to derive using the stepwise refinement method?

The first two questions are considered together, taking representations of a stack abstraction as examples. We must constrain the third by accepting the limitations of strong typing in this family of languages, but look at the representation of a class of set below.

The examples use slightly bastard CLU. They are not guaranteed syntactically correct, and treat variables and objects (values) more conventionally than in CLU, but attempt to demonstrate the power of the abstraction mechanism.
2.3.1.1 Alternative representations of a stack

Consider everyone's favourite example, the abstract stack. The skeleton of this is:

\[
\text{stack} = \text{CLUSTER} \quad \text{[base:TYPE]} \quad \text{IS} \quad \text{push, pop, create, empty};
\]

That is, "stack" is an abstract data type constructor which can be applied with a type parameter to produce a type whose operations are push, pop, create and empty. This may be fleshed out by assuming another abstract type "seq" which is an indexable sequence.

PROGRAM 1

\[
\text{stack} = \text{CLUSTER} \quad \text{[base: TYPE]} \quad \text{IS} \quad \text{push, pop, create, empty};
\]

\[
\text{REP} = \text{seq} \quad \text{[base]};
\]

create = OPER () RETURNS CVT;
RETURN REP $ create ();
END create;

push = OPER (x: CVT; n: base);
REP $ grow (x, 1, n);
END push;

pop = OPER (x: CVT) RETURNS base;
RETURN REP $ index (x, 1);
REP $ shrink (x, 1);
END pop;

empty = OPER (x: CVT) RETURNS boolean;
RETURN REP $ empty(x);
END empty;
END stack.

This refinement of the stack cluster asserts that push and pop take and yield values of type "base", and defines these operations by stating that the REPresentation of a stack is a "seq" of the same base type. The symbol CVT signifies a notional type conversion between the abstract type and its representation. A formal parameter (or result) of an exported operation accepts (or yields) an object of the cluster's type, as if its specification had the cluster name as its
type. But within the operation definition the same object is seen as
being of the REP-type. Thus the formal parameter "x" of "push",
specified to be "CVT", is of type "seq[base]", whereas a corresponding
actual parameter must be of some "stack" type.

The operations of a cluster are called by naming the cluster
(e.g. REP (in some other cluster) or stack[INT]) followed by "$" and
the operation name and actual parameter list (e.g.
"REP $ grow(x, l, n)" or "stack[INT] $ push(z, 22)"").

The cluster "seq" has the following properties:

seq = CLUSTER [base: TYPE] IS create, grow, shrink, empty;

We shall consider two implementations of this cluster, called
"by_list" and "by_container". The former represents the sequence by a
linked list, as follows:

PROGRAM 2
by_list = CLUSTER [base: TYPE] IS create, grow, shrink, empty;
listel = RECORD [el: base; next: REF listel];
REP = REF listel;
create = OPER () RETURNS CVT;
RETURN REP $ nil;
END create;

find = OPER (list: REP; ind: INT) RETURNS REP;
c: INT := 1; currel: REP := REP $ nil;
WHILE c < ind AND currel =/ REP $ nil DO
  c := c + 1;
  currel := currel.next;
END do;
RETURN currel;
END find;

index = OPER (s: CVT; ind: INT) RETURNS base;
try: REP := find (s, ind);
IF try = REP $ nil THEN error
ELSE RETURN REP $ deref (try) . el;
END if;
END index;
grow = OPER (s: CVT; ind: INT; n: base);
   COMMENT insert n as the ind-th member of s;
END grow;

shrink = ...
empty = ...
END by_list.

(Note: the reference type constructor and operations here are not
standard CLU, but apply CLU syntax to Pascal semantics. Variable
declarations take the form

variable-identifier "::" type-expression [":="expression]

the initial value assignment being optional.)

The alternative representation we wish to consider is close to
the conventional stack-within-array. In this case it grows from the
top of this array downwards to make the stack as defined above work
efficiently. Array indexing uses conventional syntax; we use
FIXED_ARRAY rather than CLU's flexible ARRAY constructor, and provide
a single bound-pair as its two parameters. The array declarations
here are equivalent to the Pascal "ARRAY [1 .. maxsize] OF base".

PROGRAM 3
by_container = CLUSTER [base: TYPE; maxsize: INT]
   IS create, grow, shrink, empty;
REP = RECORD [bot: INT;
   container: FIXED_ARRAY (1, maxsize) OF base];

create = OPER () RETURNS CVT;
   RETURN REP $ {maxsize,
      FIXED_ARRAY(1,maxsize)OF base{}};
END create;

index = OPER (s: CVT; ind: INT) RETURNS base;
   IF ind <= 0 OR ind > (maxsize - s.bot) THEN error
   ELSE RETURN s.container [s.bot + ind] END
END index;
grow = OPER (s: CVT; ind: INT; n: base);
COMMENT insert n by copying lower elements
one place down;
END grow;

shrink = ...
empty = ...
END by_container.

These two representation abstractions are both implementational
definitions of the abstraction "seq". The two representations behave
differently, and the programmer may therefore wish to choose between
them for each instance of the "stack" cluster. We consider two vari-
ables of the same stack type:

a: stack[INT]; b: stack[INT].

The declaration of "stack" given so far is incomplete, because no
representation of "seq" has been given. If we are permitted to rename
a type by declaration, we can declare

seq [base: type] = by_list [base];

to choose the linked-list representation; substitution of the other
representation is clearly very simple.*

But this forces us to use the representation "by_list" for all
instances of any seq in the scope of this declaration: in particular
both "a" and "b" are represented alike. This is insufficient for
efficient implementation of the program as a whole: the mix of opera-
tions used on "a" may be quite different from those on "b", requiring
different representations.

* The equivalent selection is normally made at link-time in
practice.
Modifying the definition of "stack" allows the representation to be provided as a parameter.

```
stack = CLUSTER [base: TYPE; seq: TYPE] IS push,pop,create,empty
WHERE seq HAS create,grow,shrink,empty;
REP = seq; ...END;
```

We can now write

```
a: stack [INT, by_list [INT]];  
b: stack [INT, by_container [INT, 60]].
```

It is important to note that although both representations implement the one abstraction, they form distinct types: "a" and "b" are not assignment compatible, for instance.*

Choosing a representation by parameterization is adequate for this case where only one choice need be made. But in general many representation choices determine the concrete structure representing a particular abstract data structure. For instance, the elements of the stack must also be represented: if they may be shared with other variables of the base type, there is a choice of representing this by substituting references for the elements (which we may call "pushout"**) or by maintaining multiple copies of the elements. The elements may be a discriminated union type and require a further representation decision: to tag the variable and provide sufficient space for the largest variant, or tag a reference, or refer to tagged records, and so on. The variable declaration might then appear as

* This is apparently also true of the alternative representations grouped in a scheme.
** Described by a transformation of this name in Chapter 4, below.
There is no way to distinguish here between the conceptually distinct notions of the logical type of this variable and its representation. The problem is not merely a matter of syntax: the representation choices must be associated both as a group with a particular declaration and severally with its constituents. Naming particular type and representation combinations would simplify this declaration but would lose much of the point of parameterizing capsules by representation, further obscuring the abstract type of the variable. Schemes simplify the naming somewhat, but the parameters selecting representations continue to be confused with those modifying abstract constructors. Even heavier use of parameterization within the CLU framework, and the trapping of assignments and coercions written for inter-representation transfers, are needed to meet these requirements; they do so only poorly and at excessive programming expense.

2.3.1.2 A less refined example - the set

A further example arises with sets, where the stepwise refinement of capsules cannot describe the representation. We shall consider sets of objects (i.e. variables), not values: two variables with the same value may or may not be members of the one set, and changing the value of a variable must not affect its membership of any sets.

```
set = CLUSTER [base: TYPE] IS create, isin, union ...
```

We shall consider two representations. The first is Hoare's "sparse set" technique: members of the set are maintained as members
of a sequence (representations of which have been mentioned above: in this case the member-objects are shared, which may affect the representations of the sequence).

Note:

- "shared [base]", here used in REP, is intended to emphasize that the elements of the set (and representing sequence) are indeed shared - the representation of this sharing is omitted from this example;

- the iterator construct "FOR m:<type> IN rep$members(s)DO" uses an iterator (undefined here) intended to yield all members of the sequence, assigning them in turn to "m" and executing the statement (this is a standard CLU construct);
"same_object" is an object comparison operator of the "shared" abstraction, not detailed here: its semantics are those of a comparison of two references to the objects yielded by the parameters.

The second representation is commonly used when the number of sets and the universe of possible members are known. Each potential member is associated with a Boolean array variable with one component for each set, the value of which states whether the object is an element of the set. To map the identities of the sets onto indices of the array requires an additional cluster controlling each group of sets with common member universe; this cluster must be added to the program but cannot be contained within the new "set" cluster. In addition the members of the set must be modified to carry the bitmap array. But this cannot be done without altering the cluster defining the type of the members, and such modification of another cluster clearly cannot be done within the stepwise methodology.

There are further examples of representations whose effects are non-localized. One common representation method is the linearization of arrays of arrays. In capsule terms we are left with no entity corresponding to the components of the original structure. This representation is described under the name "incorporate" in Chapter 4, below.
2.4 Conclusions: transform a Broad Spectrum Language

2.4.1 Representation and abstract types

These examples show that stepwise refinement of capsules goes only part way towards data independence, and is inadequate to express certain representations. The central reason is that the logical refinement of an abstract data type and the specification of its representation are inseparable. The desirable situation of a separate, although parallel, development of abstraction and representation is not possible. As it is, those similarities of operations in alternative representation capsules which cannot be captured in a common parametrized capsule are inexpressible. The representation requirements of refinement capsules have come to dominate their abstract properties.

Concrete representation types and abstract types form two classes, which sit uneasily together because they are forced into the one type space. The rules of that space enforce the desirable properties of abstract types: they are hard-edged and mutually incompatible, related only by constituency decomposition. But the relations between an abstract type and its representations should not be of this kind, nor should the relations between concrete types. Berry et al. recognize a distinction between these two type spaces; however their suggested separation is too weak to solve the problem of separate refinement and representation. We conclude that a wider spectrum of data types is needed, in different data spaces and type spaces, with some means of relating them that is more powerful than the stepwise
refinement of structured programming.

2.4.2 Stepwise refinement and transformation

The essence of structured programming is that the progress of the computation should map simply onto the text of the program [Dijkstra 1972, p. 16]. This is achieved by requiring (inter alia) that the transfer of control between program parts textually embedded in a control structure be determined by that structure (e.g. "IF P THEN A ELSE B FI" evaluates A or B, depending on the value of P; no effect of A can cause the evaluation of B also, or of the whole statement repeatedly). The corresponding principle for a "structured" data abstraction is that the interface of a capsule (i.e. those operations that it exports) is very "hard": the strong interactions between these operations (e.g. "pop" and "push") are confined to the "REP" data space within the capsule. The program external to the capsule is unaffected by this interaction in the same way as the conditional construct is unaffected by the internal detail of its constituent constructs. The principle is recursive, in that the behaviour of a capsule with respect to its refining constituent capsules is the same as that of the external program to the capsule.

Such an analysis of the program's structure is similar to the linguists' "immediate constituent analysis" (i.e. the analysis of a compound object in terms of the surface interactions of its compound constituents, each of which can likewise be analysed in isolation). However it does not enable us to explain such notions as the equivalence between "IF P THEN A ELSE A FI", "BEGIN P; A END", and "A"
(given that the evaluation of P has no side-effects). It is equivalences like these that we need to express for data: interactions at deeper than surface level. Chomsky suggested the use of transformational grammars to increase the power of linguistic analysis beyond immediate constituents [Chomsky 1957; Greene 1972], and program transformations have the power required for this problem.

The program transformation systems of Loveman [1977] and Standish et al. [1976] (described in section 2.2.3 above) are transformational in the same sense as transformational grammars. These systems are goal-directed [Wegbreit 1976], towards providing an optimum representation. As production systems they are easily modified by adding or removing transformations and letting the system "decide" which to apply, where and in what order. Much of their predicate manipulation and suggestion facilities are necessary for this goal-directed applicative structure and are not essential to the definition of the transformations themselves.

The data structuring system developed in this thesis therefore uses transformations to generate realizations of abstract programs. It differs strongly from the systems described above in not being a production system, either in the means of invoking particular transformations or in the complexity of and control mechanisms within each transformation. The reasons for this are discussed below [Chapter 4].

Program transformations need a single programming language over which to operate. A standard way of formalizing a data space is to use a mini programming language to encapsulate it [Jones and Muchnik
1978] and distinct languages for distinct data spaces [Ellis 1974]. With source and target programs in different languages, however, step-wise transformations would suffer a discontinuity at the point of changing languages, and logically identical transformations would have distinct expressions for each language. This discontinuity (and to a lesser degree the increased number and domain of the transformations) would obscure the understanding of the problem of data structure representation that language design and transformations together exhibit. A single language, although not isolating a single data space, is therefore necessary.

A conventional language is inadequate because it does not cover a wide enough spectrum of data spaces and type behaviour. A single conventional language suffices for the transformations of Loveman and Standish because they remain within a single data space, and type behaviour is not an issue. For this wider study we need a language with a range encompassing near-machine-level array, record and pointer types for representations, and higher level types than these for the logical refinement of abstract types. There must be different rules for the behaviour of representation and refinement types. No existing language design approaches these requirements;* the next chapter describes the "Broad Spectrum Language" which does.

* The major papers concerning the Munich "Project CIP" Wide Spectrum Language were published in 1979 [Bauer 1979; Partsch and Broy 1979] while this thesis was being written up, and my efforts at that time to obtain the earlier Munich departmental reports were unsuccessful. However I would not in any case have made use of their work: their approach is formal and algebraic, has no one concrete form of language, and does not go beyond single representations for abstract types. My thesis addresses language design and centres on the question of multiple representations, and there appear
to be few points of coincidence with CIP. Nevertheless, I have borrowed the term "spectrum", and the notion of its width from them, and note their support for stepwise transformations in a single wide spectrum language.
3 THE BROAD SPECTRUM LANGUAGE

3.1 Introduction
3.1.1 A Koala example
3.1.2 A Maclan example

3.2 Koala - a referenceless language
3.2.1 Objects, values and bindings
3.2.1.1 Name, object, value
3.2.1.2 Assignment and equality
3.2.1.2 Object generators
3.2.1.2 Kinds of binding; bindable constructs
3.2.1.2.1 Possibility
3.2.1.2.2 Shareability
3.2.2 Expressions
3.2.3 Operations and calls
3.2.4 Capsules, operations and visibility
3.2.4.1 Visibility of identifiers; remote access
3.2.5 Unions
3.2.6 Data structures
3.2.6.1 Sequences
3.2.6.1.1 Indexing
3.2.6.1.2 Traversers
3.2.6.2 Trees
3.2.6.2.1 Recursive operations
3.2.6.2.2 A tree example
3.2.7 Other syntactic constructs
3.2.8 Discussion
3.2.8.1 Dynamic flexibility
3.2.8.2 Strong textual correspondence
3.2.8.3 Sharing
3.2.8.4 Selective update, value assignment
3.2.8.5 Trees and sequences
3.2.8.6 Reference replacements

3.3 Maclan - a machine-oriented language
3.3.1 Objects, values and bindings
3.3.2 Bundles, sequences, references
3.3.2.1 Structs, sequences, operations in Maclan
3.3.2.2 References and sharegroups, visibility
3.3.3 Overlay union
3.3.4 Generic operations
3.3.5 Discussion
3.3.5.1 Machine orientation
3.3.5.2 The BSL Spectrum
3.3.5.3 Multiple representations
3.1 Introduction

The Broad Spectrum Language is complicated, particularly at the abstract end of the spectrum. It is defined in some detail in this chapter so that the reader is aware of properties which are crucial to the transformations described in later chapters. To assist the reader we present a basic introduction followed by some illustrative examples.

The Broad Spectrum Language (BSL) is procedural, strongly typed, capsule-centred, and object-oriented. Its capsules are similar to those of the capsule languages reviewed in Chapter 2; a capsule defines a type and exports its operations, which are functions and procedures. A bundle contains a number of capsules defining alternative forms of the same abstract type, and a number of generic operation definitions which apply to any of the types defined by these capsules.

The supplied data type constructors cover a wide range: struct, sequence, tree, reference, discriminated and overlay union, and enumeration. The supplied types are integer, boolean and character; these, together with user-defined reference and enumeration types, are simple types. Objects of simple type contain a value.

All other types are structured types. String is a supplied sequence type. From objects of structured type can be selected objects which are their components. Traversers range over the components of individual sequence or tree objects. Variables and reference values can share objects within the same sharegroup, which is a
The data model for BSL is as follows:

Environ : Identifiers + SELF -> Names
Bind : Names -> Objects + NONEX
          (many-one)
Objects = Structured_objects + Simple_objects
Structured_objects = Structs + Sequences + Trees
Values = Plain_values + References
References = Object_refs + NIL
Select : (Structs * Identifiers) + (Sequences * Plain_values) -> Names
Follow : Object_refs -> Names
Contain : Simple_objects -> Values
          (many-one)
Indicate : Traversers -> Names + FINISHED
          (many-one)
Tselect : (Trees + Sequences) * Identifiers -> Traversers
          (many-one)

All functions are one-to-one except those shown as many-one. Because objects can be shared Bind is many-one. The evaluation function E produces a name:

E : Expressions -> Names.

The key relationships can be shown diagrammatically:
A data space is a data model constrained by the rules of a programming language. The spectrum of the Broad Spectrum Language is a spectrum of data spaces based on the same model. At one extreme is the data space defined by the constraints of "Koala" (a name loosely derived from "capsules or abstractions language"); at the other is "Maclan" (machine-oriented language). BSL programs from points in between these extremes contain a mixture of the two.

Going from Koala to Maclan the types tend towards the simpler. Koala allows the binding of names to be updated by binding assignment, and allows dynamic variation in the size of sequences and trees. It also permits the containment function to be updated, by total-copy assignment. It has no overlay union or reference types.
Maclan permits no modification in the binding of names, having only total-copy assignment. It permits no change in the size of sequences; it does not allow tree-structured types or traversers. It has no discriminated union, but does have overlay union and reference types.

In the same direction the encapsulation grows more complex. Koala has only capsules; Maclan may enclose them in bundles with accompanying generic operations.

3.1.1 A Koala example

For an example of programs in BSL, consider an abstract definition of data concerning people, organized into electoral rolls. The abstract definition is expressed in Koala capsules called "person" and "elect_roll". Each capsule contains a state-space definition designated "STATE"; the STATE of "elect_roll" is a sequence.

CAPSULE elect_roll = (STATE: SEQUENCE UPTO max constituency OF person SHAREABLE population TRAVERSED over_roll; print = SELF PROC; ...).

This declares the sequence's components to be of type "person". The objects to which the component names are bound (loosely, "the component objects") may be shared, but only with names which are also members of the sharegroup called "population".

A Koala sequence may have fixed, bounded (as here) or unbounded length. Its operations include selecting a component name by its index position (like indexing a conventional array), and updating its
binding or its bound object. Variable-length sequences are modified by removing a component name and its bound object ("shrink"), or inserting a new one ("grow"), at any position.

The sequence has one traverser called "over_roll". It indicates a component name of the sequence, and by it the same operations can be applied as by an index value (access, updating, shrinkage, growth). The programmer establishes its position by RESETTing it to the first component of the sequence and advancing it over successive components (by the statement "NEXT over_roll"), until it becomes finished. An operation to print all members of an elect_roll object uses the traverser thus:

```capsule elect_roll =
  (state: sequence upto max_constituency
    of person shareable population
    traversed over_roll;

  print = self proc;
  begin
    writeln ("start electoral roll")
    reset over_roll;
    while not finished over_roll
      do print() of (# over_roll #);
         next over_roll
    enddo;
    writeln ("end electoral roll");
  end
);
```

A "detraverser" expression like "(# over_roll #)" yields the name indicated by the traverser, which here is of type "person". The parameterless operation call "print() of (# over_roll #)" is therefore made for each "person"-member of the "elect_roll" in turn. This form of call, using "OF" followed by an expression, identifies the "print" operation in the "person" capsule, which prints the components of one
Sharing is established by binding assignment, whose legality is constrained by names' shareability as well as their type. The component objects of an elect_roll's STATE sequence may be shared only with other names of the same type in the "population" sharegroup (and names of that type tagged "ANY" shareability). Thus a person variable can be declared and a new object (subsequently shared as the first component of an elect_roll) assigned to it:

```
VAR mabel: person SHAREABLE population <- NEW person (...);
VAR flintshire: elect_roll ...;
[1] OF flintshire <- mabel;
```

The left arrow "<-" is the binding assignment operator; it updates the binding relation of the left-hand side name. The sharing thus established means that the one person object can be accessed and updated through more than one access path.

Trees are permitted in Koala. The tree is a non-cyclic, non-converging graph structure. A family tree, for instance, may be defined with an operation traversing one path from root to some tip by:

```
CAPSULE roots =
(STATE: TREE OF person SHAREABLE population
  ARTICULATED mother, father
  TRAVERSED tracer);
```
print_maternal_line = PROC (family: roots ANY);
BEGIN
  REROOT tracer OF family;
  WHILE NOT FINISHED tracer OF family
    DO
      print() OF (# tracer OF family #);
      ADVANCE tracer OF family BY mother;
    ENDDO;
  END;
).

The ARTICULATION clause in the declaration of this tree type defines two one-to-one "articulation" functions (of the data model kind) between component names of the tree. The functions are called "mother" and "father". A tree traverser can be reset to the tree object's root component or advanced to indicate the name that is the value of either of the articulation functions for the name it currently indicates.

3.1.2 A Maclan example

In a Maclan program there may be several representations for what is in Koala a single capsule such as "elect_roll". The representations are defined by distinct capsules; together they form a bundle. The properties of copying assignment and comparison between objects are maintained from the Koala abstraction to the Maclan representation: part of the mechanism for this is the explicit definition of these operations in the bundle, with generic parameter types (denoted by "identifier 'from bundle"'). The bundle also contains a copy of the original abstract (Koala) capsule.
BUNDLE elect_roll = [CAPSULE ABSTRACT =
(State: sequence UPTO max_constituency of
  person shareable population
  traversed over_roll;
  print = self proc;
  begin
    reset over_roll;
    ...
  end;)
  (* end of abstract *)
]

CAPSULE linked_roll =
(State: (* first representation *) ...);

CAPSULE arrayed_roll =
(State: (* second representation *) ...);

"" := proc (left: lt from bundle any;
             right: rt from bundle any); ...

"=" := ...];

VAR previous_roll: elect_roll $ arrayed_roll;
VAR next_roll: elect_roll $ linked_roll;

next_roll := previous_roll;

The capsules in a bundle are identified by the bundle identifier modified by the capsule identifier, connected by "$". Total-copying assignment for structured types is not a primitive operation in Maclan, but operators can be declared explicitly as generic operations of a bundle. They are called with the same syntactic form as the primitive total-copy operator for simple types.

Type constructors are restricted versions of those in Koala, with the same declarative forms. The Maclan sequence can be of fixed length only, with no traversers, for instance (and hence is equivalent to a conventional one-dimensional array). The representation "arrayed_list" has such an array as one component of its STATE; the others are two variables which index the array: one indicates the size
of the abstract sequence this state represents, the other substitutes for its traverser:

\[
\text{CAPSULE} \text{ arrayed list} = \quad \text{(STATE: STRUCT \{members: SEQUENCE max\_constituency OF REFTO (person SHAREABLE population); top\_person: 0 .. max\_constituency; over\_roll: 0 .. max\_constituency + 1\strtotime{)\}}) =}
\]

A Maclan reference type \((\text{REFTO} \ldots)\) is qualified by both a type and one of its sharegroups; it is compatible only with types of the same qualification (like a Euclid "collection" or Ada "access type").

We now describe Koala and Maclan in more detail. A full syntax of BSL is given in Appendix 2. We describe in Appendix 1 the variant of extended BNF which is used throughout this thesis.
3.2 Koala - a referenceless language

A language to describe abstract data structures requires higher-level primitives than the array and pointer. We have seen in Chapter 2 that the constructors of data types in the capsule languages are a compromise between abstract and machine levels, and that the reference is a major point of difficulty in defining satisfactory abstractions. An appropriate language will therefore be referenceless, in the sense that the functions of the reference in a conventional language will be divided among a small number of distinct notions. It will conquer the complexity of the uses of references by imposing an ideal model upon them.

The reference is central to dynamic and recursive data structuring in algorithmic languages (Simula 67, Algol 68, Pascal, PL/I etc.), but is very indiscriminate. Its effect on definitions of structured data can be compared to that of the GOTO on control structure [Hoare 1975a], with the additional drawback that the number of reference variables, and their abstract meanings, are subject to alteration during evaluation. GOTOS, which are acknowledged disrupters of program structure, are at least fixed properties of the program text. The GOTO does not by itself signal whether its function is as part of a loop, a closed (or insecurely closed) procedure, a conditional skip, or a mixture of these. The reference variable is also used in the expression of many different abstract notions, and its appearance or manipulation conveys very little information about the intended effects. It may, to take a few instances, be applied to defer binding of variables, to construct selectively and to traverse recursively-
typed structures, to pass parameters other than by value, to bind variables dynamically in traversing structures, to manipulate the existence of bindings (the NIL-valued reference), to distinguish the identity from the value of objects, or to establish dynamic sharing.

The lack of clarity of intent in uses of references inhibits both intellectual manageability and security. Describing abstractions with references makes it unclear whether the result is that intended; the intellectual difficulties are indicated by the problems of providing formal proof techniques for programs in languages which allow references [Berry et al. 1976; Kyne 1978].

In conventional languages the only reference-replacing construction is "by variable" parameter passing (e.g. Pascal's VAR parameter). Simula 67 adds a distinction between an object's identity and value in the case of its TEXT type, providing distinct "by reference" (:-) and "copying" (:=) assignment operators. In Koala the range of reference-replacing constructs includes objects with dynamically changeable structure, traversers for structures, controlled sharing, and manipulable and nonexistent bindings.

3.2.1 Objects, values and bindings

3.2.1.1 Name, object, value

A referenceless language must differ significantly from conventional ones in its treatment of objects and variables. In the BSL data model the environment maps an identifier, such as that declared
by a VAR declaration or formal parameter specification, onto a name. In Maclan the name's binding is fixed and its bound object's containment can be varied; a Maclan variable is an object. In Koala the binding of the name can be varied as well; a Koala variable is a name.

The data model states that evaluating an expression yields a name. Where no name exists (such as in an expression that indexes a sequence beyond its bounds, or applies the indication of a finished traverser) the computation is in error and will abort.

A name bound to "NONEX" is called "unbound", and is said to have a "nonexistent" binding. The use of such a name and its binding is legal in restricted program contexts, described in section 3.2.1.2.1 below.

We shall use a diagrammatic form to represent objects. For instance, a "person" object with a "full_name" component equal to the string "Joe Doe" and "age" equal to 37 can be shown as follows. The type "string" is a sequence of the simple type "char".
An object is shown as a box (labelled with its type) and a number of "select" relations, each labelled with the selecting identifier (field identifier for a struct type) or indexing value (for a sequence type). A traverser is shown by its tselect and indicate functions; e.g. an "elect_roll" of length 3, where the traverser "over_roll" indicates its second component, can be shown as follows:
3.2.1.1 Assignment and equality

assignment = object_expression assignment_operator expression;

assignment_operator = '<-' | ':=' | 'yield' ;

relational_operator = '<->' | ':=' | '=' | '<=' | '>=' | '<<';

There are three kinds of assignment and corresponding equality operators in Koala. In each case the types of left- and right-hand sides must be compatible; further, in the case of binding assignments the types must be identical. Types are identical by name equivalence, i.e. every capsule declaration defines a unique type. Types are compatible if they are identical, or are simple types which are both subranges of identical types, where a type is a subrange of itself.

Binding assignment ('<->'), as the name suggests, changes the binding of the name on the left-hand side to equal that of the right-hand side. That is,

\[ \text{Bind}^\prime (E \ (\text{lhs})) = \text{Bind} (E \ (\text{rhs})). \]

The "same as" relation ('<->') between the two names becomes true. Thus the assignment "y <- x" makes "y <-> x" true, and has this
effect:

If the objects are of simple type then simple value assignment occurs with the same effect as in Pascal:

\[ \text{Assign}(x, y) = \text{Assign}(\text{bind} (x), \text{bind} (y)) \]

Where the objects are structured, the assignment is made such that each has the same structure of components, their corresponding component names have the same type, and each has the same structure of components. Edit total-copying assignment recursively on corresponding pairs of these components.

\[ (\text{select}) \]

The shareability of the right-hand side of a binding assignment must be acceptable to the left-hand side, and if the left-hand side is always bound, the right-hand side name must be bound at the time of assignment. These declared attributes of names are explained in section 3.2.1.2.

Total-copying assignment (" :=") applies to both simple and structured types. It affects the value of objects and makes " = " (totally equal) true between them. By "value" of a structured object we mean its form (defined by its select relation) and its components' values. When evaluated " := " requires the names on each side to be bound to
objects, but no restriction is placed on their shareability.

If the objects are of simple type then simple value assignment occurs with the same effect as in Maclan:

\[
\text{Contain'} (\text{Bind} \ (E \ (\text{lhs}))) = \text{Contain} (\text{Bind} \ (E \ (\text{rhs}))).
\]

Where the objects are structured, the simplest case is that each has the same number of components, their corresponding component names being bound to objects of the same type. Here the effect of total-copying assignment is to perform total-copying assignments recursively on corresponding pairs of these components.

It is possible in Koala, however, for objects of the same type to have different numbers of component names; or for a component name to have no binding to an object; or for objects bound to corresponding selectors to be of different types themselves. The first occurs where objects of a variable-length sequence type, or tree type, differ in length or composition; the second where a struct field name is unbound; the third where the component type is a union. The assignment is legal in these cases and modifies the structure of the left-hand object so that it becomes similar to that on the right. Missing component names are "grown", redundant ones "shrunk"; unbound and union-typed names are rebound. These new bindings of component names on the left are to freshly generated objects which are "total-copies" of the corresponding right-hand component objects. (A total-copy of an object has the same structure and component values as the original without sharing them [section 3.2.1.1.2].)
Top-copying assignment ("=:") also requires an object to be bound to the name on either side of the assignment. Its effect is similar to that of total-copying assignment, modified by the shareability [section 3.2.1.2.2] of the objects' components. For components that are shareable a binding assignment is made from the component name on the right to that on the left. Where components are not shareable, a recursive top-copying assignment is applied to them in the same way as for total-copying assignment, above. Where new component objects must be generated, a top-copy is made.*

Contrast the three assignment operators applied to some shareable elect_roll variables, with the types declared earlier [section 3.1]:

VAR flintshire_79: elect_roll SHAREABLE rolls ...;
VAR flintshire_80: elect_roll SHAREABLE rolls ...;
VAR mabel: person SHAREABLE population <- [2] OF flintshire_79;

Assume the initial situation to be that "flintshire_79" is bound to an object with three components, "flintshire_80" to one with two:

* For the motivation for distinguishing between top-copy and total-copy, see the discussion in section 3.2.8 below.
A binding assignment "flintshire 79 <- flintshire 80" changes the binding of the name associated with "flintshire 79", leaving its previously bound object inaccessible:

From the same initial situation a total-copy assignment "flintshire 79 := flintshire 80" affects the values of all components of the left-hand object, but not their bindings:
The binding of the name corresponding to the variable "mabel" is unchanged; but the value of the person object so bound is updated because it is also a component of the "flintshire_79" object.

Again, from the same initial situation, a top-copy assignment "flintshire_79 :- flintshire_80" leaves the components of the two elect_roll objects shared, by rebinding the component names of the left-hand object and shrinking its later components:

The expressions "TOPCOPY (x)" and "TOTALCOPY (x)" each yield a copy of the same type as the argument and bound in the same way as the argument is. That is, if the argument is unbound then so is the result. If bound to an object, the result is bound to a unique new object that is top- or total-equal to the argument.

A new generator creates a unique object of the same type, binding the generator's actual parameters in the same order as the object's components. Thus
3.2.1.1.2 Object generators

Generating new objects is done by construction with a "new_generator", or by copying an existing object.

copier = ("topcopy" | "totalcopy") (^expression);
new_generator = "new" type identifier actual para_pack;
actual_para_pack = (^expression / ^, ^) * ^,

The expressions "TOPCOPY (x)" and "TOTALCOPY (x)" each yield a name of the same type as the argument and bound in the same way as the argument is. That is, if the argument is unbound then so is the result; if bound to an object, the result is bound to a unique new object that is top- or total-equal to the argument.

A new_generator creates a unique object of the named type, binding the generator's actual parameters in the same order as the object's components. Thus
flintshire_66 <= NEW elect_roll (mabel, 
    NEW person ("John Doe", 16), 
    TOPCOPY (mabel))

creates an "elect_roll" as a sequence of three person objects, the 
first of which is the object bound to "mabel", the second a new "per­
son", the third another new person that is total-equal to that bound 
to "mabel".

As described above, total-copying of an object with cycles in the 
selection and binding of its components would lead to infinite expan­
sion as its descendents (components, and components of components, 
etc.) are copied. This is avoided by the copy's establishing sharing 
of its descendents as soon as possible.

Simple value denotations such as "6", "'p'", "true" are also 
object generators that yield a name bound to a new object containing 
the value. They can therefore be written on the right-hand side of a 
binding, as well as a copying, operation: the actual component "16" 
is bound as the "age" component of a "person" in the generator above, 
for example.

3.2.1.2 Kinds of binding; bindable constructs

Bindable constructs are the declarative programming language con­
structs which introduce new names into the data model's environment. 
The bindable constructs in Koala are: a variable's declaration, a 
function's yield, an operation definition's formal parameter, a struc­
tured type's field, the SELF of an operation. "Type" is the attribute 
usually associated with similar constructs in other languages, such as
variable or parameter declarations. In Koala, in addition to type the
bindable constructs declare two attributes of the corresponding names:
possibility and shareability. Possibility is declared by the presence
or absence of "POSSIBLY" in the declaration. Shareability is declared
by identifying a "sharegroup" to which the name belongs.

The simplified syntax of these constructs is as follows (the com­
plete syntax in Appendix 2 reflects the constraints which the context
puts on the kinds of shareability that can be declared for names):

\[
\text{variable declaration} = \text{`var` formalpara [`<-` expression]};
\]
\[
\text{structdec} = \text{`struct` formalpara_pack};
\]
\[
\text{seqdec} = \text{`sequence`} [(\text{`upto`} integer_expression)
\text{`of` formaltype [`traversed` id_list]};
\]
\[
\text{operation declaration} = \text{identifier } = \text{ [`self` [shareability]] [`traversing`] (\text{`proc` | `func`})
\text{[formal_trav_pack] [formalpara_pack]}
\text{[`:` `possibly`] formaltype `:` `block`};
\]
\[
\text{share_declaration} = \text{`sharegroup` identifier `of` formaltype};
\]
\[
\text{formaltype} = (\text{type_identifier | subrange | union | enumeration})
\text{[shareability]};
\]
\[
\text{shareability} = \text{`shareable` sharegroup_identifier | `any`};
\]
\[
\text{formalpara} = (`possibly`) identifier `:` `formaltype`.
\]

3.2.1.2.1 Possibility

In Koala some of the functions of the conventional reference
variable are provided by the variable binding of an object to a name.
In a language with references the NIL value may be given to any refer-
ence variable, and experience shows that unexpected NIL values are a major cause of difficulty in programs. Declaring a name to be always bound corresponds to disallowing a NIL value for a reference variable; the alternative is for it to be possibly unbound.

By default a bindable construct is always bound. That is, although its name's binding to objects may vary, it always exists in that the name is always bound to some object. Declaring the bindable construct with POSSIBLY, however, makes the name possibly unbound and allows it to have no binding from time to time. Thus the variable

VAR POSSIBLY friend: person SHAREABLE population;

need not always have a "person" object bound to its name, but

VAR mate: person SHAREABLE population ...

must do so.

One consequence is that this second declaration must have an initializing binding, such as

VAR mate: person SHAREABLE population <- NEW person ("Joe", 20).

The more important consequence of a declaration without POSSIBLY is that the programmer need never consider the possibility that this name is unbound (and the implementation need not check for this on every use of the variable).

Names declared with different possibilities remain compatible for binding and copying assignment, but a run-time check is needed to determine whether an assignment, or some other operations (those requiring an object) are in error. For example,

friend <- mate
needs no check, because "mate" is always bound and the previous binding of the left-hand side makes no difference to a binding assignment;

   mate <- friend

requires a check that "person" is actually bound at the time, because "mate" must always be bound. Copying assignments need checks in more cases; both of

   friend := mate;     mate := friend;

require objects to be bound on each side, and therefore the existence of "friend"'s binding must be checked.

"NONEX" (a "nonexistent" binding) denotes an unbound name of any type. Like a binding to an object the nonexistent binding is transferable by binding assignment, and can be tested for by same-as comparison. It yields "true" to the monadic "EXISTS" operator:

   friend <- NONEX;     spouse <- friend;

leaves both "friend" and "spouse" unbound, and the expression

   (NOT (EXISTS friend)) AND (spouse <-> NONEX)

then evaluates as true.

Only some of the bindable constructs in Koala can be declared possibly unbound. The component declaration of tree and sequence, and SELF, cannot. A sequence or tree therefore has an object bound to each of its names. The SELF of an operation declaration denotes a name bound to an object that constitutes the environment in which a call is evaluated, and therefore must be always bound.
3.2.1.2.2 Shareability

The ability to establish sharing of objects without constraint would give Koala's binding many of the disadvantages of the reference variables we are replacing. The shareability attribute of names partitions all names of a given type into disjoint subsets, and the sharing of objects between names in different subsets is forbidden.

A sharegroup is similar to a "collection" in Euclid or an Ada "access type". It is declared to contain names of a particular type:

SHAREGROUP population OF person;
SHAREGROUP minors OF person.

Each bindable construct declares the shareability of its name. For user-defined names the possibilities are to belong to a visible sharegroup of the same type, or to be unshareable:

VAR mate: person SHAREABLE population <- friend;

VAR POSSIBLY child: person SHAREABLE minors;

VAR singleton: person <- NEW person (...);

(the last variable, "singleton", is unshareable).

An object that is bound to a name of some sharegroup may be also bound to (i.e. shared by) any number of names, but only those of the same sharegroup. The corresponding property of Koala program constructs is that names of differing sharegroups cannot be mixed in binding assignments; the binding

child <- mate;

would be illegal, as would any binding from "singleton". Koala is designed so that all its constructions that establish bindings can be
statically checked for their sharegroup properties.*

To write operations for capsules it is frequently convenient to be able to write a single definition to apply to all objects of the type, ignoring differences in their shareability. This is not possi­ble with the classes of shareability described so far, so we need a class which accepts all names. The "print" operation of the "person" capsule provides an example of this, declared as

\[ \text{print} = \text{SELF ANY PROC;} \]

and called legally for any person:

\[ \text{print}() \text{ OF mate;} \]
\[ \text{print}() \text{ OF child.} \]

The "ANY" shareability of the operation definition allows any name bound to an object of type "person", whatever its shareability, to call the operation and thereby be temporarily bound to its SELF.

In the case of "print" the desired effect could be got with no shareability on the SELF, by making a copy of the person on each call, as in "\text{print}() \text{ OF TOPCOPY(mate)}". Other operations may need to update the object of the call, however, and sharing with SELF is essential for these.

The use of ANY names is restricted. To maintain shareability as a statically checkable property of the language, bindings with an ANY name as source are legal only with an ANY name as destination. (Oth­erwise an object bound to a name of one sharegroup, or to an

* This model of persons unfortunately excludes the real­world possibility that a "person" bound as a "minor" may later become bound as a member of the voting "population".

unshareable name could become shared by a name of another group.) Furthermore, the amount of "extra" sharing provided by ANY is strictly limited by allowing it only for the SELF of operations. No explicit binding assignments can be made to these names, only operation calls establishing new bindings. The user cannot escape from shareability controls on binding by using ANY for his own variables.

The attribute of shareability applies to all names and objects in the language. But an object generator such as

\texttt{NEW} person (...) ;

or

\texttt{TOTALCOPY} (mate);

must be capable of being the source of a binding to names of all shareabilities, including none (as shown with the declaration of the unshareable "singleton" above). Object generators therefore have a further category of shareability called "generated" shareability, acceptable to names of all shareabilities.

The acceptability of shareabilities can be shown as follows, taking the shareabilities of type "person" as an example:

\begin{verbatim}
sharegroups:  population minors (singleton) ...

\texttt{ANY}

\texttt{generated}
\end{verbatim}

A name with a particular shareability is acceptable as source of a binding to a name with shareability higher in the diagram. It is also
acceptable to a name of the same sharegroup or category as itself except in the unshareable categories like that shown here as "(singleton)".

3.2.2 Expressions

The evaluation of an expression proceeds conventionally and yields a name. The name may be unbound or bound to an existing or newly generated object. The type, shareability and possibility of the name are derived from those of the expression's constituents.

The arithmetic and logical operators yield a name bound to a newly generated simple object of the conventional result type and value. The name is therefore always bound and has generated shareability. Operators evaluate all their operands, apart from CAND and COR which evaluate partially, left to right.

Expression primaries include literals (numbers; string, enumeration and character literals; NONEX) and object expressions. Object expressions are the syntactic forms whose evaluation yields a name from an environment (e.g. a variable or formal parameter name), selects one from an object (a component name), or generates a name for a new object. Object expressions include generators, function calls, detraverser expressions, dot-selection, applied indexing, object-copying expressions, and the "object of the call" of an operation (which is denoted by SELF).

object_expression = selection | {selection "of"}* (selection | object_head);
object_head = 'self' | new_generator | copier | yield_expr;

selection = applied_index | selector_identifier | function_call | detravexpr;
  
  applied_index = '[` expression `]';
  detravexpr = `(# traverser_expr #)`.

For example

    re OF acomplex
    age OF bear_child() OF [31] OF mother_list
    vote (demo_nat) OF (# over_roll OF flintshire #).

(The last is a function applied to a person yielded by a detraverser expression.) The attributes of these object-heads and selections are:

- selector-, variable-, or parameter-identifier: defined by the
  "formalpara" in the identifier's declaration;

- applied index and detraverser expression (which yield a component
  name from a sequence- or tree-structured object): type and
  shareability as defined for the components in the type definition, always bound;

- function call: as specified in the function header [section 3.2.1.2];

- new generator: generated shareability, named type and always bound;

- copy: generated shareability, same type and possibility as argument;
- SELF: the type defined by the enclosing capsule; the shareability of the SELF marker on the heading of the enclosing operation; always bound [section 3.2.1.2.1].

3.2.3 Operations and calls

```
operation_declaration = identifier `='
    [ `self` [ shareability ] ] [ `traversing` ]
    ( `proc` | `func` )
    [formal_trav_pack] [formalpara_pack]
    [`:` [`possibly`] formaltype] `;` block;
```

(a yield specification (``: `possibly` formaltype") is required and allowed only for a function)

```
formaltype = [shareability]
    (type_identifier | subrange | union | enumeration);
```

```
operation_call = [type_identifier `$`] operation_identifier
    [trav_para_pack] actual_para_pack;
```

```
actual_para_pack = `/(` {expression / `,`}* `)`
```

```
procedure_call = operation_call `[of` object_expression]
```

```
function_call = operation_call;
```

Procedures and functions are referred to collectively as "operations". In operation calls parameters correspond by position. Parameter passing is by binding. The environment maps a formal parameter identifier onto a name, giving it all the capabilities of a variable local to the function; its binding is initially to the actual parameter. The actual and formal parameters therefore must have identical type, be matched for existence and possibility, and have acceptable shareabilities exactly as for normal binding assignment (described earlier).
The yield of a function is a new name. Its attributes are specified in the function header, and it is bound to the result of the dynamically last yield-assignment to this function identifier. A yield assignment is like a binding assignment, with the function identifier on its left-hand side: e.g.

factorial YIELD n * factorial (n - 1).

Self-operations and traverser parameters are described below [sections 3.2.4.1 and 3.2.6.2.1].

3.2.4 Capsules, operations and visibility

A capsule consists of a state declaration and a list of operation and capsule declarations. The visibility and hiding properties of the capsule interface are not under user control: all the operations except traversing operations are visible from outside the capsule (using compound naming); but none of its capsules are, nor are the types that they define. The state-declaration is hidden within the capsule and none of its fields, traversers, indexing or associated implicitly defined operations of structure can be referred to outside the capsule.

3.2.4.1 Visibility of identifiers; remote access

Scoping of identifiers in Koala is block structured, as in Algol 60, extended by the implicit and explicit provision of remote access to capsule exports.
Remote capsule access is the naming of identifiers exported from capsules. We have seen examples of both its forms earlier. **Remote access by type** uses a type-name and "$" separator to form the name of an operation exported from the capsule defining the type, as in CLU. For example

```
person $ is_older (mabel, jim).
```

**Remote access by object** starts with an object expression, rather than a type identifier, and denotes a self-operation in the capsule defining the object's type. The self-operation is evaluated with the yield of that object expression, called the **object of the call**, bound to its SELF, as in

```
print() OF mabel
```

(where "print" and "is_older" are operations of the capsule "person"). This is similar to calling a procedural attribute of a class object in Simula 67 where it would be written "mabel.print".

In a self-operation's body there is unqualified access to the properties of the object's state, and to other self-operations of the capsule. In the body of a normal capsule operation the detail of its capsule's state are visible, but only by syntactic selection: "age OF x" might appear in the body of "is_older", for instance, where "x" is one of its formal parameters. But in a self-operation such as "print" the object-expression "age" means "age OF SELF", equivalent to "age OF mabel" in the evaluation of the call "print() OF mabel".

A self-operation definition thus expresses an operation particularly associated with single objects of the capsule's type: normal operations typically express functions of several objects or less
object-oriented operations of the capsule. It is because a single object is involved, and the operation can apply to all instances of the type whatever their shareability, that ANY shareability is introduced for the SELF of self-operations.

3.2.5 Unions

Union types exist for names but not for objects. The union is discriminated [Hoare 1972a]: i.e. the types included in the union need not be distinct, but are uniquely labelled in the context of that union:

```plaintext
union = 'union' "(" {identifier ":" simple_formaltype / "," } ")";
```

for example,

```plaintext
CAPSULE vehicle = (STATE: UNION (own_car: automobile,
                                bike: bicycle,
                                company_car: automobile SHAREABLE fleet) ...).
```

A name of union type may be bound to objects of any of the constituent types of the union ("automobile" and "bicycle" objects in this example). The binding, not the object, carries the discrimination. It is provided by specifying a discriminant label when binding an object yielded by an expression of constituent type. For example

```plaintext
VAR my_vehicle, your_vehicle: vehicle SHAREABLE family_transport;
VAR redcar: automobile SHAREABLE fleet;

(i) my_vehicle <- bike : NEW bicycle (...);
(ii) your_vehicle <- my_vehicle;
(iii) your_vehicle <- TOTALCOPY (my_vehicle).
```

TOTALCOPY and TOPCOPY of union names include the discriminant of the union-bound object, and its shareability is not the discriminated binding.
The purpose of having union types is to express variations in type structure directly, without sacrificing type security. In Koala objects and union-typed names remain strongly typed, but not at the expense commonly paid (in Algol 68, for instance) of forbidding updating of the name's binding or of the object to which it is bound. This security is enforced by allowing the discrimination statement to be the only means of accessing the object bound to a union-typed name. Like an arithmetic CASE statement the discrimination statement consists of a number of branches, each uniquely labelled by a discriminant label of the union or the symbol "OTHERWISE". The corresponding branch is selected according to the labelling of the name's union-binding at the time the discrimination is evaluated.

```
union discrimination = 'with' object_expression 'discriminate'
{discrim_branch / ','} ['otherwise' statement_list]
'enddiscrim';

discrim_branch = 'when' [identifier':[topcopy|totalcopy']]
discriminant_identifier [shareability]
'then' statement_list 'endwhen'.
```

A discrimination branch may have a form of variable declaration as well as a label; the object which is bound to the union-typed name is also bound to this variable when its branch is selected. This branch variable is implicitly of the type corresponding to this variant of the union. Through it may be had secure access to operations of the object's own type. The branch variable must be declared with shareability compatible with that declared for the union variant. Alternatively, the branch variable may be declared as a copy of the union-bound object, and its shareability is not thus constrained.
For example, assume that an interpreter for a simple programming language is to be written. Two of the data types needed are "constant" (which has a single integer component, accessible by the function "value", and an identifying "tag" - a string) and "variable" (which has a string "ident" and an integer "current_val", to which the operation "revalue" assigns a new value). The type "simple_operand" unites these types and others such as "proc".

CAPSULE simple_operand =

STATE: UNION (const: constant SHAREABLE constgroup,
varble: variable SHAREABLE allvars,
proced: proc ... ) ... ).

One of the operations of this capsule is the function "succ" which returns a "simple_operand" which is a "variable", with value one greater than the value of the operation parameter. If the parameter is a "variable" then that object is updated and returned; for a "constant" a new variable is created and returned. Any other type of actual parameter causes an error. A tracing facility prints out information about the "simple_operands" as they are used.

succ = FUNC (anop: simple_operand SHAREABLE ops):

BEGIN
string $ print ("perform succ");
WITH anop DISCRIMINATE
WHEN aconst: const SHAREABLE constgroup THEN
   anop <- varble: NEW variable (tag OF aconst,
      value OF aconst + 1);
string $ print ("constant tagged");
string $ print (tag OF aconst);
integer $ print (value OF aconst);
ENDWHEN,
WHEN avar: varble SHAREABLE allvars THEN
  string $ print ("update variable");
  revalue (current_val OF avar + 1) OF avar
ENDWHEN,
  OTHERWISE
    error ("succ of unvalued simple_operand attempted")
ENDDISCRIM;
  succ YIELD anop
END.

This example demonstrates two key features of Koala union type discrimination: the object bound to a name of union type can be updated via the discrimination branch variable name (as with "avar"); and updating the union binding itself is secure, even while a discrimination statement is being evaluated and its initially bound object remains accessible via a discrimination branch variable. The latter is demonstrated by the use made of "aconst" after its branch's assignment of a "variable" to "anop".

The discrimination statement is extended to the existence of possibly unbound names with similar security. For example,

VAR spouse: POSSIBLY person SHAREABLE population;

WITH spouse DISCRIMINATE
  WHEN actual: EXISTS SHAREABLE population THEN
    string $ print ("married to "); print() OF actual
  ENDWHEN
  OTHERWISE string $ print ("single")
ENDDISCRIM.

An unbound union-typed name also selects the OTHERWISE branch in a union discrimination statement.
3.2.6 Data structures

Koala has three methods for structuring data: the record-like struct, the homogeneous ordered sequence, and the homogeneous n-ary tree. A structured object of any kind consists of a number of names, selectable by methods that depend on the structure: selection by field identifier in the case of structs, by indexing and traverser in the case of sequences, by traverser alone for trees. As previously described a component name has shareability, possibility and type as does the name of an ordinary variable. In a struct type these attributes are given by the corresponding field definition; in the homogeneous types component names are always bound and all have the same shareability as well as type.

\[
\text{structdec} = \text{`struct'} \text{ formalpara}_\text{pack};
\]

\[
\text{seqdec} = \text{`sequence'} [ [\text{`upto'}] \text{ integer_expression} ] \text{`of'} [\text{shareability}] \text{ formaltype [`traversed']
}
\]

\[
\{\text{identifier } [\text{`traverser_expr } \text{`finished'} \text{`)'}] / \text{`,} \}\};
\]

\[
\text{formaltype} = \text{type_identifier} | \text{ subrange} | \text{ union} | \text{ enumeration};
\]

\[
\text{id}_\text{list} = \{\text{identifier} / \text{`,}\};
\]

\[
\text{treedec} = \text{`tree'} [\text{`upto'} \text{ integer_expression}] \text{`of'} [\text{shareability}] \text{ formaltype `articulated'} \text{ id}_\text{list [`traversed'] id}_\text{list} [\text{`parameter'} id}_\text{list].
\]

3.2.6.1 Sequences

A sequence type may be fixed in size, varying but bounded, or unbounded. Example declarations are

\[
\text{VAR class: SEQUENCE number_of_names OF string;}
\]
VAR plist: SEQUENCE UPTO 36 OF person SHAREABLE population;
STATE: SEQUENCE OF person.

In any form of sequence, component names are selected by indexing and by traversers; they are added and removed in variable-sized sequences by "grow" and "shrink" statements.

statement = ... grow | shrink | trav_update | make_empty;
grow = "grow" (("at" | "after") traverser_expr | object_expression ("at" | "after") integer_expression) with"expression;
shrink = "shrink" ("at" traverser_expression) | (object_expression "at" integer_expression);

sequence_traverser_expr = ("first" | "last" | traverser_identifier) {"by" | "next"}*["of" object_expression];
trav_update = simple_trav_update | make_coincide;
simple_trav_update = ("reset" | "next" | "finish") simple_trav_expr;
simple_trav_expr = traverser_identifier ["of" object_expression];
make_coincide = "make_coincide" traverser_identifier "with" traverser_expr;
binary_expr = ... | coincides;
coincides = traverser_identifier "coincides" traverser_expr;
make_empty = "make_empty" object_expression.

3.2.6.1.1 Indexing

Indexing selects a component name by its ordinal number in the structure: e.g. "[x + 2] OF plist". The resulting name is always bound; it is an error if the object does not have sufficient components. It can be used quite generally as a name: for example
updates the fifth component object of \texttt{plist} to have equal value to "fred";

makes the object bound to "fred" replace the sixth component of "plist".

"Grow" and "shrink" add and remove component names, together with the objects bound to them. Growth AT an index value specifies the value which will select the new name; shrinkage specifies the index of the name to be removed. For example

\begin{verbatim}
GROW plist AT 1 WITH jim
\end{verbatim}

creates a new component name as the first one of \texttt{plist}, and binds the object bound to "jim" to that name. The previous first component, if any, becomes the second, etc. Alternatively, positioning growth AFTER the index:

\begin{verbatim}
GROW plist AFTER 1 WITH TOPCOPY (mabel)
\end{verbatim}

makes the copy of "mabel" become the second component, the first component being unaffected, the previous second becoming third, etc.

The inverse operation is "SHRINK", for example

\begin{verbatim}
SHRINK plist AT (i + j).
\end{verbatim}

Sequence objects model vectors, conventional sequences and strings, including stacks, queues, and deques. A bounded stack, for instance, may be defined by the following capsule:

\begin{verbatim}
CAPSULE haystack = (STATE:
  UPTO max height OF bale SHAREABLE stooks;
  is empty = SELF FUNC: BOOLEAN;
  BEGIN is_empty YIELD SIZE SELF = 0 END;

... (definition of methods) ...
\end{verbatim}

push = SELF PROC (newb: bale SHAREABLE stooks);
BEGIN GROW AT 1 WITH newb END;

pop = SELF FUNC : POSSIBLY bale SHAREABLE stooks;
BEGIN
  pop YIELD YIELDOF (  
    IF NOT is_empty() THEN YIELD {1} OF SELF;  
    SHRINK SELF AT 1  
  ELSE YIELD NONE  
  ENDIF)
END).

("SIZE" of a sequence is the number of components it has; "SELF" can be omitted in SHRINK and GROW statements, as it is in the definition of "push").

3.2.6.1.2 Traversers

Indexing is often an inappropriate way to articulate (access the components of) an ordered sequence in general. In some instances its components are never accessed "at random", but only in order; the states of a number of readers and writers over the sequence may form part of its own state description. For this reason a Koala sequence may be manipulated by traversers as well as indexed. A traverser is associated with a single sequence object, over which it acts as a reader and writer (i.e. a "sequencer" [Pratt 1975]). A list of traversers is declared as an attribute of a sequence; each object of that type possesses its own distinct set of those traversers.

In the data model the fixed function Tselect and the variable function Indicate provide the model for traversers. A traverser either indicates a component name of its associated object, or is finished and indicates no name.
For example, for an object of sequence type

```
CAPSULE personlist = (STATE:
SEQ UPTO 36 OF person SHAREABLE population
TRAVERSED current, printa(FIRST));
```

... VAR tplist: personlist <- ... ;

the snapshot of an instance might be

```
tplist

printa personlist (tselect) current
```

At this moment, the traverser "printa" indicates the first component, "current" the second.

A traverser is referred to in the same way as the field of a struct, e.g. "printa OF tplist". The name that it indicates is got by writing a detraverser expression as an object-expression head, e.g. "(# printa OF tplist #)" (equivalent to "[1] OF tplist" at the moment of the snapshot above). Thus for instance

```
(# current OF tplist #) <- mabel
```

changes the binding of the second component to the same as that of "mabel".

What a traverser indicates initially is undefined unless its declaration includes a bracketed traverser expression or the symbol
"FINISHED"; thus "printa" is initialized to indicate the first component of the sequence, "current" is initially undefined.

Traversers' values are updated by RESET, FINISH, NEXT and MAKE_COINCIDE operations. "RESET current OF tplist" makes the traverser indicate the first component name; successive evaluations of "NEXT current OF tplist" advance it over successive component names of the sequence. A make-coincide statement assigns one traverser's value to another, and can be meaningful only if the traversers belong to the one object. The syntax ensures this by allowing for only a single object expression in the statement: for example in

```
printa MAKE_COINCIDE current OF tplist;
```

"OF tplist" implicitly applies to "printa" as well as "current". The dyadic operator "COINCIDES" is true of two traversers which indicate the same name, or are both finished.

A traverser is "finished" when it indicates no component name. It may reach this state by being advanced from the last component name; by being RESET when the object is empty; by being the subject of an explicit FINISH statement; or by make-coincide assignment from a finished traverser. The monadic boolean operator "FINISHED" tests this state.

Traverser expressions may be assigned to traversers or used directly themselves. They take the form of any number of advances (written "BY NEXT") from a traverser variable or a primitive traverser-expression ("FIRST" and "LAST"); thus

```
(# current BY NEXT BY NEXT OF tplist #)
```
delivers the name of the component that is two in advance of that indicated by "current", and

IF FIRST COINCIDE current OF tplist THEN ...

is a convenient way to test for a traverser in its post-RESET state.

For example, consider a procedure to insert a person (no one person more than once) in a sequence of persons which is ordered by the string relation "less_than" on the names of the persons:

CAPSULE sorted person list =
(STATE: SEQUENCE OF person SHAREABLE population
TRAVERSED curr;
insert = SELF PROC (candidate: person SHAREABLE population);
BEGIN
RESET curr; (* i.e. curr OF SELF *)
WHILE NOT FINISHED curr
CAND (NOT (candidate <-> curr)
AND string $ less than (full name OF candidate,
full name OF (# curr #)))
DO NEXT curr ENDDO;

IF FINISHED curr
THEN
GROW AFTER LAST WITH candidate
ELSE
IF NOT (candidate <-> curr)
THEN
GROW AT curr WITH candidate
ENDIF
ENDIF
END;

Grow and shrink statements also apply to traverser expressions:
for example

GROW AT printa WITH candidate;

GROW AFTER curr BY NEXT OF some_list WITH fred.

The traverser expression in GROW or SHRINK must not be finished. The sole exception is growth from an empty sequence, in which the finished traverser expression FIRST is allowed. If an identified traverser is
used in a GROW statement, it continues to indicate the same component name as before, even though its ordinal number changes in GROW AT.

"SHRINK AT current" leaves the traverser indicating the successor to the removed name, or makes it finished if there is none.

This ability to alter the structure of an object while maintaining traversers over it can cause a "dangling traverser" problem: i.e. some traverser may attempt to indicate a component that has been shrunk out of the object by means of some previously coinciding traverser expression. This is prevented by requiring that before any size-changing operation (SHRINK or GROW or copying assignment) is executed on an object, all traversers on that object must be finished. If this is not true an error results. The named traverser, if any, in the operation's own traverser expression can clearly be safely and conveniently excluded from this. Note that the statement "FINISH printa" is available to force traversers to become finished instantly if necessary.

3.2.6.2 Trees

The Koala tree structures specify objects that are true trees, with no cycles or common sub-trees. The declaration specifies the type of the tree's components, identifiers for the articulation functions, and identifiers for the traversers. The syntax of the declaration is given at the head of section 3.2.6. For example, an inverted family tree:
The articulation of both tree and sequence is a property of their component names, independently of the objects bound to them. A particular person object may be bound to more than one name in a given "sequence of (shared) person", for instance, or more than once in a particular tree. Although this corresponds to no real-life situation modelled by an "ancestral_tree", a model of disciplinary hierarchies of persons does provide an example where a single person may hold two positions, each with its own set of superior and inferior positions: the relationships between positions are independent of the identity of the persons holding them. This notion of a "position" is exactly represented by a component name ("node") in a tree with its articulation relations.

The diagram of such a situation modifies our use of a single arc for the "select" relation between structured object and component name. In the diagram below, all names that may be reached by any path of "root", "mother", and "father" arcs from the representation of the "ancestral_tree" object are components of that object.
The only method of accessing a tree's components is by the use of traverser expressions. To manipulate traversers, in place of the sequence-traverser's "RESET curr" we have "REROOT ancestor"; to advance a traverser we must now specify which branch (articulation function) to follow from the current point, and in place of "NEXT curr" we have a choice of "ADVANCE ancestor BY mother" and "ADVANCE ancestor BY father". These were seen in the introductory example earlier.

The form of a tree object is modified only by growth and shrinkage of component names. "SHRINK AT ancestor OF my family VIA mother" removes the name indicated by "ancestor", and also the entire sub-tree of names accessed via the "father" relation of that name; the "mother" name remains in the tree, now at the position that the removed name
had. The VIA clause may be omitted to remove both sub-trees as well as the node itself. In GROW operations a VIA clause specifies the relation established between the new node and the old, whose related sub-trees remain intact; without a VIA clause the old node and its relatives are deleted.

### 3.2.6.2.1 Recursive operations

The manipulation of tree data structures in general is frequently done by recursive algorithms, which may access every node of the tree, for instance. The traversers described so far are incapable of visiting every node of the tree in a single pass, because they cannot be advanced "up" the tree to visit branches already by-passed, nor can a finite number of traversers save enough states for one to traverse an arbitrarily deep tree. The use of more than one traverser would also preclude any shrinkage or growth during the traversal, which would be a considerable drawback.

Parameter traversers are a consequence of wishing to use recursive operations to traverse and update trees selectively with complete security. A second kind of traverser declaration combined with a new form of formal parameter to operations has the required properties.

A parameter traverser is declared by the clause headed "PARAMETER". While indicating a component name of an individual tree object, each parameter traverser is at the same time a traversing parameter of one or more operation definitions marked "TRAVERSING". For example, to traverse a whole tree in symmetric order:
CAPSULE int_bitree = (STATE: TREE OF integer
                     ARTICULATED left, right
                     TRAVERSED constructor
                     PARAMETER printer, eql1, eql2;

print_in_order = TRAVERSING PROC [printer] (follower: character);
   BEGIN
      IF NOT FINISHED printer
      THEN print_in_order [ADVANCING left] (follower);
         integer $ print ((# printer #));
         character $ print (follower);
         print_in_order [ADVANCING right] (follower)
      ENDIF
   END).

The operation's formal traversing parameter "printer" saves the
state of the parameter traverser with the same identifier as it was on
entry to the operation. That state is accessed by their common iden-
tifier, as in

   integer $ print ((# printer #)).

The actual parameter for a traversing operation's call specifies this
state by an expression which is an advance relative to the traverser's
current state. Such a relative traversal expression is a list of
"ADVANCING articulation_identifier" elements. For example

   print_in_order [ADVANCING left]

in the example above calls the operation recursively, advancing
"printer" to "printer BY left";

   print_in_order [ADVANCING left ADVANCING right]

would make a double advance for this call.

A call of a traversing operation with an absolute traverser
expression as actual parameter is used to initialize the traverser's
state. The same traverser can subsequently be thus re-initialized
only when the operation last called has returned, i.e. when there are
no saved states of this parameter traverser.

E.g. print_names [ROOT OF my_family];
print_in_order [constructor BY left OF some_tree_var];
IF equal_trees [ROOT OF x, ROOT BY left OF y] THEN ...

where the latter operation might be defined as

\[
equal\_trees = TRAVERSING\ A\ FUNC\ [eql1, eql2] : BOOLEAN;
BEGIN
IF FINISHED eql1 AND FINISHED eql2
THEN equal\_trees YIELD TRUE
ELSE IF FINISHED eql1 OR FINISHED eql2
THEN equal\_trees YIELD FALSE
ELSE IF NOT (\# eql1 \#) = (\# eql2 \#)
(* compare component objects *)
THEN equal\_trees YIELD FALSE
ELSE equal\_trees YIELD
equal\_trees [ADVANCING left, ADVANCING left] CAND
equal\_trees [ADVANCING right, ADVANCING right]
ENDIF ENDIF ENDIF
END.

Traversing operations are implicitly also self-operations, and
any traversing operation can be called from within another with the
same traversing parameter(s). Mutual recursion is thereby possible.

Every simple traverser over a tree or sequence object must be
finished before the object is structurally changed, to avoid dangling
traversers. So must every state saved by a parameter traverser. The
exception is the traverser used in specifying the change.

The restrictions on the manipulation of parameter traversers
ensure that
(1) a single traverser can traverse the whole of a tree by advancing in turn by all the articulation functions from each node;

(2) security is preserved in traversal, a traverser being unable to indicate names other than those in its own tree object;

(3) security is preserved in updating, because none of the states saved can be shrunk from the tree by any operation on the current state of the traverser: all are "higher" in the tree.

3.2.6.2.2 A tree example

For an example of the use of such trees, consider the representation of simple unary and binary arithmetic expressions by a binary tree with components of type

\[
\text{UNION (const: constant, variable: variable SHAREABLE variables, \ rator: operator).}
\]

Unary and binary expressions consist of an "operator", with one or two arithmetic expressions respectively; they are labelled "lefrtrand" and "rightrand" (a unary "operator" has a "rightrand" only). "Variable" and "constant" nodes are leaves of the tree. (This relationship between the actual type of the object bound at a node, and the number of branches that actually exist at that node, cannot be made explicit with Koala trees.) We define an operation "simplify" on this type to evaluate constant sub-expressions, and simplify expressions which add zero or multiply by one. The operation is a self-operation which takes a shared parameter - a table giving current values for "variables".
CAPSULE arith_expr = (STATE: TREE OF
  UNION (const: constant,
    varble: variable SHAREABLE variables,
    rator: operator)
  ARTICULATED leftrand, rightrand
  TRAVERSED builder, evaluator
  PARAMETER simpexpr;

simplify = SELF ANY PROC
  (current_vals: value_table SHAREABLE global_tables);
  BEGIN
    rec_simple [ROOT] (current_vals)
      (* i.e. ROOT OF SELF *)
  END;

rec_simple = TRAVERSING PROC
  [simpexpr] (a_list: value_table SHAREABLE global_tables);
  BEGIN
    (* we do not expect simpexpr to be FINISHED *)
    WITH (# simpexpr #) DISCRIMINATE
      (* i.e. the component at the node *)
    WHEN op: TOTALCOPY rator THEN
      (* simplify operands of this rator *)
      rec_simple [ADVANCING rightrand] (a_list);
      IF NOT FINISHED (simpexpr BY leftrand)
        THEN
      ENDIF;
    CASE op IN
      WHEN "*" THEN
        (* a (binary) multiplication *)
        WITH (# simpexpr BY leftrand #) DISCRIMINATE
          WHEN cleft: TOTALCOPY const THEN
            IF cleft = NEW constant(0)
              THEN (* product is also zero *)
                SHRINK AT simpexpr VIA leftrand
            ELSE IF cleft = NEW constant(1)
              THEN (* product is other operand *)
                SHRINK AT simpexpr VIA rightrand
            ELSE (* is right op a constant also? *)
              WITH (# simpexpr BY rightrand #) DISCRIMINATE
                WHEN cright: TOTALCOPY const THEN
                  (* replace expression by evaluated product *)
                  (# simpexpr #) <- const : NEW constant
                    (cleft.value * cright.value);
                  SHRINK AT rightrand OF simpexpr;
                  SHRINK AT leftrand OF simpexpr;
                ENDWHE
ENDDISCRIM
ENDIF ENDIF
ENDWHEN (* left operand is constant *),
 OTHERWISE
(* similarly at the right operand *)
WITH (# simpexpr BY rightrand #) DISCRIMINATE
WHEN c2right: TOTALCOPY const THEN
IF c2right = NEW constant(0) THEN
SHRINK AT simpexpr VIA rightrand
ELSE IF c2right = NEW constant(1) THEN
SHRINK AT simpexpr VIA leftrand
ENDIF ENDIF
ENDWHEN
ENDDISCRIM
ENDDISCRIM
ENDWHEN (* op is "*" *),

WHEN "-" THEN
(* may be binary or unary expression *)
IF FINISHED (simpexpr BY leftrand) THEN (* unary minus *)
WITH (# simpexpr BY rightrand #) DISCRIMINATE
WHEN op2: TOTALCOPY rator THEN
IF op2 = "-" THEN
IF FINISHED (simpexpr BY rightrand) BY leftrand THEN (* operand is also unary minus. Make the replacement
\((-(-x))) \Rightarrow (x)\) *)
SHRINK AT simpexpr VIA rightrand;
SHRINK AT simpexpr VIA rightrand;
ELSE (* we would like to replace \((-(-x-y))\) with \((y-x)\), but cannot swap
left and right sub-trees as we would need to *)
ENDIF
ENDIF
ENDWHEN
ENDDISCRIM
ELSE (* binary minus - similar to "*" *)
ENDWHEN (* simpexpr is an operator *),

WHEN avar: var SHAREABLE variables THEN
(* substitute its current value *)
simpexpr <- const: lookup(avar) OF a_list;
ENDWHEN
(* when a constant then cannot simplify *)
ENDDISCRIM
END ) ;
VAR expr : arith_expr <- ... ;
VAR some_values: value_table SHAREABLE global-tables <- some_others;
simplify (some_values) OF expr.

3.2.7 Other syntactic constructs

The yield statement is the remaining unconventional statement in Koala. It allows a compound statement to yield a value, as if it were an in-line function body:

n := YIELD OF ( IF x >= maxx THEN YIELD maxx ENDIF;
X := X + l•
YIELD x ) ;

The type of all yields in a yield statement must be the same, and their shareabilities compatible; the shareability of the expression result is the "highest" among its yields, in terms of the shareability acceptability diagram [section 3.2.1.2.2].

3.2.8 Discussion

Koala's design is founded on two requirements within the constraint of being a mini-language for investigating data structuring:

1) it defines a high-level secure data space abstracted from implementation considerations;
2) it is one extreme of a Broad Spectrum Language, within whose spectrum its programs will be transformed.

The first requirement is met by basing the language on a strongly declarative use of a referenceless abstract data space, and on the textual strength of its operations. This latter is important for the second requirement also, as we shall see in the next chapter. But until we have described the rest of the Broad Spectrum we can consider only the first requirement, and here the most important feature of Koala is its referencelessness. The reasons for wishing to omit references from an abstract language have been argued above [section 3.2]; the examples since then show that a referenceless language is feasible, and that any additional complexity and loss of references’ raw power is compensated by increased security.

We now consider some of the ramifications of how these design aims are met in Koala, concluding with a comparison with the conventional uses of references.

3.2.8.1 Dynamic flexibility

Two distinctions made strongly in many languages are here both blurred: differences in lifetimes for storage allocated for data objects, and differences between objects of fixed and variable sizes.

(1) No strong distinction is made between "static" and "dynamic" storage allocation (as is made in Algol 68, Pascal and Simula 67, for instance): logically, all storage allocation in Koala is dynamic and by object. The division of objects into stack and heap allocation
categories is an implementation decision, closely tied to representation, and is therefore not part of the abstract language Koala. Distinguishing categories of object lifetime is not an issue addressed in the Koala design; if it had been, the Algol 68 solution of modified generators would have been favoured.

(2) The distinction between objects of fixed and variable size is weak in Koala: no major change of form is required to specify or manipulate a sequence of variable length as against one of fixed length. The differences in the abstract are not so great, compared with the common features, as to justify a distinct notation. This distinction is a source of confusion in conventional languages, where its main reason for existence is the quite different implementations to which the cases reduce, a concern deliberately factored out of Koala.

3.2.8.2 Strong textual correspondence

A strong correspondence between textual forms and properties of objects is designed into Koala. This shows in its strongly declarative nature (strong typing, and similarly strong shareability) and strong control of manipulations (the discrimination of unions, for example). The result is that programs are more readily understood both informally and by formal techniques, and redundancies are introduced that make programs more secure. Further, statements concerning the program's surface structure, and manipulations of that structure, correspond directly to statements and manipulations of its syntax.
This property makes the language suitable for the program transformations that are necessary to express representation in general, as we shall see in Chapter 4.

3.2.8.3 Sharing

The control of aliasing is an important issue in programming languages, because uncontrolled aliasing through references is a major source of difficulty in understanding and controlling programs. Suggested solutions in the literature are (i) prohibit aliasing completely [Kieburtz 1976]; (ii) restrict the domain of a reference type (and hence possible aliasing) to a smaller subset of objects than a type [Euclid; Ada]; (iii) constrain aliasing within a single module [Berry et al. 1976]. In Koala the problem is reduced by distinguishing some of the more restricted forms of aliasing (those which appear in traversing structured objects) from the more general, and containing the use of the former within a capsule. A particularly error-prone form of aliasing is thereby eliminated, resulting in improved clarity and security. For the remaining cases of aliasing we choose the "collection" solution in the form of the comparatively unrestraining "sharegroup".

The effectiveness of sharegroups in ensuring program security depends to some extent on their wise use by the programmer, and cannot be shown in the compass of brief examples. But it is obvious that the use of as many sharegroups as possible in a program breaks the population of objects into smaller and hence more intellectually manageable groups: the size of each group and hence the range of possible aliases
should be no larger than the abstract program strictly requires. This practice is encouraged by making the definition of a sharegroup and its members as easy as possible for the programmer. The ease of definition stems from separating all notions of distinct typing from those of sharegroup membership. The properties of a type are defined in a capsule: its set of values (which allows value-copying and comparison between all its instances) and its abstract operations. Aliasing is a property of the variables, parameters and object components that bind to the type's objects, not of the type itself.

3.2.8.4 Selective update, value assignment

Koala supports the view that selective updating is a natural way to express the manipulation of complex data. This contrasts with a frequently stated view (e.g. Hoare [1975a]) that selective update of structured objects' components is a way of expressing, for efficient naive implementation, an operation with semantics we can paraphrase as "replace the current value of this whole variable with a fresh value, different from the first only in the particular respect indicated". In this view, for example, in the Algol 68 fragment

```plaintext
COMPL c; c := (21.5, -13.2); im OF c := 17.9
```

(where COMPL is the mode "STRUCT (REAL re, im)"), the selective update of the "im" field of c is equivalent to the whole-variable update "c := (re OF c, 17.9)". This view of updating is often valuable; it has convenient consequences for formal proof techniques, for instance. The presence of sharing and structured objects with components that are themselves structured makes this explanation untenable, however:
it is no longer clear which "value" is being replaced, since the
establishment of "d is an alias of c" implies that the subsequent
selective update
\[
\text{im OF } c := 17.9
\]
means
\[
\begin{align*}
\text{BEGIN } & c := (\text{re OF } c, 17.9), \\
& d := (\text{re OF } d, 17.9) \text{ END};
\end{align*}
\]
i.e. the value that is replaced is the union of all aliases. Where
aliasing may be established dynamically, the only sufficiently general
notion to explain a selective update, therefore, is that the "value"
involved is the entire state of the computation, which is replaced for
every assignment or small change to any data item. While this expla­
nation is certainly not false, it discards the cartesian factoring of
the data space description (the environment into variables, structures
into components) which was used to specify the selective update and
which is essential for the intellectual manageability of many pro­
grams.

This is seen particularly in the case of trees and sequences.
Selective updating of trees is often significantly clearer than their
decomposition and reconstitution by recursive procedures (in the
time-honoured Lisp fashion, e.g. Hoare [1975a]). The changes made to
the structure are specified explicitly, rather than being implicit in
any differences between the decomposition and reconstruction.

Selective updating of union-typed variables is possible with com­
plete security by Koala's union-discrimination construct, applying
name and binding semantics. Shared bindings in its discrimination-
statement branches solve the "dangling union" problem cleanly: i.e.
the problem that updating the variable being discriminated while con­
trol is within a particular discriminant branch may invalidate the
supposedly secure view of its type, which violates strong typing.
Algol 68's solution is to forbid (somewhat deviously) any updating of
the discriminated value, which is always a copy of the union value. An
alternative solution is to lock the union variable dynamically against
type-changing updates while a discrimination is in progress, but
besides reducing the power of the language, this violates the princi­
ple of strong textual correspondence in Koala's design.

It is inappropriate, however, to extend the notion of selective
update to every operation on all data types. Some data types, such as
integer, real, boolean, and all enumeration types, are best viewed as
whole values. Copying assignments for entire structured objects are
also present in Koala because not all manipulation of values is selec­
tive (consider "COMPL z := c" rather than "im OF z := im OF c; re OF z
..."). The two forms of copying assignment reflect two notions of
modelling data with objects and selection relations.

(1) One form of abstract entity can be modelled as a network of
objects: a Koala data object and all objects which are its components,
components of these and so on. For example, a struct modelling a
motor car's construction may have as its components descriptions of
the car sub-assemblies, which have their own components and so on down
to nuts and bolts. The abstract "motor car" is the whole collection
of these objects, with the structure imposed on them by the component
relations of the Koala data objects. The appropriate method of
copying such a car is by TOTALCOPY, and "car" variables would be total-copy assigned.

(2) Alternatively, an abstract entity can be an element in a network of relationships that are regarded as distinct from the entity. For example, a "person" is described as a struct, some of whose field identifiers (e.g. "spouse") can be regarded as denoting relationships to other "persons". This relationship to spouse is an attribute of the "person" being modelled; the entire connective network of such relationships taken transitively (and including "parent", "employer", say), however, is not. TOPCOPY and top-copying assignment of a person capture this model of data. The distinction between component objects which form part of the abstract object, and its external relationships to other objects, is made on the basis of which components are shareable.

3.2.8.5 Trees and sequences

We have taken a repetitive rather than a recursive view of extensive data structures because sharing and security require the attributes of the whole structure to be different from those of its constituents. The security of a tree abstraction has two aspects: maintaining its true tree properties, preventing the formation of cycles or common sub-trees; and permitting traversal and updating of the structure with clear (and therefore secure) programming constructs.

Hoare [1975a] and Kieburtz [1976] address this problem, dealing also with the finite-valued structures of proper trees and sequences.
as examples. Hoare applies a discriminated union construction in a recursive type definition to avoid infinite recursion in instances of the type. Kieburtz does much the same with more mathematical formality and a type-marker playing much the same role as POSSIBLY. In both cases the manipulation of the tree-structured values so defined is constrained to allow a variable to have as value not a node of the tree, but an entire tree; but the formation of cycles and lattices is prevented because the construction of trees from component trees is by copying their values not binding their identities.

Hoare expresses traversal by pure functions for recursively decomposing a tree, and selective update by reconstructing a fresh tree from its parts. Structural security is ensured by tying the manipulation of the tree very tightly to these recursive functions, by means of the semantics of copying sub-trees in constructors. But this allows no separation of the abstract process of traversing the tree from specific actions taken during specific traversals, including the construction of an updated version. The difficulties of expressing selective update in this way have been argued above.

Kieburtz allows a single "selector variable" for each structure, and this variable behaves very like what we have called a traverser. A selector variable may be moved over a single structure by a constrained form of assignment (e.g. "L SELECTS L[right]" is equivalent to our "ADVANCE L BY RIGHT") and reset to the root of that structure ("RESET L"). Selective update within the tree is expressed by assigning copies of trees to this selector variable and its expressions: e.g. the statement "L[left], L[right]:=L[right], L[left]" is a
parallel assignment swapping the two branches of the binary tree currently selected by L. \(\{(L)\}\) is the basetype value at the root of the selected tree, equivalent to our detraverser \((\# L \#)\).

Koala relaxes Kieburtz's constraints (which forbid any aliasing) to allow multiple traversers per object, removing an irksome limitation on the programmer. Because traversers are declared as a property of the tree type, and not as freely as normal variables, they maintain strong correspondence of program text and computation by making plain in the program the close relations between all of one object's traversers, and between traversers and the object that they traverse. Koala also allows traversers closely coupled to recursive procedures, but these also remain tightly associated with an owning object. Secure sharing of components and selective updating without necessarily copying are achieved by Koala's separation of the tree's structure (a tree of names) from its component objects (variably bound to those names).*

3.2.8.6 Reference replacements

We may ask, with Kieburtz [1976], "where have all the references gone'? Koala makes finer distinctions in their uses than those he mentions (i.e. inverted files, linked dynamic structures, data graphs, and sharing without copying), and the Broad Spectrum Language allows us to distinguish further between the use of references that have a purely representational purpose and the use of those that make logical

* We have omitted from this description of trees operations to grow with top-copies of sub-trees rather than simply with single components at a time; merely syntactic additions are needed for this.
distinctions in the abstract. Those uses that we identify as "logical" and which therefore might appear in Koala are:

- the distinction between static and dynamic storage allocation (which normally finds great application in linked representations of dynamic data structures). This distinction has not been made here.

The accompanying differences in programming language notation largely result from the differences in implementation, and removing them does not decrease expressive clarity in a higher-level language.

- using fixed-size items with reference components to form flexibly sized linear and higher order recursive structures, and independent reference variables and parameters to traverse and manipulate these structures. These uses are replaced by flexible sequences and trees which have their own traversers and traverser parameters.

- denoting the possible nonexistence of an attribute or an association, or the limits of a linked data item, by using a specific NIL reference value. The repetitive data type mechanism described here implicitly includes proper termination of recursive data structures without use of references, while variables, selectors etc. whose binding may possibly be deferred or deleted are merely noted with the modifier POSSIBLY. FINISHED and NONEX denote some of the states denoted by a NIL reference value.

- the distinction between objects and values and the establishment of aliasing or sharing, normally available through the mechanism of parameter passing by simple name as well as the dynamic manipulation of reference variables. Koala makes sharing explicit by
distinguishing copying assignment from sharing-establishing assignment with distinct sets of operators, and controls the extent of sharing declaratively.

In conventional programming practice, a single reference variable may combine several of these functions, making the Koala equivalent appear more lengthy. The benefits of distinguishing the logical uses of references, and further separating them from representational uses, are expected to be great, however. The analogy between references and gotos referred to earlier holds here, and in both cases clarity of intent and separation of functions are more important than brevity of expression.

Because it is part of SSL, Nacian has a common semantic and semantic basis with Koala. The similarities in outward form are obvious; there is also a central description of what data needs to remain.

Many of the differences between the two are in the constructs. Nacian imposes on the freedom in Koala's semantics, which are reflected in correspondingly simpler syntax. Structured objects are fixed in once created; the sequence type constructor is allowed only in its fixed-bound form, and there is no free declaration. Bindings cannot be freely manipulated; there is no binding assignment operator; no union declaration, no possibility of a reuse being subsumed. The remaining data types have less power; there are no transformers. Not Nacian adds type constructors and manipulations that are not in Koala, namely reference and overlay union types.

These restrictions and extensions to Koala give Nacian a data space that can be mapped almost directly onto the machine. The Nacian data space is closely comparable to that of Pascal's scalar, pointer, second and one-dimensional array types, extended to have the semi-symmetric determination of Algol 60's array bounds. Thus Nacian can be described as "machine-oriented", albeit at a higher level than is
3.3 Maclan - a machine-oriented language

Maclan is the machine-oriented end of the Broad Spectrum Language. Its underlying data space is therefore simpler than that of Koala. But its declarative forms are more complex because of the addition of the "bundle", which encloses a number of related capsules and the operations between them.

Because it is part of BSL, Maclan has a common syntactic and semantic basis with Koala. The similarities in outward form are obvious; there is also a common description of objects bound to names. Many of the differences between the two are in the constraints Maclan imposes on the freedoms in Koala's semantics, which are reflected in correspondingly simpler syntax. Structured objects are fixed in size once created; the sequence type constructor is allowed only in its fixed-bound form, and there is no tree declaration. Bindings cannot be freely manipulated; there is no binding assignment operator, no union declaration, no possibility of a name being unbound. The remaining data types have less power; there are no traversers. But Maclan adds type constructors and manipulations that are not in Koala, namely reference and overlay union types.

These restrictions and extensions to Koala give Maclan a data space that can be mapped almost directly onto the machine. The Maclan data space is closely comparable to that of Pascal's scalar, pointer, record and one-dimensional array types, extended to have the semi-dynamic determination of Algol 60's array bounds. Thus Maclan can be described as "machine-oriented", albeit at a higher level than is
usual for this term.

3.3.1 Objects, values and bindings

In Maclan the BSL data model is constrained to give variables and structures much more conventional behaviour: a newly created name is unchangeably bound to an unshared object, and the number of component names of a structured object is fixed at the moment of its creation. Dynamically variable relations between objects are only possible if represented by the class of "reference" types, comparable to a Pascal pointer. Loosely, the relationship of a reference value to the object bound to the name to which the value refers will also be called "refer to".

3.3.2 Bundles, sequences, references

We can demonstrate some features of Maclan with a bundle of representations of a queue of integers. A bundle may contain an abstract definition in the form of a capsule expressed in Koala:

```koala
BUNDLE queue_int = [
    CAPSULE ABSTRACT =
        (STATE: SEQUENCE UPTO max_queue OF integer
         TRAVERSED inspector;

        initialize = SELF PROC; BEGIN MAKE_EMPTY SELF END;

        insert = SELF PROC (joiner: integer);
        BEGIN GROW AT LAST WITH TOTALCOPY joiner END;

        remove = SELF FUNC: integer;
        BEGIN remove YIELD TOTALCOPY FIRST;
        SHRINK AT FIRST
        END;
```

is_present = SELF FUNC (key: integer) : boolean;
  VAR found: boolean := false;
  BEGIN RESET inspector;
    WHILE NOT found AND NOT FINISHED inspector
      DO found := (# inspector #) = key;
        NEXT inspector;
      ENDDO;
    is_present YIELD TOTALCOPY found
  END

The queue is bounded, and is described by a bounded sequence on which
new members are grown at the sequence's LAST (the tail of the queue)
and removed from FIRST (the head). The members of the queue are not
shared and so are handled by copying. The function "is_present" uses
the traverser "inspector" to compare the queue members with a given
"key" value.

3.3.2.1 Structs, sequences, operations in Maclan

One implementation of a bounded queue is as a circular buffer.
This requires a fixed vector of the same bounds, and an index variable
that indicates the current "front" of the queue within the buffer;
another variable contains its length. Together these two indicate the
position of the tail of the queue.

 BUNDLE queue_int = [
  CAPSULE ABSTRACT = (...);

  CAPSULE circular =
    (STATE: STRUCT (cbuff: SEQUENCE max_queue OF integer;
                    length: 0 .. max_queue;
                    front, inspector: 1 .. max_queue);
    initialize = SELF PROC; BEGIN length := 0; front := 1 END;
insert = SELF PROC (joiner: integer);
VAR tail: integer :=
  (front + length - 1) MOD max_queue + 1;
BEGIN
  IF length = max_queue
  THEN error ("queue overflow")
  ENDIF;
  [tail] OF cbuff := joiner;
  length := length + 1
END;

remove = ... ;

is present = SELF FUNC (key: integer) : boolean;
VAR found: boolean := false;
VAR i : integer := 0;
BEGIN inspector := front;
  WHILE NOT found AND i < length
  DO found := [inspector] OF cbuff = key;
     inspector := inspector MOD max_queue + 1;
     i := i + 1
  ENDDO;
  is present YIELD TOTALCOPY found
END

The Koala STRUCT constructor is unchanged in Maclan. The SEQUENCE constructor is still available, but although the type constructor takes the same three forms (fixed, bounded, and unbounded) as in Koala, its objects are always of fixed size from the time of their creation. In the fixed-length sequence seen here the bound on any "cbuff" component is the value of the global variable "max_queue" at the time a "circular" variable declaration is evaluated. An instance of the bounded or unbounded forms gets its fixed size from the number of components in its first assigned value. The Maclan sequence has no traversers, but is indexed in the same way as in Koala. The GROW and SHRINK operations used in the abstract capsule are clearly not possible in Maclan.
Parameters are passed to operations by binding in both Koala and Maclan. An actual parameter in Maclan can therefore only be newly generated, since the same rules of binding apply and it cannot be shared:

```plaintext
VAR one_q: queue int $ circular; VAR x: integer;
...
initialize () OF one_q;
insert (3) OF one_q;
IF is present (TOTALCOPY (x)) OF one_q ...
```

Sharing is possible only for the SELF of self-operations. This form of sharing cannot be manipulated directly in either part of the spectrum.

### 3.3.2.2 References and sharegroups, visibility

Maclan's reference types are qualified by a sharegroup as well as by a type. This is for the same reasons of control of sharing as holds for Koala, though the sharing in Maclan is by variables having equal reference values rather than identical bindings. A representation of the queue by linked nodes illustrates reference types:

```plaintext
BUNDLE queue_int =
  CAPSULE ABSTRACT = (...);
  CAPSULE circular = (...);

  CAPSULE linked =
    (SHAREGROUP links OF lnode;
     CAPSULE lnode =
       (STATE: STRUCT (item: integer;
                      next: NREFTO (lnode SHAREABLE links)));

     STATE: STRUCT (qhead: REFTO (lnode SHAREABLE links));
     qtail: REFTO (lnode SHAREABLE links);
     inspector: REFTO (lnode SHAREABLE links));
```
initialize = SELF PROC;
  BEGIN qhead := REFTO NEW lnode (DONTCARE, NIL);
  qtail := qhead
  END;

insert = SELF PROC (joiner: integer);
  BEGIN item OF FOLLOW (qtail) := joiner;
    next OF FOLLOW (qtail)
      := REFTO NEW lnode (DONTCARE, NIL);
    qtail := next OF FOLLOW (qtail)
  END;

remove = ...

is present = SELF FUNC (key: integer) : boolean;
  BEGIN inspector := qhead;
    WHILE NOT found AND NOT inspector = qtail
      DO found := item OF FOLLOW (inspector) = key;
        inspector := next OF FOLLOW (inspector)
    ENDDO
  END

There are two forms of reference type constructor, both applied here: REFTO and NREFTO. Each is qualified by a type and a sharegroup, which define the class of objects to which the reference values may refer. All the reference variables in this capsule are qualified in the same way: to refer only to "lnode" objects of the sharegroup "links". The difference between the constructors is that types with NREFTO include among their set of values a NIL value, which refers to no object at all. REFTO types do not include this value and therefore (unlike NREFTO and Pascal pointer types) their values always refer to some object.

Assignment in Maclan is only total-copying and is possible only for simple types. Assignment compatibility depends on structural
equivalence of type declarations (whereas Koala requires name equivalence). REFTO and NREFTO types with the same qualification are therefore subranges of the same type, and compatible. Thus "qhead", "qtail" (both of type REFTO (lnode SHAREABLE links)) and "next" (NREFTO (lnode SHAREABLE links)) are assignment compatible. The result is undefined if a NIL-value happens to be assigned to a REFTO variable, however.

The symbol REFTO is also used to generate reference values. The object referred to must be declared shareable, with the same share-group as the desired reference, or else have generated shareability, as in this example:

\[
\text{qhead} := \text{REFTO NEW lnode (DONTCARE, NIL)}.
\]

Extending the "linked" capsule might require a variable to be shared by a reference, for instance:

\[
\text{VAR lvar: lnode SHAREABLE links;}
\]

\[
... \text{qhead := REFTO lvar;}
\]

De-referencing is always indicated explicitly, as "'follow'
"'reference_expression')". It yields the name of the object referred to, causing an error if the reference is NIL; the name has the shareability of the reference's qualification. The expression

\[
\text{REFTO FOLLOW (qhead)}
\]

is therefore legal and synonymous with "qhead".

DONTCARE (used in "insert") is an explicit denotation for an undefined value of any type, useful in generating objects where there is as yet no need for any particular value.
The "linked" capsule uses the type "lnode" to link the queue members. It is declared locally to capsule "linked", and happens in this case to have no declared operations. In Koala the internal structure of such a capsule would be hidden, and no use could be made of it; but in Maclan these visibility restrictions are dropped, and all declarations exported from a capsule. The "item" and "next" fields of "lnode" objects can thus be referred to directly in the operations of the outer capsule, "linked".

3.3.3 Overlay union

Maclan's overlay union type is undiscriminated, allowing the programmer to interpret an object's value as being of any one of a set of possible types. The result is however undefined unless the value most recently assigned to the object was of the same type. The type constructor is OVERLAY, applied to a list of formal types.

A variation on the previous example might have a discriminated (Koala) union type in its abstract definition:

```
BUNDLE queue2 = [
    CAPSULE ABSTRACT =
    (STATE: SEQUENCE UPTO max_queue OF
      UNION (number: integer;
        fruit: apple);
    ...
)
```

If we can assert that one property of this type is that all members of one queue have the same type (either all integers or all apples) then a possible representation is in the form
The state of the "linked2" capsule carries a "qsort" component that the programmer uses to record the kind of objects in each instance by a value of the enumeration type "(ints, apples)". To assign an integer value to an overlay union object such as "item OF FOLLOW (qhead)" a normal total-copy assignment is used:

\[
\text{item OF FOLLOW (qhead) := 16.}
\]

To refer to the value of the overlay union object, on the other hand, requires an explicit cast to the desired type. We use QUA for this, as in Simula 67:

\[
\text{integer$ print (item QUA integer OF FOLLOW (qhead)).}
\]

3.3.4 Generic operations

A bundle may contain operation definitions as well as capsules defining various representations of its abstract capsule. Bundle operations have the special property of being allowed generic types for their parameters, function results, and local variables. This means that they can be applied to objects of different types, as long as these belong to the bundle. Bundle operations also have the property of being immediately visible outside the bundle (where their names may be overloaded), unlike its capsules which must be named in the composite form "bundle_identifier `$` capsule_identifier". For
example, we can add a "head_compare" function to the "queue_int" bundle shown earlier. The function removes the head of each of two queues and decides whether the removed values are equal:

```plaintext
BUNDLE queue int = [
  CAPSULE ABSTRACT = (...);
  CAPSULE circular =
    (...;
      insert = SELF PROC (joiner: integer);
      ...
      remove = SELF FUNC : integer;
      ...);
  CAPSULE linked =
    (...;
      insert = SELF PROC (joiner: integer);
      ...
      remove = SELF FUNC : integer;
      ...);
  head compare = FUNC (left : le FROM BUNDLE;
                      l.
                      right: re FROM BUNDLE): boolean;
BEGIN
  head compare YIELD remove() OF left = remove() OF right
END;
].
```

The formal parameters of the function "head_compare" are of generic types, denoted in general by "identifier 'from bundle". A bindable construct of generic type accepts any of the types defined by capsules of the containing bundle, in this case any objects of "linked" or "circular" type. Thus with variables declared to be of both types (these declarations may lie outside the bundle):

```plaintext
VAR ac: queue int $ circular;
VAR al: queue int $ linked;
VAR bl: queue int $ linked;
VAR bc: queue int $ circular;
```

the function can be called with any mix of parameters:

```plaintext
head_compare (TOTALCOPY(ac), TOTALCOPY(bc))
head_compare (TOTALCOPY(bl), TOTALCOPY(ac)).
```

Within a generic operation definition the identifiers attached to the generic types may be used as type identifiers. In a particular
call of the generic operation such an identifier denotes the type of
the corresponding actual parameter, and may be used to require that
another parameter also have the same actual type, as in

\[
\text{same\_rep\_equals} = \text{FUNC} (\text{left: } t \text{ FROM BUNDLE; } \\
\text{right: } t ) : \text{boolean;}
\]

or to declare the type yielded by a function, or the type of a local
variable:

\[
\text{copy} = \text{FUNC} (\text{arg: } \text{arg\_type FROM BUNDLE}): \text{arg\_type; }
\text{VAR local\_temp: } \text{arg\_type; }
\text{BEGIN initialize()} \text{OF local\_temp; }
\text{WHILE arg\_type $ length (TOTALCOPY(arg)) > 0 }
\text{DO insert (remove() OF arg) OF local\_temp }
\text{ENDDO; }
\text{copy YIELD local\_temp }
\text{END.}
\]

This example further supposes a "length" function in each capsule of
the bundle, whose heading in capsule "circular", for instance, is

\[
\text{length} = \text{PROC (q: circular): integer.}
\]

The body of the generic function "copy" applies various self-
operations to its generically-typed parameter and local variable:
"initialize", "insert", and "remove". Calls of "copy" with different
types of actual parameter invoke the named operations from the cap-
sules defining those types. Thus "copy(bl)" calls "length", "initial-
ize", "insert", "remove" from the "linked" capsule; "copy(bc)" calls
those operations in "circular".

The statement assigning the function a value is of note here:

\[
\text{copy YIELD local\_temp.}
\]

According to the rules given so far this statement is illegal, because
the yield of "copy" and the variable "local\_temp" are not declared to
be shareable (so the statement would be illegal even in Koala), and in
Maclan the binding established by yield assignment is normally forbidden to create sharing anyway. However this case is an example of an extension to this rule: where an object is unshareable but bound to a variable local to a function, it may be "transiently shared" and bound as yield of that function. Although this establishes a condition of "sharing", in that the names of the function's yield and its local variable are bound to the one object, it does not constitute "aliasing": there is at no time more than one expression denoting that object.

The result type of a generic function can also be generic; the context in which the function is called determines the actual type of this generic. A more widely applicable "copy" operation, for example, takes a parameter of any representation and returns a copy in the representation required by the context:

\[
\text{any_copy} = \text{FUNC (arg: source FROM BUNDLE): dest FROM BUNDLE.}
\]

The importance of the form, scope and overloading of generic operations is that they permit the bundle to contain definitions of copying assignment, equality, and generating operators that apply between objects of all representations in one bundle. None of these operations is supplied as primitives for any structured types in Maclan. The definitions are functions and procedures with the appropriate symbols as identifiers, where the equality operations are boolean functions and the assignment operations are two-parameter procedures:

\[
\begin{align*}
\text{BUNDLE queue_int} &= [ \\
\text{CAPSULE ABSTRACT} &= (...); \\
\text{CAPSULE linked} &= (...);
\end{align*}
\]
The parameters here have ANY shareability, permitted only for
generic parameters. Without this no calls of the operations such as
"bc := al" or "TOTALCOPY(ac)" would be legal; their actual parameters
could not have compatible shareability, for all possible actual types,
with the formal parameter. General forms for the bodies of these
operations, and the articulating operations they require in each cap-
sule of the bundle, will be described in section 5.3.

3.3.5 Discussion

3.3.5.1 Machine orientation

Maclan is not machine-oriented in the usual sense of that phrase
[van der Poel and Maarsen 1974] because it retains strong typing and
an object-oriented viewpoint. But compared to Koala, Maclan's data
structures and powers of binding are machine-oriented to the degree
that its elementary data types and their primitive operations map
directly onto machine structures and operations. The mapping is similar to that of the Pascal data space onto the machine; it is clear that there is a direct Pascal equivalent for nearly all facets of Maclan's data space [table 3.1].

Table 3.1 Data Space Correspondence

<table>
<thead>
<tr>
<th>Maclan</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCT type</td>
<td>RECORD type, no variants</td>
</tr>
<tr>
<td>OVERLAY union type</td>
<td>RECORD type, variant without tag</td>
</tr>
<tr>
<td>VAR declaration</td>
<td>VAR declaration</td>
</tr>
<tr>
<td>initialized ditto</td>
<td>ditto, assignment at head of block</td>
</tr>
<tr>
<td>primitive total-copying assignment, equality</td>
<td>assignment, equality</td>
</tr>
<tr>
<td>primitive total-copy of actual parameter</td>
<td>parameter by value</td>
</tr>
<tr>
<td>sequence, fixed length</td>
<td>one-dimensional array</td>
</tr>
<tr>
<td>REFTO, NREFTO; NIL</td>
<td>pointer type constructor; NIL</td>
</tr>
<tr>
<td>FOLLOW (reference)</td>
<td>pointer value ⬅</td>
</tr>
<tr>
<td>SELF of self-operation</td>
<td>hidden VAR parameter</td>
</tr>
<tr>
<td>function YIELD</td>
<td>function return extended to return structures</td>
</tr>
</tbody>
</table>

Clearly generic types have no equivalent in Pascal. Points where the correspondence is not obvious are the generation of new objects (e.g. "NEW circular (DONTCARE, 0, 1, DONTCARE)") and of reference values ("REFTO x"), and the most general form of sequence ("SEQUENCE max_queue OF ...", where "max_queue" is not a constant). In Pascal new
objects may be generated freely only upon the heap, and such objects are the only ones that may be, and always must be, referred to via references. In Maclan the question of where storage is allocated is not addressed, and references may be generated to any objects whether their Pascal equivalent is on the stack (a variable, for instance) or possibly on the heap (a new object generator). Allocating on the heap those objects that may be referenced (known from inspection of reference type qualifiers) is sufficient to preserve the integrity of the stack against dangling references.

The Maclan sequence type constructor is such that objects of one type may be of different sizes, determined at the time they are generated, and yet need no dope vector in their representation. The sequence maps directly onto a number of adjacent groups of storage units as does the Pascal array; but Pascal arrays are permitted only manifest size, whereas Algol 60 deferred static-sized arrays require a dope vector or equivalent. Three restrictions make it possible for Maclan to dispense with dope vectors: there can be only one degree of freedom in any Maclan type definition; no whole-object operations such as value assignment or equality are permitted; and index values out of bounds need not be detected.

A manifest-sized type has zero degrees of freedom; the elementary types (including reference types) are of manifest size. A structured type has as many degrees of freedom as the sum of its components', plus (in the case of a sequence) those of its bound. A fixed bound of manifest value gives no degree of freedom; any other bound gives a single degree. Thus any form of struct or sequence comprising
components which have manifest size (elementary types, reference
types, structs or manifest-sized sequences of manifest-sized com-
ponents) has one degree of freedom or less and is permitted; for exam-
ple

\[
\text{SEQUENCE max\_hands OF SEQUENCE 5 OF card}
\]

\[
\text{STRUCT (forename, surname: REFTO ((SEQUENCE OF char) SHAREABLE strings);}
\]

\[
\text{offspring: SEQUENCE OF REFTO (person SHAREABLE population)).}
\]

But two components of free size, or an unfixed or non-manifest-sized
sequence of free size, are illegal:

\[
\text{SEQUENCE max\_hands OF SEQUENCE deal\_size OF card (*illegal*)}
\]

\[
\text{STRUCT (forename, surname: SEQUENCE OF char) (*illegal*)}
\]

Any object of a type thus restricted maps simply onto a contigu-
ous section of machine storage so that the address of any component of
the object is a function of constants known at compile time, the base
address of the object, and the values of any index expressions. No
additional information need be stored for the object, provided we can
dispense with knowing its upper bound, which is why Maclan imposes the
above restrictions on operations and bounds errors. A programmer wish-
ing to preserve this information may of course do so explicitly, and
use it in the implementation of his own whole-object operations such
as TOTALCOPY.

The binding of names to objects in Maclan is also restrained,
such that a name corresponds to the machine address of the storage
representing the object. Maclan variables and components of structured
objects are thus direct equivalents of the same constructs in Pascal. The same is true of Maclan non-generic formal parameters and Pascal value parameters. Although Maclan passes parameters by binding, as does Koala, the actual parameter is required to be a newly generated object or copy; such a binding can be rewritten as a copying assignment [see section 4.2.2.2] to a hypothetical formal parameter object, which is the mechanism by which Pascal passes value parameters.

3.3.5.2 The BSL Spectrum

There is no difficulty in seeing Maclan and Koala as parts of a common language. The Broad Spectrum Language has them at its extremes, and we describe the invariants of the language at all points in its spectrum by what the extremes have in common.

The semantics of Koala and Maclan use the data model identically for many constructions having the same syntax: the passing of parameters, the structure and articulation of objects, operations on elementary objects, shareability. In these parts of the language the only differences are that Maclan restricts the constructions’ power and extends them to reference types. In syntactic form and in the strength of correspondence between text and computational objects Koala and Maclan are also similar.

Differences come in scope and visibility of capsule contents, and the introduction of overloadable generic operations. In general Maclan’s rules apply to BSL except at the Koala extreme. For example, the severe visibility restrictions on capsule exports are a feature
only of Koala; at any other point in the language spectrum the rules are those of Maclan. Koala requires an explicit initial binding to always-bound names; elsewhere in the spectrum a binding to a DONTCARE-valued object is implicit.

What, then, is an intermediate point in the language spectrum? It is the union of both extreme language definitions, exemplified by a program containing elements of both, with traversers and possible unbindings alongside references and generic operations. The extreme languages are merely particular restrained subsets of the whole definition.

The syntax for the whole of BSL is given in Appendix 2, where the constructs that do not appear at one extreme or the other are noted.

There is one construct there which is not part of either Koala or Maclan: the "unprotected traverser" type constructor, e.g.:

\[
\text{inspector: UNPROTECTED TRAVERSER OVER cbuff.}
\]

An object of such a type cannot be shared and cannot be assigned to. It is a traverser over the object named or objects of the type named, but is "unprotected" in that it need not be finished when the object changes shape. It is therefore an insecure traverser that may be left dangling. We illustrate mechanisms ensuring its safe use in the following chapters.

As we shall see in Chapter 3, generic types relax strong typing.

An operation added in the intermediate spectrum is "PREVIOUS", applicable to any sequence traverser; it is the inverse of NEXT.

Strict sharing control is relaxed where there is temporary aliasing strongly tied to program constructs; discrimination branch
variables may be bound to share the yield of the discriminated expression, without having a declared shareability.

3.3.5.3 Multiple representations

Bundles provide the textual form to couple several concrete types representing the same abstract type. The selection of a representation is explicit, by naming "bundle $ capsule", and is not driven by parameters as in a Mitchell and Wegbreit "scheme". But the consequences of selecting one representation rather than another for a particular variable appear only in the program's performance, and not in the form of the operations applicable to this variable or between it and others of the same abstract type, whatever their representation.

An implicit requirement for this to be true is that all capsules in a bundle must provide both the operations of the abstract type and those operations needed to implement the bundle's generic operations. The signatures of these operations are not necessarily similar in each capsule, but when they are not, compensatory changes in their calls are needed to achieve the same semantics for each. The program transformation techniques described in the next chapter implement these changes safely and consistently.

As we shall see in Chapter 5, generic types relax strong typing sufficiently to allow the different representation types to be considered as aspects of one abstract type. It is necessary to restrict the operations on generic parameters to preserve security with this relaxation, but what remains is sufficient to allow the inter-
representation copying and comparison operations to be defined. The accompanying notions of immediate export and overloading of these operations make Maclan's scoping more complex than that of Koala, but it follows established paths of languages that allow the explicit declaration of operators [Algol 68, Ada, ELL].
4.1 Introduction

4.2 The major transformations
4.2.1 Specifying applications
4.2.2 Brief catalogue of transformations
4.2.2.1 Major transformations
4.2.2.2 Auxiliary transformations

4.3 Transformations in action
4.3.1 Transforming state spaces
4.3.2 Transforming operations

4.4 Catalogue of transformations
4.4.1 Pushout - binding (push out by reference)
4.4.2 Incorporate - structural
4.4.2.1 Incorporate by markers
4.4.2.2 Incorporate by table
4.4.3 Evert - structural
4.4.4 Nullfornonex - binding/structural
4.4.4.1 Nullfornonex by range (extensionvalue) - structural
4.4.4.2 Nullfornonex by structure - binding
4.4.5 Padout - structural (pad out with fillers)
4.4.5.1 Padout at finish
4.4.5.2 Padout at start and ends
4.4.6 Overlay - structural (union becomes overlay union)
4.4.7 Multicopy - binding (multiple copies)
4.4.8 Indices - structural (indices for traversers)
4.4.9 General comments
4.4.10 Principles of representation

4.5 Formal description and implementation
4.5.1 Introduction to Xlan
4.5.2 Parsing
4.5.3 Decomposition
4.5.4 Determining context
4.5.5 Repetition operations
4.5.6 Pattern matching
4.5.7 Implementation
4.5.8 Experimental work

4.6 Discussion: transformations and the machine end of BSL
4.6.1 A structured description of representation
4.6.2 Comparison with other transformation systems
4.6.2.1 Size of transformations
4.6.2.2 Invocation
4.6.2.3 Notation
4.6.3 The transformations and BSL
4.6.3.1 BSL's effect on the transformations
4.6.3.2 Influence of transformations on the languages' design
4.6.4 Limitations of Xlan
4.1 Introduction

The remainder of this thesis describes a transformation system for BSL by which a programmer can derive a range of Maclan representations for a Koala abstract data type. The system comprises a small library of "major" transformations, whose application is specified by the programmer; and a number of "auxiliary" transformations, applied automatically by the system. This chapter describes these transformations and shows how they allow the programmer to derive representations for isolated capsules. Chapter 5 extends the application of the system to "multiple representations", describing how a bundle of different but assignment-compatible capsules are derived from a common abstract ancestor. Chapter 6 presents an extended example of this multiple representation ability.

All the transformations are "meaning preserving" in the sense of maintaining the input-output function of any program. A series of transformations thus creates a series of input-output equivalent programs; a Maclan program derived from a Koala program by a series of transformations is its equivalent, and therefore an implementation of it.

Formal proof that the transformations preserve meaning is not attempted in this work. However, our confidence in a small number of simple transformations can be high without formal proof; and we can extend this confidence to the systematic use of these transformations in combination.
We shall concentrate on a small number of transformations and the requirements of the transformation system. In particular, no transformations specific to tree structures will be considered.

4.2 The major transformations

4.2.1 Specifying applications

Each major transformation effects some structural change in a capsule's state, or some change in the binding of a name or group of similarly declared names; and it changes operations correspondingly in both cases. The "structural" transformations include one that provides the basis for representing a sequence as a linked list, for example; the "binding" transformations include ones which replace variable bindings by references, or discriminated unions by overlays.

Transformations are specified by name with a list of arguments. Their arguments are program components named from the point of view of the outermost level of a program: a capsule is named by dot-notation combining the names of all enclosing declarative constructs (bundle, capsule, and operation declarations); a bindable construct is named by prefixing its variable or parameter identifier, with "OF", to the name of its enclosing declarative construct. Thus in the following skeleton BSL program the structural transformation called "evert" is applied to the "edition" and "sorting_list" capsules by

\[ evert (book.edition, book.print.sorting_list) \]

while the binding transformation "pushout" is applied by

BEGIN
CAPSULE person = (...);
SHARELIST population OF person;
CAPSULE book = (
    CAPSULE edition = (STATE:
        STRUCT (author: person SHAREABLE population;
                title: string; language: language_name);
    );
    STATE: SEQUENCE UPTO 8 OF edition;
    print = SELF FUNC: integer (* number of items printed *);
    CAPSULE sorting_list =
        (STATE: SEQUENCE UPTO 8 OF edition; ...);
    BEGIN ... END;
);

BEGIN (* main program *) ... END.

For both structural and binding transformations the dotted part of the argument is referred to as the "environment part", and the binding transformation's argument also has a "prefix". We make use of these terms in Chapter 5 in particular.

The argument of a binding transformation may be any bindable construct. Its prefix is an identifier (of a variable, formal parameter, or struct component), or one of the reserved words:

YIELD - the yield of a function, as in

"pushout (YIELD OF book.print)"; or

FOLLOW - the qualification of a reference type, as in

"pushout (FOLLOW OF book.edition.author)"; or

ALL - the component specification of a sequence or tree, as in

"pushout (ALL OF book)".
Brief catalogue of transformations

We list the library of transformations very briefly here. A detailed description follows later [section 4.4].

1. **Pushout (binding)**

   replace a binding to an object by a binding to a reference to it.

2. **Incorporate (by markers, by table - both structural)**

   incorporate the components of the components of an object as direct components of that object.

3. **Evert (structural)**

   substitute a chain of objects with explicit successor components for the nodes and successor relations of a sequence.

4. **Nullfornonex (by range - structural; by structure - binding)**

   substitute an extended range of values in an object’s type for a possibly nonexistent binding.

5. **Padout (at first, at last, at both ends - all structural)**

   pad out a variable but bounded sequence with logically unused "filler" components to a fixed size.

6. **Overlay (binding)**

   substitute an undiscriminated overlay union plus tag field for a discriminated union.
7. Multicopy (binding)

emulate shared bindings of one object by single bindings to mul-
tiple copies of it.

8. Indices (structural)

replace traversers by indexing variables associated with the sequence.

4.2.2.2 Auxiliary transformations

1. Binding assignment into copy assignment

\[
\begin{align*}
x & \leftarrow \text{TOPCOPY}(y) \Rightarrow x := \text{TOPCOPY}(y) \\
x & \leftarrow \text{TOTALCOPY}(y) \Rightarrow x := \text{TOTALCOPY}(y) \\
x & \leftarrow 6 \Rightarrow x := 6 \\
x & \leftarrow y + z \Rightarrow x := y + z
\end{align*}
\]

where x is always bound and not shareable.

2. Top-copying into total-copying

\[
\begin{align*}
x & \leftarrow y \Rightarrow x := y \\
x & \leftarrow y + z \Rightarrow x := y + z
\end{align*}
\]

where x has no shared descendent components.

3. Redundant copying

\[
\begin{align*}
x & := \text{TOPCOPY}(y) \Rightarrow x := y \\
x & := \text{TOTALCOPY}(y) \Rightarrow x := y
\end{align*}
\]

and other similar cases where copying operations are mixed with copying assignments.

4. Conditional flattening

\[
\text{IF } x = c \text{ THEN } S(c) \text{ ELSE } S(x) \text{ ENDIF} \Rightarrow S(x)
\]
where \( x \) and \( c \) are side-effect-free expressions and \( S(c) \) and \( S(x) \) are sequences of statements identical apart from the substitution of \( c \) in \( S(c) \) for all occurrences of \( x \) in \( S(x) \); all constituent bindable constructs of the expressions \( x \) and \( c \) are unshareable, and \( S(x) \) contains no assignments to any constituent bindable constructs of \( x \), nor any operation calls that might lead to such assignments.

5. Redundant de-referencing

\[
\text{REFTO FOLLOW}(x) \Rightarrow \text{TOTALCOPY}(x) \\
\text{FOLLOW}(x) \Leftrightarrow \text{FOLLOW}(y) \Rightarrow x = y
\]

i.e. generating a reference to an object obtained by following a reference expression is equivalent to generating a copy of the expression; the "same-as" comparison of followed references is equivalent to equality comparison of the references themselves.

6. Yield propagation

\[
\text{factorial YIELD YIELDOF} \Rightarrow \begin{cases} 
\text{IF } n = 0 \text{ THEN YIELD 1} \\
\text{ELSE YIELD } n \ast (n - 1)
\end{cases} \\
\text{THEN factorial YIELD 1} \\
\text{ELSE factorial YIELD } n \ast (n - 1)
\]

i.e. nested yield-expressions, and yield-expressions in functions' YIELD assignments, may be collapsed into a single level of expression.
4.3 Transformations in action

4.3.1 Transforming state spaces

We can follow these transformations through a simple example. Consider a catalogue of books. Each book may be published in several different editions in different languages, and the catalogue is a list of books, each book being characterized as a list of its different editions. The abstraction "catalogue" is defined in terms of a SEQUENCE of "book"; "book", a SEQUENCE of "edition"; "edition", a STRUCT containing title, author and language.

(catalogue) SEQUENCE OF book TRAVERSED c_printer;

(book) SEQUENCE UPTO 8 OF EDITION TRAVERSED thro_langs, best_copy;

(edition) STRUCT (title: text;
  author: name SHAREABLE all_authors;
  language: (English, Russian, French, Spanish) ).
A particular catalogue object is pictured as

\[
\begin{array}{c}
\text{catalogue} \\
\downarrow \\
\text{cprinter} \\
\downarrow \\
\text{book} \\
\downarrow \\
\text{thro langs} \\
\downarrow \\
\text{best copy} \\
\downarrow \\
\text{edition} \\
\downarrow \\
\text{title} \\
\text{author} \\
\text{language} \\
\text{author} \\
\text{name} \\
\end{array}
\]

One method of representing a "catalogue" object is as a reference-linked list of "edition" objects, each "book" group of these separated by an object of distinctive type. The realizations of a traverser over the sequence of "books", and of traversers over the "editions" of each "book", are as references to elements of the list of "editions". The conventional picture is

\[
\begin{array}{c}
\text{M} \\
\downarrow \\
\text{E} \\
\downarrow \\
\text{E} \\
\end{array}
\]

The head of the catalogue contains a reference to a linked list of
items and an "edge reference" (i.e., a pair of references [Wise 1976]) representing the catalogue traverser "cprinter". The list links are variants of a struct type, tagged as either "marker" or "edition", with the format

marker item

| M | thro langs | best_copy | next |

dition item

| E | title | author | language | next |

The hatched area represents logically unused storage locations.

The derivation of this representation as a series of elementary transformations is:

1. Incorporate by markers (catalogue)

The catalogue sequence becomes one of a mixture of editions and marker objects, the markers delimiting the logical boundaries of books and incidentally carrying their traversers.

(catalogue) SEQUENCE OF UNION (member: edition; mark: marker) TRAVERSED cprinter;


2. Evert (catalogue)

The state of "catalogue" becomes a list of elements, each comprising an "edition" or "marker" and the succeeding such element. Because the sequence may be empty, the state of a catalogue object may be an empty list. The traverser "cprinter" becomes a field of the "catalogue" struct which is bound to an edge-reference into the list. The list elements are as yet still linked by componency relations, not references. The state of the...
"catalogue" is now

```
STRUCT (seq: POSSIBLY element SHAREABLE list;
cprinter: edgelem SHAREABLE list),
```

where "element" is defined as

```
STRUCT (elem: UNION (member: edition; mark: marker);
succ: POSSIBLY element SHAREABLE list);
```

and "edgelem" as

```
STRUCT (POSSIBLY curr: element;
    POSSIBLY pred: element).
```

The last element of the list has no successor, indicated by its "succ" component being unbound.

3. Overlay (elem OF element)

Instances of "marker" are differentiated from those of "edition" by the value in a component of the replacement struct. The type "element" is now defined as:

```
STRUCT (elem: STRUCT (kind: (mark,member);
    thing: OVERLAY (marker, edition));
succ: POSSIBLY element SHAREABLE list).
```

4. Pushout (seq OF catalogue,
    succ OF catalogue.element,
    curr OF catalogue.edgelem,
    pred OF catalogue.edgelem)

The state of "catalogue" becomes

```
STRUCT (seq: POSSIBLY REFTO (element SHAREABLE list);
cprinter: edgelem SHAREABLE list);
```

and of "element"

```
STRUCT (elem: STRUCT (kind: (mark, member);
    thing: OVERLAY (marker, edition));
succ: POSSIBLY REFTO (element SHAREABLE list)).
```
5. Nullfornonex by range (NIL) (ditto)

The "POSSIBLY REFTO" formaltypes become "NREFTO", and the end of
the list is now marked by a NIL value. "Element" becomes

\[
\text{STRUCT (elem: STRUCT (kind: (mark, member);
thing: OVERLAY (marker, edition));
succ: NREFTO (element SHAREABLE list)).}
\]

The representation has not yet been fully specified, because the
definition of "edition" still contains a shareable component (author).
We have a choice of two ways of removing this direct shareability:
one is to push out "author", leaving only an indirect sharing through
references; the other is to "multicopy" author, replacing any sharing
by copying. The multicopy transformation requires that there be no
selective updating on objects in its argument's sharegroup, and this
fits the modelling of a book's author.

6. Multicopy (author OF catalogue.edition)

The change to the state of "edition" is very small:

\[
\text{STRUCT (title: text;
author: name;
language: (English, Russian, French, Spanish)).}
\]

This series of six transformations has achieved the representa-
tion pictured above. Alternative representations that can be obtained
include:

- a linked list of "books" (evert (catalogue); pushout (succ OF
catalogue.element));
- a continuous vector (padout (catalogue)) of "books" of fixed and equal size (padout at end (book));

- a vector of references to reference-linked lists of editions (evert and pushout each book; pushout the resulting "book"; padout the sequence of references to fixed size);

- a vector of references to vectors of eight references to some representation of "edition" objects.

4.3.2 Transforming operations

The more complex part of data structure transformation is in transforming the operations that apply to the affected structures. Some of the flavour of our transformation system can be seen in a short example.

CAPSULE holder =
  (STATE: SEQUENCE UPTO max OF person SHAREABLE population
   TRAVERSED inspector;
   insert = SELF PROC (p: person SHAREABLE population);
   BEGIN GROW AT FIRST WITH p END;
  replace = SELF PROC (p: person SHAREABLE population);
  VAR done: boolean <- false;
  BEGIN RESET inspector;
  WHILE NOT FINISHED inspector AND NOT done DO
   IF p <-> (# inspector #) THEN done := true;
   ELSE IF p = (# inspector #) THEN done := true; (# inspector #) <- p
   ELSE NEXT inspector ENDIF ENDIF
  ENDDO
  END
  VAR x: holder <- NEW holder (mabel, arthur, george).
The "replace" operation searches a "holder" for a person which is either the same as the parameter "p" or total-copy equal to it. In the latter case, which could amount to "finding someone with the same description" in the world being modelled, it replaces that person by the parameter.

The structural transformation "pushout (ALL OF holder)" replaces the sequence's component definition with a reference type and modifies all operations on components appropriately. (Affected lines are marked with "]".)

CAPSULE holder =
  (STATE: SEQUENCE UPTO max OF REFTO (person SHAREABLE population)
   TRAVERSED inspector;

  insert = SELF PROC (p: person SHAREABLE population);
  BEGIN GROW AT FIRST WITH REFTO p END;

  replace = SELF PROC (p: person SHAREABLE population);
  VAR done: boolean := false;
  BEGIN RESET inspector;
  WHILE NOT FINISHED inspector AND NOT done DO
      IF p <-> FOLLOW ((# inspector #))
          THEN done := true
      ELSE IF p = FOLLOW ((# inspector #))
          THEN done := true; (# inspector #) := REFTO p;
          ELSE NEXT inspector
      ENDIF
  ENDDO
  END)

VAR x: holder <- NEW holder (REFTO mabel,
   REFTO arthur, REFTO george).

The pushout transformation distinguishes left-hand binding uses of its argument from all others. Right-hand value uses of components of the sequence, whether used with binding or copying operators, are transformed to FOLLOW the reference value that they have become.
Left-hand uses in a binding operation (in which we include growth of the sequence, as in "insert", as well as binding assignment to an indexation or detraversal, as in "replace") transform the corresponding right-hand side of the binding to create a reference value. This includes the actual parameters of the object generator NEW, which are the right-hand sides of bindings to the sequence components.

The binding assignment with left-hand use of a sequence component remains a binding assignment under the pushout transformation:

\[
\text{(¬ inspector #) ← p} \Rightarrow \text{(¬ inspector #) ← REFTO p}
\]

and an auxiliary transformation takes a binding assignment between a generated value (the reference) and an always-bound, unshared name (the reference variable component of the sequence) and produces a copying assignment:

\[
\Rightarrow (\text{¬ inspector #}) := \text{REFTO p}.
\]

The same auxiliary transformation has modified the initialization of "done".

A further step towards obtaining a Maclan version of the program is to remove the remaining shared bindings. We substitute references for the operation parameters:

\[
\text{pushout (p OF holder.insert, p OF holder.replace)}
\]

which results in the procedure headers

\[
\text{insert = SELF PROC (p: REFTO (person SHAREABLE population))};
\]

\[
\text{replace = SELF PROC (p: REFTO (person SHAREABLE population))}.
\]

The capsule's state is unaffected, naturally, but the operations
involving these formal parameters, and the form of the corresponding actual parameters in their calls (not shown), are changed. The operations are transformed in exactly the same way as those on the sequence components above, but in this instance more auxiliary transformations are made possible thereby.

The body of "insert" has a right-hand valued use of "p", in "REFTO p". This will always be replaced by a FOLLOW operation

\[
\text{REFTO } p \Rightarrow \text{REFTO FOLLOW}(p)
\]

which the redundant de-referencing auxiliary takes to "TOTALCOPY(p)".

Likewise in "replace":

\[
(# \text{inspector } #) := \text{REFTO } p \Rightarrow (# \text{inspector } #) := \text{REFTO FOLLOW}(p) \\
\Rightarrow (# \text{inspector } #) := \text{TOTALCOPY}(p) \\
\Rightarrow (# \text{inspector } #) := p;
\]

the identity comparison is a right-hand use, and the redundant de-referencing auxiliary applies to the result:

\[
p \leftrightarrow \text{FOLLOW}((# \text{inspector } #)) \\
\Rightarrow \text{FOLLOW}(p) \leftrightarrow \text{FOLLOW}((# \text{inspector } #)) \\
\Rightarrow p = ( # \text{inspector } #)
\]

The pushout transformation generally substitutes one simple expression for another; by contrast, a structural transformation such as padout achieves a mapping of one structure onto another by substantially modifying its operations. We start with a similar capsule to the one above, with a simple "extract" operation instead of "replace"; insert and extract apply first-in, first-out rules.
CAPSULE simple_holder =
(STATE: SEQUENCE UPTO max OF person SHAREABLE population;
insert = SELF PROC (p: person SHAREABLE population);
BEGIN GROW AT FIRST WITH p END;
extract = SELF FUNC: person SHAREABLE population;
BEGIN extract YIELD (# FIRST #);
SHRINK AT FIRST END
)
VAR x: simple_holder <- NEW simple_holder (mabel, arthur, george).

Applying "padout at start (simple_holder)" maps the variable-length sequence onto one of fixed length, where growth and shrinkage at its first component become movements of an indicator (a traverser) within the fixed sequence.

CAPSULE simple_holder =
| (STATE: SEQUENCE max OF person SHAREABLE population
| TRAVERSED limit (LAST);
| insert = SELF PROC (p: person SHAREABLE population);
| BEGIN IF FINISHED limit
| THEN error ("sequence bound exceeded")
| ELSE (# limit #) <- p;
| PREVIOUS limit
|ENDIF END;

extract = SELF FUNC: person SHAREABLE population;
| BEGIN IF FINISHED limit
| THEN extract YIELD (# FIRST #)
| ELSE extract YIELD (# limit BY NEXT #)
| ENDIF;
| NEXT limit
END
)
VAR x: simple_holder <-
| NEW simple_holder (DONTCARE, DONTCARE, mabel, arthur, george).

Explicit error detection has appeared in "insert". In general, explicit detection is introduced wherever the operation resulting from the transformation would not detect an error in the original, or (as
in this case) would detect a different error. Without explicit detection the error here would refer to detraversing a finished traverser ("limit") rather than the logical error that the original sequence's bounds have been exceeded. "Limit" always indicates one component position below the logical first of the original sequence and is therefore initialized to LAST. When this traverser is finished the padded sequence is full.

In transforming the object generator for the declared variable the value of "max" has been taken to be 5, requiring two DONTCARE objects to pad out the sequence to its full length initially.

We distinguish between "class context" and "object context" object expressions in the program being transformed. An object expression always yields a name, as described above. A name context is the nearest equivalent in SAD to a left-hand value, and applies an operation to the name itself (generally so that its binding may be modified). An object context (right-hand value) is concerned only with an object or its value, and uses the name only to follow its binding in this end. Thus in "a = b", "a" is in a name context, "b" in an object context; but in "a := b", "a = b" and "a := b" both "a" and "b" are in object context. (Note that the copying assignments modify the contents of left-hand side objects, not the binding of their names.) An actual parameter always in object context to its formal parameter's name context, in both operation calls and object generators.
4.4 Catalogue of transformations

We now give informal descriptions of the major transformations, in varying amounts of detail. Most detail is given for samples of a binding transformation (pushout) and of a structural one (evert); these two are also used as examples of the formal description and implementation method described below [section 4.5]. Two terms found useful in contrasting a portion of program before and after a particular transformation is applied are the adjectives "pretrans" and "post-trans".

In describing binding transformations ("pushout" in particular) we distinguish between "name context" and "object context" object expressions in the program being transformed. An object expression always yields a name, as described above. A name context is the nearest equivalent in BSL to a left-hand value, and applies an operation to the name itself (generally so that its binding may be modified). An object context (right-hand value) is concerned only with an object or its value, and uses the name only to follow its binding to this end. Thus in "a <- b", "a" is in a name context, "b" an object context; but in "a := b", "a :- b" and "a <-> b", both "a" and "b" are in object context. (Recall that the copying assignments modify the contents of left-hand side objects, not the binding of their names.) An actual parameter stands in object context to its formal parameter's name context, in both operation calls and object generators.
4.4.1 Pushout - binding (push out by reference)

RESTRICTIONS:

none.

SUMMARY:

replaces a binding to an object by a binding to a reference object referring to it.

DECLARATIONS:

a: car
b: car SHAREABLE autos
POSSIBLY c: car

OPERATIONS:

Object contexts

\[ x := a; \quad \Rightarrow \quad x := \text{FOLLOW}(a); \]
\[ \text{number OF } b \quad \Rightarrow \quad \text{number OF } \text{FOLLOW}(b) \]
\[ \text{print()} \text{ OF } a; \quad \Rightarrow \quad \text{print()} \text{ OF } \text{FOLLOW}(a); \]
\[ y < - b; \quad \Rightarrow \quad y < - \text{FOLLOW}(b); \]
\[ z < - c; \quad (\text{where "z" is possibly unbound}) \quad \Rightarrow \quad z < - \text{YIELDOF (WITH c DISCRIM WHEN exc: TOTALCOPY EXISTS THEN z < - FOLLOW(exc) ENDWHEN OTHERWISE z < - NONEX ENDDISCRIM)}; \]
\[ x < - c; \quad (\text{where "x" is always bound}) \quad \Rightarrow \quad x < - \text{YIELDOF (WITH c DISCRIM WHEN exc: TOTALCOPY EXISTS THEN x < - FOLLOW(exc) ENDWHEN OTHERWISE error ("nonexistent binding") ENDDISCRIM)}; \]
a := x;  
\Rightarrow \text{FOLLOW}(a) := x;

Name contexts

b <- y;  
\Rightarrow b <- \text{REFTO} y;

c <- x;  
(\text{where } "x" \text{ is always bound})
\Rightarrow c <- \text{REFTO} x;

c <- z;  
(\text{where } "z" \text{ is possibly unbound})
\Rightarrow \text{WITH} z \text{ DISCRIM}
\Rightarrow \text{WHEN} \text{exz: EXISTS THEN}
\Rightarrow c <- \text{REFTO exz}
\Rightarrow \text{ENDWHEN}
\Rightarrow \text{OTHERWISE} c <- \text{NONEX}
\Rightarrow \text{ENDDISCRIM};

COMMENTS:

- Several auxiliary transformations are triggered by the reference expressions introduced by pushout. For example, in "x <- y", if both "x" and "y" are pushed out the result is a total-copy assignment, in whichever order the two are transformed. In one case we have

\[
x <- y \Rightarrow x <- \text{REFTO} y \Rightarrow x := \text{REFTO} y
\]
\text{pushout}(x)  \text{auxiliary 1}
\Rightarrow x := \text{REFTO} y
\text{auxiliary 2}
\Rightarrow x := \text{REFTO} \text{FOLLOW}(y) \Rightarrow x := \text{TOTALCOPY}(y)
\text{pushout}(y)  \text{auxiliary 5}
\Rightarrow x := y
\text{auxiliary 3}

while in the opposite order
x <- y => x <- FOLLOW(y)
pushout(y)

=> x <- REPTO FOLLOW(y) => x <- TOTALCOPY(y)
pushout(x)
auxiliary 5

=> x := TOTALCOPY(y)
auxiliary 1

=> x := TOTALCOPY(y)
auxiliary 2

=> x := y
auxiliary 3

4.4. Incorporate components (interpret components' components)

Restrictions:
- components of argument type must not be a sequence type
- components of argument type must not be of sequence type
- TOTALCOPY and TOTALCOPY must not be a sequence type
- TOTALCOPY and TOTALCOPY must not be an auxiliary

4.4.2. Incorporate by markers

Summary:
flattens the two-level sequence of sequences of all type "x" into a single level sequence of a union type. The union is of type "x" and a marker type; a marker precedes each group of "x" components that is derived from a common prntmap sequence, and itself bis components representing the prntmap sequence's traversals by unprotected traversers over the top-level sequence.

Declarations:


CAPSULE book = (STATE: SEQUENCE INTO n OF edition
TRAVERSED thru langs...
all print = SELF PROC (* of book *)
BEGIN ... END;
4.4.2 Incorporate - structural (incorporate components' components)

RESTRICTIONS:

- argument must be a sequence type with components of sequence type;
- components of argument type must not be shareable;
- TOTALCOPY and TOPCOPY must not be applied to pretrans components.

4.4.2.1 Incorporate by markers

SUMMARY:

flattens the two-level sequence of sequences of any type "x" into a single level sequence of a union type. The union is of type "x" and a marker type; a marker precedes each group of "x" components that is derived from a common pretrans sequence, and itself has components representing the pretrans sequence's traversers by unprotected traversers over the top-level sequence.

DECLARATIONS:

capsule catalogue = (state: sequence of book traversed cpr;

 capsule book = (state: sequence upto 3 of edition traversed thro langs;
    all print = self proc; (* of book *)
    begin ...
    end;
 )

CAPSULE catalogue = (STATE: SEQUENCE OF
 UNION (member: edition;
 mark: marker)
 TRAVERSED cpr;

CAPSULE marker = (STATE: STRUCT (
thro_langs: UNPROTECTED TRAVERSER OVER catalogue));

all print = PROC (b: UNPROTECTED TRAVERSER OVER catalogue);
 BEGIN ... END;
)

OPERATIONS:

Generators

NEW catalogue (NEW book (el, e2), NEW book (e3, e4, e5))

=> NEW catalogue (mark: DONTCARE, member: el, member: e2,
 mark: DONTCARE, member: e3, member: e4, member: e5)

Traversers

RESET cpr;

=> RESET cpr;

NEXT cpr;

=> BEGIN VAR done: boolean := false;
 NEXT cpr
 WHILE NOT done AND NOT FINISHED cpr
 DO WITH(# cpr #) DISCRIM
 WHEN mark: done := true
 ENDWHEN
 ENDDISCRIM ENDDO END;

RESET thro_langs OF (# cpr #)

=> MAKECOINCIDE
thro_langs OF (# cpr #)
 WITH cpr BY NEXT

FINISHED thro_langs OF (# cpr #)

=> (FINISHED thro_langs OF (# cpr #)
 COR YIELDOF (WITH (# thro_langs OF (# cpr #) #) DISCRIM
 WHEN mark THEN YIELD true ENDDISCRIM
 WHEN member THEN YIELD false ENDDISCRIM))
COMMENTS:

- This transformation corresponds to dynamically linearizing a two-dimensional array.
- The restrictions on TOTALCOPY, TOPCOPY and sharing of the argument's components (the "books" here) are due to the transformation's eliminating distinct objects of that type.

4.4.2.2 Incorporate by table

SUMMARY:
flattens the two-level sequence of sequences of any type "x" into a struct of two components: (i) a sequence of type "x", which preserves successor relations of components of a common second level sequence, and the order of those sequences themselves; and (ii) a sequence of structs (one for each second level sequence object) containing unprotected traversers into the other component to indicate boundaries of the second level sequences and their traversers.

DECLARATIONS:

CAPSULE catalogue = (STATE: SEQUENCE OF book TRAVERSED cpr;
CAPSULE book = (STATE: SEQUENCE UPTO 8 OF edition TRAVERSED thro langs;
all print = SELF PROC; (* of book *)
BEGIN ... END;
)
)
\[
\text{\# 4.3  

\text{\textbf{Clustering }}}
\]

- 160 -

\[
\rightarrow \text{\textsc{Capsule} catalogue} = \text{(state:} \\
\text{\textsc{struct} cmpts: \text{sequence of} edition;} \\
\text{\text{table: \text{sequence of} book mark} } \\
\text{\textsc{traversed} cpr)}; \\
\]

\[
\text{\textsc{capsule} book mark} = \text{(state:} \\
\text{\textsc{struct} firstb, lastb, thro langs:} \\
\text{\text{unprotected \text{traverser} over catalogue))};} \\
\]

\[
\text{all print} = \text{proc} (b: \text{book mark};} \\
\text{\textit{begin} ... \textit{end};} \\
\]

\textbf{operations:}

\textbf{traversers}

\textbf{reset thro langs} \rightarrow \textbf{makecoincide} \\
\text{of self} \texttt{(* a book *')} \\
\text{thro langs of b} \\
\text{with firstb of b}

\textbf{finished thro langs} \rightarrow \textbf{finished thro langs of} (\# cpr #) \\
of (\# cpr #)

\textbf{next thro langs of} (\# cpr #)

\rightarrow \text{if thro langs of} (\# cpr #) coincides lastb of (\# cpr #) \\
then \textbf{makefinished thro langs of} (\# cpr #) \\
else \textbf{next thro langs of} (\# cpr #)

\textbf{comments:}

- This transformation reduces a variable number of variable-length 
sequences to two variable-length sequences.
4.4.3 Evert — structural

RESTRICIONS:
- argument must be a sequence.

SUMMARY:
the sequence is broken into structs which contain recursive-typed
components representing the sequence's successor relation (and
also its predecessor relation, if any PREVIOUS operation is
applied to any instance of the argument).

DECLARATION:

CAPSULE catalogue = (STATE: SEQUENCE OF book TRAVERSED cpr; ...)  
(where no PREVIOUS operation is applied)

=> CAPSULE catalogue = (STATE:
  STRUCT (POSSIBLY seq: element SHAREABLE eltgroup;
    cpr: edgelem);
  SHAREGROUP eltgroup OF element;
  CAPSULE element = (STATE:
    STRUCT (elem: book;
      POSSIBLY succ: element SHAREABLE eltgroup));

CAPSULE edgelem =
  (STATE: STRUCT (POSSIBLY curr: element SHAREABLE eltgroup;
                    POSSIBLY pred: element SHAREABLE eltgroup));

index = SELF FUNC (indexpr: integer): edgelem;
VAR counted: integer <- 0;
VAR indextrav: edgelem;
BEGIN
  curr OF indextrav <- seq;
  succ OF indextrav <- NONEX;
  (* equivalent to RESET indextrav *)
IF indexpr < 1
THEN error ("illegal negative index")
ELSE
WHILE counted < (indexpr - 1)
AND EXISTS curr OF indextrav
(* transform of "NOT FINISHED indextrav" *)
DO counted := counted + 1;
(* NEXT indextrav *)
pred OF indextrav <- curr OF indextrav;
curr OF indextrav <- succ OF curr OF indextrav
ENDDO;
IF NOT EXISTS curr OF indextrav
THEN error ("illegal index - too large")
ELSE index YIELD TOPCOPY (indextrav)
ENDIF
ENDIF
END.

MAKEFINISHED cpr => BEGIN pred OF cpr <- DONTCARE;
curr OF cpr <- NONEX
END
NEXT cpr
=> WITH curr OF cpr DISCRIM
WHEN excurr: EXISTS SHAREABLE eltgroup
THEN pred OF cpr <- excurr;
curr OF cpr <- succ OF excurr
ENDWHEN
OTHERWISE error ("advancing finished traverser")
ENDDISCRIM.
RESET cpr => BEGIN pred OF cpr <- NONEX;
curr OF cpr <- seq
END.

all_print() OF (# cpr #)
=> all_print() OF elem OF curr OF cpr.
GROW BEFORE cpr WITH some_book

=> WITH curr OF cpr DISCRIM
    WHEN excurr: EXISTS SHAREABLE eltgroup THEN
        BEGIN VAR newel: element SHAREABLE eltgroup;
            elem OF newel <- some_book;
            succ OF newel <- excurr;
            WITH pred OF cpr DISCRIM
                WHEN expred: EXISTS SHAREABLE eltgroup
                    THEN succ OF expred <- newel
                    ENDWHEN
            OTHERWISE seq <- newel
        ENDDISCRIM;
        pred OF cpr <- newel
    ENDWHEN
    OTHERWISE error ("grow before finished traverser")
    ENDDISCRIM.

Indexing

    all_print() OF [16] OF some_catalogue

=> all_print() OF elem OF curr OF index(16) OF some_catalogue.

COMMENTS:

- The posttrans operations have the familiar forms of linked list manipulation, adapted to the absence of references and the use of nonexistent bindings in place of nil references.
- All indexing operations use the closed-form "index" function.
- Edge-references representing traversers and yielded by "index" include a possible predecessor as well as the expected current element, because the GROW BEFORE operation requires it to maintain the linked list correctly.
4.4.4 Nullfornonex – binding/structural (null value for nonexistent binding)

4.4.4.1 Nullfornonex by range (extension value) – structural

RESTRICTIONS:
- argument must be a scalar type with an extensible range of values: an enumeration, subrange, or REFTO type;
- extension value must be potentially compatible with argument type but not already a member of its set of values. NIL is the only value allowed for reference types.

SUMMARY:
extends the argument type with the given value, which is used as a value of objects always bound to previously possibly unbound names of the type.

DECLARATIONS:

STATE: 1 .. 25; => STATE: 1 .. 26;
(where 26 is the extension value)

STATE: (red,blue,black); => STATE: (red, blue, black, colourless);
(where "colourless" is the value)

STATE: REFTO (amode); => STATE: NREFTO (amode);
OPERATIONS:

Discrimination
- where discriminated expression side-effect free

WITH hue DISCRIM
  WHEN has_hue: TOTALCOPY EXISTS THEN
    colour$ print (has_hue)
  ENDWHEN
  OTHERWISE colour$ print (black)
ENDDISCRIM

=> IF hue <> colourless
  THEN colour$ print (TOTALCOPY (hue))
  ELSE colour$ print (black)
ENDIF

- where discriminated expression may have side-effects

WITH colour$ read() DISCRIM
  WHEN has_hue: TOTALCOPY EXISTS THEN
    in_colour := has_hue;
    colour$ print (has_hue)
  ENDWHEN
  OTHERWISE string$ print ("unknown colour");
    in_colour := black
ENDDISCRIM

=> BEGIN VAR has_hue: colour := colour$ read();
   IF has_hue <> colourless
     THEN in_colour := has_hue;
     colour$ print (has_hue)
   ELSE string$ print ("unknown colour");
     in_colour := black
   ENDIF END

Object contexts
- where nonexistence unacceptable

VAR x: colour := hue;  =>  VAR x: colour := YIELDOF (  
   IF hue <> colourless
     THEN YIELD hue
   ELSE error ("nonexistent object")
ENDIF)
where nonexistence acceptable

(This situation arises only where there is type-splitting [see Chapter 5] in which the type of "hue" is transformed while that of "y" is not.)

VAR POSSIBLY y: colour  =>  VAR POSSIBLY y: colour $ colourl
  <- TOTALCOPY(hue);  <- YIELDOF (IF hue <> colourless
                        THEN YIELD hue
                        ELSE YIELD NONEX
                        ENDF)

hue <- TOTALCOPY(y);  =>  hue <- YIELDOF (WITH y DISCRIM
                                      WHEN exy: TOTALCOPY EXISTS
                                        THEN YIELD TOTALCOPY(exy)
                                        ENDDISCRIM
                                      OTHERWISE YIELD colourless
                                      ENDDISCRIM);

COMMENTS:

- Despite first appearances this is a structural rather than a binding transformation, because the variables' original type is lost altogether in the transformation, and not simply displaced as in pushout or nullfornonex by structure, for example.

- In split-type cases the posttrans program depends upon the compatibility of enumeration types one of which has a subrange of the other's values, such as "(a, b, c)" and "(a, b, c, null)."
4.4.4.2 Nullfornonex by structure - binding

RESTRICTIONS:

none.

SUMMARY:

replaces a possibly unbound declaration by one always bound to a structured type consisting of a logical existence tag and an always bound component of the pretrans type.

DECLARATIONS:

VAR POSSIBLY a: amode SHAREABLE aa

=> VAR a: STRUCT (present: boolean; item: amode SHAREABLE aa)

OPERATIONS:

Discrimination

- where discriminated expression side-effect free

WITH a DISCRIM
  WHEN exa: EXISTS SHAREABLE aa
  THEN amode $ print (exa)
  ENDDISC
  OTHERWISE
    string $ print ("none")
ENDDISCR

=> IF present OF a
  THEN
    amode $ print (item OF a)
  ELSE
    string $ print ("none")
ENDIF
Object contexts

\[ y := a; \]

\[ \Rightarrow \quad y := \text{YIELDOF} ( \]
\[ \quad \text{IF present OF } a \]
\[ \quad \text{THEN YIELD item OF } a \]
\[ \quad \text{ELSE error} \]
\[ \quad ("nonexistent object") \]
\[ \quad \text{ENDIF} \]

Name contexts

\[ a <- y; \]
\[ (\text{where } y \text{ is always bound}) \]

\[ \Rightarrow \quad \text{present OF } a := \text{true; } \]
\[ \text{item OF } a <- y; \]

\[ a <- x; \]
\[ (\text{where } x \text{ is possibly unbound}) \]

\[ \Rightarrow \quad \text{WITH x DISCRIM} \]
\[ \quad \text{WHEN exx: EXISTS SHAREABLE aa} \]
\[ \quad \text{THEN present OF } a := \text{true; } \]
\[ \quad \text{item OF } a <- x \]
\[ \quad \text{ENDWHEN} \]
\[ \quad \text{OTHERWISE present OF } a := \text{false} \]
\[ \quad \text{ENDDISCRIM} \]
4.4.5 Padout - structural (pad out with fillers)

RESTRICTIONS:
- argument must be a bounded sequence.

SUMMARY:
replaces a bounded variable-sized sequence by a sequence of the same bound but of fixed size, with traversers added to indicate one or both bounds of the logically "in use" subsequence.

4.4.5.1 Padout at finish

DECLARATIONS:

CAPSULE book = (STATE: SEQUENCE UPTO 8 OF edition TRAVERSED thro_langs; ...) => CAPSULE book = (STATE: SEQUENCE 8 OF edition TRAVERSED thro_langs, limit (FIRST); index = SELF FUNC (i: integer): UNPROTECTED TRAVERSER OVER book; BEGIN ... END; ...).

OPERATIONS:

Traversing

RESET thro_langs => RESET thro_langs

NEXT thro_langs => BEGIN NEXT thro_langs;
IF thro_langs COINCIDES limit THEN MAKEFINISHED thro_langs ENDIF END

(# thro_langs #) => (# thro_langs #)
- 170 -

Change of size

```
GROW AT thro_langs WITH TOPCOPY (some_edition)

=> IF Finished thro_langs
THEN error ("grow at finished traverser")
ELSE
IF NOT Finished best_copy
THEN error ("growth when other traverser unfinished")
ENDIF;
NEXT limit;
IF Finished limit
THEN error ("sequence bound exceeded")
ENDIF;
BEGIN
VAR growaux: UNPROTECTED TRAVERSER OVER SELF;
MAKECOINCIDE growaux WITH limit;
WHILE NOT growaux COINCIDES thro_langs DO
  (# growaux #) <- TOPCOPY ((# growaux BY PREVIOUS #));
  PREVIOUS growaux
ENDDO;
(# growaux #) <- TOPCOPY (some_edn)
END;
NEXT thro_langs
ENDIF.
```

COMMENTS:
- Size changes only advance (NEXT or PREVIOUS) the limit traverser rather than changing the size of the sequence. The check for any pretrans traverser being finished in these operations is necessary to maintain error behaviour.

4.4.5.2 Padout at start and ends

Padout at start is similar to padout at finish. Padout at ends is similar to one-ended padout, but with "upper" and "lower" limit traversers straddling the logically used subsequence which is initially centred.
4.4.6 Overlay - structural (union becomes overlay union)

RESTRICTIONS:
- argument type must be a discriminated union;
- constituents of the union must have the same sharegroup as other shareable constituents of that type, or be unshareable.

SUMMARY:
replaces a discriminated union by a structure comprising a tagged union, i.e. an enumeration component and an (undiscriminated) overlay union component.

DECLARATIONS:

CAPSULE label = (STATE:
UNION (ident: string SHAREABLE string_lits,
number: 1000 .. 9999,
local: string))

BEGIN
kind OF alab := ident;
content OF alab <- current
END

alab <- TOTALCOPY(blab)
Discrimination

WITH alab DISCRIM
  WHEN declared: ident SHAREABLE string_lits THEN
    declared := string$ read()
  ENDWHEN
  WHEN n: number TOTALCOPY THEN
    next_number := n + 1
  ENDWHEN
  OTHERWISE
  END DISCRIM

=> CASE kind OF alab IN
  WHEN ident THEN BEGIN
    VAR declared: string SHAREABLE string_lits
    <- content QUA string OF alab;
    declared := string$ read()
  END ENDWHEN
  WHEN number THEN BEGIN
    VAR n: 1000 .. 9999
    := content QUA 1000 .. 9999 OF alab;
    next_number := n + 1
  END ENDWHEN
  OTHERWISE
  END CASE

COMMENTS:

- Replacing the binding of a union variable (carrying with it a discriminant) by the binding of an overlay variable allows for further auxiliary transformations; "alab <- TOTALCOPY (blab)" ends up as "alab := blab" for instance.

- Reasons for updating the struct by assignments to individual fields of the posttrans type rather than by a semantically equivalent single assignment to a new object of that type are discussed in Chapter 6 below.
4.4.7 Multicopy - binding (multiple copies)

RESTRICTIONS:

- argument type must belong to a sharegroup whose members are never selectively updated;
- no "same-as" comparison ("<->") may be used within the sharegroup, nor value comparisons qualified by this sharegroup.

SUMMARY:

replaces shared bindings by bindings to unshared (top-)copies of the object shared.

DECLARATIONS:

VAR x: amode SHAREABLE abx => VAR x: amode

OPERATIONS:

Bindings

x <- y => x <- TOPCOPY(y)

y <- x (where x is transformed, y not) => y <- TOPCOPY(x)

REFTO x => REFTO (TOPCOPY(x))
4.4.8 Indices - structural (indices for traversers)

RESTRICTIONS:
- argument must be a sequence.

SUMMARY:
maps the sequence and its traversers onto a struct containing a
sequence, a possibly unbound index component corresponding to each
traverser, and (if not of manifest size) an integer component
maintained to give the current size.

DECLARATIONS:

CAPSULE b = (STATE:
  SEQUENCE 25 OF e TRAVERSED p;
  f = PROC (x: UNPROTECTED TRAVERSER OVER b);
  BEGIN ... END )

=> CAPSULE b = STATE:
  STRUCT (POSSIBLY p: 1 .. 25;
          seq: SEQUENCE 25 OF e);
  f = SELF PROC (POSSIBLY x: 1 .. 25);
  BEGIN ... END )

OPERATIONS:

Traversers

RESET p  =>  p <- 1
MAKEFINISHED p  =>  p <- NONEX
NEXT p

\[ \Rightarrow \]

WITH p DISCERNIMATE

WHEN exp: EXISTS THEN

IF exp = 25

THEN p <- NONEX

ELSE exp := exp + 1

ENDIF ENDWHEN

OTHERWISE

error ("advancing finished traverser")

ENDDISCRIM

(\# p \#)

\[ \Rightarrow \]

[ YIELDOF

WITH p DISCERNIMATE

WHEN exp: TOTALCOPY EXISTS

THEN YIELD exp ENDWHEN

OTHERWISE

error ("detraversing finished")

ENDDISCRIM ] OF SELF

COMMENTS:

- Operations requiring a change of size will check that each traverser (other than the one used in the operation, and the unprotected traversers) is in the unbound state corresponding to being FINISHED.

4.4.9 General comments

- In general a binding transformation "flows on" to apply equally to all discrimination branch variables in discriminations on the transformation's argument.

- Further examples of auxiliary transformations in conjunction with combinations of major transformations (such as pushout and nullfor-nonex by range (NIL), which introduce references with possible NIL values) will be seen in Chapter 6.
4.4.10 Principles of representation

The transformations embody a number of basic relationships between abstract and concrete data spaces that may be called principles of representation. Some of these principles are embodied in the transformations themselves, but others must be discerned in the way in which the application of a chosen series of transformations produces Maclan programs from general BSL programs.

Representations in general are mappings between an abstract data space and a concrete one, such of those of Koala and Maclan. A completely general form of mapping between data spaces is to map every occurrence of a particular construct in one space always onto a single combination of constructs in the other. The Koala data space has mutable binding of components, variable-sized objects, non-linear tree structures, implicit size and type information in objects; Maclan has relatively fixed bindings, fixed-size objects, linear structures only, no implicit structural information, the notion of references. Thus it would be sufficient for instance for every Koala binding (which might be mutable, or possible, or shareable, or none of these) to be represented as a Maclan NREFTO reference variable. Likewise every articulation relation of each sequence or tree could be represented as a reference, giving those relations the variability that they need to enable components to be added and removed. The transformations are indeed capable of producing such a simple equivalencing of the two data spaces using the evert, pushout and nullfornonex transformations for all bindings, with overlay for unions where these exist.
The explanatory power of such an operational semantics equivalence of data spaces is clearly inadequate and so also is the efficiency of the resulting concrete programs in practice. A satisfactory explanation (and better efficiency) requires more structuring of the mapping, and a finer distinction between specific points in the abstract data space (i.e. specific data structures); and data independence requires that the mapping give a choice of many points in the concrete space for a point in the abstract space. Both structuring and choice are provided by the stepwise application of transformations.

Distinguishing between specific abstract structures depends on the detection of regularities, invariants and restricted subsets of applied operations in the abstract type definitions. Some of these properties are necessary for certain transformations to be permitted at all. But in these cases and also those where they are not strictly necessary, it is these properties that lead to choices of representation other than simple equivalencing.

Some examples of the principles of representation we can identify in the transformations, alone and used in combination, are as follows.

(1) Regularity of structure. Maclan's restriction of types to a single degree of freedom gives it the invariant structural regularity that allows simple allocation and indexing in the machine data space (i.e. using fixed-size objects and a computed offset from a base address). Transformations that ensure that all of a sequence's elements are of the same size attain this regularity. Transforming the sequence's component type to fixed size (by padout or evert of that
type) or interposing a fixed-size object for each component (pushout of ALL of the sequence) are common. Where components are themselves variable sequences containing fixed-size components, incorporate will reduce the structure to one having fixed-size components, though of variable number. Pushout can also regularize structures in conjunction with overlay: for example, a union of types of different sizes can be represented, using pushout, as an overlay of references to the different types, the variability in size of the objects bound being reduced to variations in qualification of the references.

(2) Invariance of structure. Variation in structure (variable sequences, possibly unbound components) can be reduced to lesser degrees of variation or to none at all. Incorporate is able to represent a sequence of a variable number of objects each themselves of variable size by a fixed number (one or two) of variable sequences. Evert (in combination with pushout) reduces variability in an object's size to the more controllable regularity of a variable number of objects each of fixed size.

(3) Values to represent structure. A simpler structure can represent a more complex, encoding the "lost" structural information in the values of the structure's components. Direct applications are incorporate, overlay and nullfornonex. The broadest applications of this principle are in pushout; but this transformation's use of the principle is so broad that the "use of references" is better considered to be a distinct principle of representation. Representations of shared bindings, possibly nonexistent bindings, structural articulation and so on rely heavily on the use of references, and the one
reference variable may be able to represent all of these at once. The indices transformation takes a constrained flexible binding and represents it as a "local reference", an integer value whose arithmetic properties preserve the relations of the binding's states.

(4) Simpler instances. Data types or instances that use only certain subsets of their possible primitive operations can be seen as small islands of concrete data space in a sea of the more abstract. Expressing them as simpler equivalents makes it evident that they are closer to the concrete space than first appeared. The auxiliary transformations apply this principle; multicopy does also, and shares the attribute of having stringent restrictions. These conditions constitute the testing of the actual data space behind the abstract notation.

(5) Simple equivalence. It is always possible to apply the following transformations to map Koala onto Maclan:

- indices, for traversers of any sequence (the element of choice here is in the timing of applying the transformation, and the treatment of the resulting possibly unbound components);
- pushout, followed by nullfornonex by range (NIL) on every binding;
- evert, pushout and nullfornonex on all sequences (an extension to evert is needed for sequences which apply the PREVIOUS operation).
4.5 Formal description and implementation

4.5.1 Introduction to Xlan

A more formal description of the transformations serves also as their implementation. For this we employ the transformation language Xlan, developed and implemented for this project. Xlan is a procedural, block-structured programming language which manipulates program text in the form of parse-trees. Parse-trees are structured objects constructed from character string values and other parse-trees. Parse-trees are typed and can be decomposed in ways depending on the structure of their type. Any parse-tree object has a unique parent parse-tree. A type is defined by a rule in a labelled EBNF grammar, and possible structures are concatenation, alternation (including optionality), and repetition. The grammar applied by a particular program is defined by a set of declarations in the program.

Statements in Xlan include conventional assignment to variables, conditional statements and procedure calls. The flow of control is also regulated by selective traversal of parse-trees and by pattern matching. Parse-trees are selectively updated by the REPLACE operation which replaces a component parse-tree by another of the same type.

As its first action an Xlan program implicitly reads in an input text file, parses it as an instance of the grammar's "program" rule, and assigns it to the variable "the_program". At the end of the program execution this parse-tree, possibly modified, is output as text. An Xlan program therefore constitutes a general source-to-source
transformer for programs in a given grammar. Our concern is the description of transformations for BSL; in what follows we shall use only the labelled EBNF grammar for BSL given in Appendix 3.

An example of an Xlan statement is part of the auxiliary transformation to reduce copying redundancies, described above [section 4.2.2.2, no. 3].

FOR ALL asg : assignment IN the_program SUCH THAT asg.rator MATCHES "":="
AND asg.rhs MATCHES "copier"
DO WITH asg MATCHING "objexp` := r: copier"
DO REPLACE asg.rhs BY COPY (r.copied)
ENDWITH ENDALL

This implements the part of the transformation that deals with total-copy assignment of a top- or total-copied expression, and has effects such as

a := TOPCOPY(b) => a := b
b := TOTALCOPY(a) => b := a.

The iterative construct "FOR ALL ... ENDALL" selects from a particular parse-tree all instances of a named rule (in this case, all assignment statements) and assigns them in turn to its control variable, "asg". If the IN clause is omitted, the program parse-tree is taken by default. For each asg value satisfying the SUCHTHAT predicate, the statements between DO and ENDALL are executed.

In the above example the predicate depends on the decomposition of an assignment parse-tree, and simple pattern-matching. This form of decomposition applies the labels from the "assignment" rule to select
component parse-trees from the "assignment" parse-tree currently assigned to "asg". The relevant rules are:

```
assignment = lhs: objexpr  rator: asgrator  rhs: expression;
```

```
asgrator = '(-' | ':-' | ':=' | '%yield'.
```

"asg.rator" selects an "asgrator" from "asg"; "asg.rhs" selects an "expression".

The first part of the predicate succeeds for all BSL total-copy assignment-statements for which the "asgrator" component is of exactly the value of the character-string literal to be matched. The second part names the "copier" rule whereas the type of the parse-tree to be matched is "expression". Such a match succeeds only if that parse-tree and all its descendent parse-trees, down to but excluding a "copier" instance, are of types which have alternation structures; alternations are transparent in matching. The "expression" parse-trees for "TOPCOPY(x)" or "TOTALCOPY(a + b)", for example, consist of alternation instances ("expression", "factor", "objhead") down to a "copier", and for them the matching succeeds.

For successful values of asg the action taken is a further pattern match, which in this example always succeeds. The pattern match has the side-effect of assigning the "copier" descendent of the "assignment"’s rhs "expression" to the implicitly declared variable "r". The parse-tree for the inner expression of "r" then replaces the whole rhs of the assignment; it has the correct type ("expression") but must be copied because every parse-tree must have a single parent.
4.5.2 Parsing

A parse-tree of a particular type can be constructed by evaluating a PARSE operator which names that type as its first argument. The second argument is a "parse-string", which is written as the concatenation of text strings (delimited by single quotes) and parse-trees, all between double quotes. In the pushout transformation, for example, a new object-expression containing FOLLOW is created for applications of the argument in BSL object contexts.

PROCEDURE pushout_object_context (argument: identifier);
BEGIN
(* allow for argument to be field, variable or function *)
(* assume declarations have unique identifiers *)
FOR ALL ob: objexpr
  SUCHTHAT ob MATCHES "[identifier '§'] argument [actualparapack] ['qua' formaltype] possofobjexpr"
  AND NOT in name context(ob)
  DO REPLACE ob BY PARSE objexpr "'%follow (' ob ')')" ENDALL END

The PARSE operator attempts to create an "objexpr" parse-tree from the whole of its parse-string. If it fails the program aborts; if it succeeds the new parse-tree is the value of the operator.

The operation of parsing has the following characteristics: it

- proceeds top down, left to right with backtracking;

- absorbs the minimum number of elements into repetitions, and takes the absent case of optionalities, first;

- splits text string constituents of the parse-string into lexical tokens, and disregards any excess white space;
accepts the next constituent of the resulting parse-string if

- it is a parse-tree of exactly the type wanted next (types
  have name equivalence),

- it is a lexical token of equal text string value to that
  wanted;

- incorporates copies of parse-tree constituents of the parse-
  string into the parse-tree being constructed.

More complex parse-strings are formed using conditional expres-

- sions and an explicit concatenation operator "&". The evert transfor-

- mation acting on a "GROW BEFORE" statement [section 4.4.3], for exam-

- ple, has a result that depends on whether that statement's traverser-

- expression is "FIRST" or not. If it is, the posttrans statement's

- inner discrimination ("WITH pred OF cpr ..." in that example of "GROW

- BEFORE cpr ...") is unnecessary, and its default case ("seq <- newel")

- is all that is needed. Consider transforming the inner block of the

- posttrans statement:

- "seq <- newel"
FOR ALL growst: grow SUCH THAT...
DO WITH growst.grower MATCHING "%before" t: travexpr
DO ...
PARSE statement
"%begin %var newel: element %shareable eltgroup
'elem %of newel <- growst.wither ";'
'succ %of newel <- excurr;"
& IF NOT t.termtrav MATCHES "%first"
THEN "%with pred %of"
t.termtrav.traverser QUA identifier
t.hdr %discrim
"%when expred: %exists %shareable eltgroup
"%then succ %of expred <- newel %endwhen
"%otherwise"
ELSE "" ENDIF
& "seq t.hdr <- newel"
& IF NOT t.termtrav MATCHES "%first"
THEN "%enddiscrim;
"pred %of t <- newel"
ELSE "" ENDIF
& "%end"

The use of the variable "t" is noteworthy in this example. In transforming a GROW statement that applies to a remote object, such as

GROW BEFORE cpr OF x OF y WITH volume2

"t.termtrav" is a "termtravexpr" matching the identifier "cpr";
"t.hdr" would match "%of x %of y". Consequently "seq t.hdr" in the third part of the parse-string parses as an object-expression, "seq %of x %of y".

Transforming a named traverser expression also uses the variable "t". The posttrans equivalent of a named traverser-expression is an object-expression, because the traverser is replaced by a struct component. The posttrans component identifier is the same as the pre-trans traverser identifier. But the parse-tree structure of the
object-expression differs from that of the traverser-expression, although the text strings they contain happen to be the same: "cpr %of x %of y". To form the object-expression from the traverser-expression parsing of this string it is necessary to retrieve the "identifier" parse-tree from the "travexpr" parse-tree "t", to locate a parse-tree acceptable to the parsing as a descendent of "objexpr". (QUA forces the selection of one case from an alternation; in this example we have omitted the surrounding predicates which ensure that this forcing is permissible.)

4.5.3 Decomposition

Parse-trees are decomposed according to the structure of their rule. The structuring methods of EBNF are concatenation, alternation and repetition; the corresponding methods of decomposition are:

- concatenation - selection, using label
- alternation - discrimination, using rule name
- repetition - indexing

Selection has been seen in the example above.

Discriminating an alternation parse-tree may be by a discrimination statement or by forcing a particular expectation using QUA. The discrimination statement resembles a BSL discrimination, but the branches apply the rule names or text string literals of the alternation. An "optional" construction is regarded as an alternation with one label, "EXISTS".
Consider for example a function to determine whether a BSL object-expression has generated shareability.

```
FUNCTION is_generated (ob: objexpr): boolean;
BEGIN
WITH ob DISCRIMINATE
WHEN cmpndobjex
THEN RETURN false END WHEN
WHEN hd: objhead
THEN WITH hd DISCRIMINATE
    WHEN "%self"
    THEN RETURN false END WHEN
    WHEN follower
    THEN RETURN false END WHEN
    OTHERWISE (* newgen and copier cases *)
    RETURN true
END DISCRIM
ENDDISCRIM END.
```

A repetition parse-tree is indexed by an integer expression within square brackets, or by dotted selection using one of the reserved words FIRST or LAST. The n-th actual parameter expression in a BSL "opncall" (operation call) "c", for example, is denoted by `c.paras.params[n].`

4.5.4 Determining context

The effect of a transformation on a particular construct frequently depends upon the context of that construct in the program. Xlan therefore also articulates parse-trees to give access to the "context" parse-tree of any parse-tree. The language rules enforce the property that any parse-tree is a component of at most one other parse-tree, and therefore has a unique immediate ancestor. The "context" of a parse-tree is the closest ancestor (possibly at some
remove) which is of concatenation or repetition structured type. In general a parse-tree of a particular type may be in contexts of many different types.

We discount alternations in determining context in order to approach as closely as possible to the intuitive notion that the context should be the smallest unit containing the text strings immediately surrounding those in the parse-tree of interest.

Because of the variety of possible contexts, a further form of discrimination statement is used to determine context. We can find out whether a given objexpr is in a name context [section 4.4], for example, by the following:

```
FUNCTION in_name_context (ob: objexpr): boolean;
BEGIN
  WITH ob DISCRIMINATE CONTEXT
  WHEN lhs OF asg: assignment THEN
    RETURN asg.rator MATCHES "< | *yield"
  ENDWHEN
  OTHERWISE RETURN false
ENDDISCRIM END.
```

The discrimination branch, if it were "WHEN asg: assignment", would choose all assignments that had the objexpr on either left-hand side or right-hand side (parsed via "factor" to "expression" in the latter case); in each case the context of the objexpr would be an assignment. Stipulating the selector ("lhs") of a possible context is optional in general, but necessary in this case to determine that the objexpr is indeed in name context, i.e. on the left-hand side of a binding assignment, and not on the right. In this branch of the statement "asg.lhs" is the same parse-tree as that assigned to "ob".
To express some transformations requires a parse-tree's nearest ancestor that is of some particular type. For this Xlan provides an ANCESTOR operator. An example of its use is to select all those object-expression applications of the BSL reserved word "SELF" that relate to a particular capsule being everted, rejecting those relating to any capsules nested within it.

PROCEDURE evert_selves (arg: capsuledec);
BEGIN
FOR ALL sobh: objhead IN arg
   SUCHTHAT capsuledec ANCESTOR sobh = arg
       AND sobh MATCHES "%self"
DO
  evert context of self (sobh)
ENDALL END.

Equality, like assignment, considers the identity of parse-trees rather than simply their values. Hence the first part of the predicate is true only for those "objheads" whose smallest enclosing "capsuledec" (capsule declaration) is that one assigned to "arg". This procedure shows an example of an iterator constrained to descendents of a particular parse-tree. In this example the predicate would select the same values of "sobh" if the constraining clause were absent; however its presence reduces the size of the candidate set of "objexpr" parse-trees, achieving advantages for the programmer and a cheap, simple implementation of Xlan.

4.5.5 Repetition operations

A parse-tree whose context is a repetition has no selector, but the expression "ORDINAL(p)" returns the ordinal number of its position in the list. This property is applied in the pushout transformation,
for example. Applied to an operation’s formal parameter, the transformation must add "%refto" to corresponding actual parameter expressions. Parameter correspondence in BSL is by position; in an actual parameter list "al", the actual parameter corresponding to the formal parameter "fp" is denoted "al[ORDINAL(fp)]". Given the "formalpara" which defines pushout’s argument in such a case, and the operation’s “identifier”, this part of the transformation is defined as follows:

```plaintext
PROCEDURE pushout_actuals (fp: formalpara; opid: identifier);
  VAR formal_position: integer;
  BEGIN
    formal_position := ORDINAL(fp);
    FOR ALL call: opncall
      SUCH THAT call.calledop MATCHES opid
      DO REPLACE call.paras.params [formal_position]
        BY PARSE expression
          "%refto" call.paras.params [formal_position]"
    ENDCALL END
```

Repetition parse-trees can be constructed by concatenation as well as by parsing. The operator LISTCONCAT, as in "a LISTCONCAT b", takes arguments of the one repetition type and yields a parse-tree of the same type, automatically copying its arguments.

Single parse-trees can be inserted as elements into a list. One requirement of the evert transformation, for example, is to insert additional operation-declarations into the everted capsule’s list of declarations.

```plaintext
INSERT PARSE incapsuledec
  "index = %self %func (indexpr: integer): edgelem;"
  "%var counted: integer := 0; ..."
  "%begin ... %end"
AFTER everted_capsule.capcontent.LAST
```

Inserting an element or concatenating lists automatically adds list-
separator elements (";" in this case). Only literals are permitted as separators, which makes this possible.

4.5.6 Pattern matching

Pattern matching is applied either as a predicate (the MATCHES operator) or in its own form of conditional (WITH ... MATCHING ... DO ... ENDWITH). The process of matching resembles parsing, with these differences:

- the target is a denoted pattern, rather than a named declared rule;
- the source is a single parse-tree, rather than a string of literals and parse-trees;
- the source parse-tree is decomposed as necessary to match a pattern element, whereas parsing never decomposes parse-trees;
- parse-trees of the source are assigned to pattern variables by sharing, whereas parsing copies them;
- a pattern may contain applied variables, as well as the rule names, literals and labels which it has in common with rules. Applied variables match a parse-tree of the same type and equal value. Value equality of parse-trees is defined as their having identical structure (the same alternative, or the same number of list elements) with each component having recursively equal values; string literals are equal in the conventional way. Previous examples have applied existing variables in patterns, but pattern variables may also be used.
An example is the auxiliary transformation called conditional flattening [section 4.2.2.2, no. 4], for the cases where the conditional-statement controls a choice of single assignment statements.

```
FOR ALL s: statement
    SUCHTHAT (WITH s MATCHING
        "'%if' x: expression '=%' c: expression
        "'%then' t: objexpr how: asgrator c
        "'%else' t how x "'%endif"
    DO
        NOT is_shareable(x) AND
        no_side_effects(x) AND no_side_effects(c)
    DO
        REPLACE s BY s QUA condl.elsie QUA EXISTS.falsebr
    ENDCALL
```

The pattern matches a BSL conditional-statement that has a condition that consists of a total-copy equality comparison, and branches that are assignments of the one kind ("how") to the one left-hand side objexpr ("t"), and whose right-hand sides are the expressions ("x" and "c") compared in the condition. We assume an Xlan boolean function that determines whether an "expression" is free of side-effects; a sufficient criterion is that the expression incorporates no operation-calls. Thus for example the result of nullfornonex by range (NIL), after evert and pushout have formed a linked list, leads to an application of this auxiliary in what was after evert a binding assignment between possibly unbound names:

```
IF exc = NIL THEN curr OF cpr ::= NIL ELSE curr OF cpr ::= exc ENDIF
=> curr OF cpr ::= exc.
```
4.5.7 Implementation

Xlan was implemented as an interpretative system written in Simula 67 on a Univa 1100/42. The interpreter read the EBNF rules as data and represented its named and anonymous phrase structures by a "grammar network" of class instances. The top-down, recursive, back-tracking parsing routine traversed this network starting from the rule for "program". It consumed lexical items from a built-in analyzer that operated on parse-strings and on the text of the BSL program to be transformed, recognizing only identifiers and numbers and removing white space. This implementation made it simple to effect grammatical and lexical changes while BSL was under development.

The parse-tree was represented by a tree of instances of two classes. One class, named "node", corresponded directly to nodes of the parse-tree. The other was a doubly linked list-element containing a reference to a node. Each node contained a reference to its type in the grammar network, a reference to its parent parse-tree or list-link node, and a reference to its descendents in a way that depended on the structure of its type. The single descendent of a node of alternation type was a parse-tree node of the chosen alternative type. A node of concatenation or repetition type had a list referring to its many descendent parse-tree nodes. Each link in a concatenation node's list could also contain a reference to a selector label within the grammar network representation of the concatenation type. Its descendent node was then the corresponding component of the concatenation.
Programs written in Xlan were translated by hand to Simula creation of class instances and calls of procedures which manipulated these representations. Decomposition of parse-trees was implemented by procedures to select labelled descendents of concatenations, and index repetition lists by counting. Ancestors and contexts were found by following chains of references to parents. Alternatives were discriminated by considering the type of the descendent node, and contexts by the type of the context node itself.

Iteration ("FOR ALL") was implemented by a Simula class "iterator" used as a generator, a "SUCHTHAT" predicate being provided as a procedural parameter to a procedure within the class. Pattern matching was not implemented as a distinct operation, but translated by hand into equivalent discriminations and use of a function comparing the value of parse-trees.

The interpretative system occupied about 6000 fairly dense lines of Simula 67 source code. Simula proved particularly well adapted to this work. List and node manipulation were straightforward; and each FOR ALL statement, to any degree of nesting, translated into the creation of a new instance of the iterator class, the declaration of a local boolean procedure for its predicate, and a loop which called the iterator's "get-next" ref (node) function and was controlled by its "there-is-more" boolean function.
4.5.8 Experimental work

The interpretative implementation of Xlan was used to demonstrate and debug an Xlan description for three transformations, one of each class. This provided a testbed for the design of both BSL and Xlan. All parts of this dissertation were influenced by this work. The transformations implemented were evert, pushout and an auxiliary to remove redundant brackets (not described here). No symbol table facilities were implemented, so each BSL declaration was required to have a unique identifier (this is true of the examples given in the preceding sections also).

The sizes of the Xlan procedures were: for evert, 330 lines; for pushout (for explicitly declared arguments only, i.e. excluding "ALL" as an argument specifier), 94 lines; for bracket removal, 15 lines. The transformation of GROW statements using traversers alone occupied 88 lines in evert. Translations by hand to calls on the Simula 67 interpretative system occupied 830, 340, and 51 lines respectively. Translation was by manual macro expansion, with no attempt to optimize.

Examples of CPU times for running these procedures were 16.6 seconds (plus 11.0 seconds spent in Simula system garbage collection) for pushout and the auxiliary applied 3 times each to distinct identifiers in a 12-line BSL program; and 18.1 seconds (plus 34.1 seconds in garbage collection) for evert applied to a 23-line program. These times included initial parsing and final printing of the program.
The size of BSL programs that could be handled was restricted by operating system limits on memory allocation, and garbage collection times were relatively high for the same reason. The method of representing parse-trees and the backtracking parser algorithm were responsible: the many class instances which represented the grammar and program structures used large amounts of memory, and backtracking in the parser caused many structures to become garbage when partly built.

Further experimental work was done to assist in the design of type-splitting, described later [section 5.2].
4.6 Discussion: transformations and the machine end of BSL

4.6.1 A structured description of representation

The differences between Koala and Maclan are sufficient to make the task of relating programs in the two a non-trivial matter. This chapter has provided the mechanism of correctness-preserving transformations of data structures and bindings which relates particular programs within the Broad Spectrum Language, and has presented a practical means for defining such transformations procedurally in Xlan.

The use of transformations to move programs along the spectrum provides a degree of data independence and structures the description of a program's implementation. Data independence derives from the choice of different possible representations for an abstraction, as was seen in section 4.3 and will be seen further in the next two chapters. The structural description comes from factoring a program's implementation along two axes.

The first factoring is by the established method of partitioning the program's set of data items and considering each partition independently. This is done, in this system and in those describing representation by refinement (the capsule languages), by declaring capsules; and in Low's system by data flow analysis.

The second factoring is by time-ordering the application of transformations to a capsule's definition and/or applications. This has no parallel in non-transformative systems. A complete description of a program's implementation is thus a time-ordered sequence of
transformations applied to specific program constructs. Not all dif­
ferent sequences of transformations produce different results:
equivalent implementations may be had with some variation in the order
in which transformations are applied. A consequence of this is
described in section 5.2.

4.6.2 Comparison with other transformation systems

The comparison with other systems of program transformation has
three aspects: the "amount of work" required of one transformation
(i.e. its size), the manner in which transformations are evoked, and
the notation in which they are expressed.

4.6.2.1 Size of transformations

The transformations contrast strongly in scope and power with the
typical transformations found in much of the literature [Loveman 1977;
Standish et al. 1976; Wegbreit 1976; Arsac 1979; Darlington and Bur-
stall 1973]. They are "large" transformations, in Arsac's phrase
[Arsac 1979, p.43]; that is, each transformation operates on more
than a highly local part of the program, and the transformations in
the system are relatively few and independent of each other. This
contrasts with the many "atomic", highly predicate-driven transforma-
tions of a typical production system [e.g. Kibler et al. 1977].

The published program transformation systems are in the main con-
cerned with automatic or systematic manual optimization of programs in
a narrow spectrum language, and work in the area of formal correctness
of transformations. Our aim is rather to transport a program systematically along a data space spectrum. Clearly our transformations are too complex to be provable in the current state of the art.

The transformation system contains many transforming modules, only some of which could be labelled "atomic transformations". These are the auxiliary transformations, which apply automatically to small contexts. The remaining modules are comparable in size to the auxiliary modules, but appear combined in groups which each constitute a major transformation. These modules differ from atomic transformations in two respects: none of them is by itself meaning-preserving; and the most apt description of their relationships is by direct transfers of control. For these reasons it is appropriate to identify the tightly coupled collections of modules, and not individual modules, as one transformation.

4.6.2.2 Invocation

Major transformations are invoked by name, in contrast to those systems that make the choice automatically. In such systems the choice is based on attached predicates, costing functions, and internal suggestions. In explicitly choosing transformations, the user is describing a chosen representation of a program. The restrictions on legal applications are on logical grounds alone, and do not prevent the choice of highly inefficient representations (such as a linked list for a frequently indexed sequence).
4.6.2.3 Notation

Xlan was designed because no other available system was appropriate for both describing and implementing transformations of the required size. Pure text-string pattern matching (e.g. SNOBOL) is inadequate for large systems because it discards the structure of the subject language. Production systems are suited to small, local transformations, but are known to be difficult to write and to debug [Moran 1973]. The flow of control is not at all evident from the form of the program, and highly structured data is described poorly.*

Xlan's strong data control and procedure control facilities are necessitated by the size of the transformations and of the BSL grammar (although by modern standards the latter is not a large grammar, having 71 rules as against 123 for Pascal). To cover a wide range of the data program's constituents in the course of one transformation requires straightforward access paths through its structure. The complexity of the transformations means that if they are to be written reliably, the notation must be able to impose definite control structure on their constituent operations, and also to structure access to data of even small scope in the data program. As well as being more reliable for this purpose, Xlan is usually also more efficient than a production system. Whereas the speed of a production system is restricted by the difficulty of selecting applicable productions from a

* The Compiler Compiler [Brooker et al. 1963] was brought to my attention after this work was completed. It provides a weaker, although similar, form of parsing and decomposition to Xlan, but lacks the context sensitivity that we have found extremely useful. It is not apparent that it is capable of internally transforming its parse-trees.
large number of possibilities, the speed of a procedural system
depends only on the number of operations actually performed.

4.6.3 The transformations and BSL

4.6.3.1 BSL's effect on the transformations

BSL's design gives several advantages to the writer of transfor-
mations. Working in a Broad Spectrum Language rather than two or many
distinct languages means that extraneous transformation between
languages is avoided. The principle of strong correspondence in BSL's
design makes the expression of the transformations relatively
straightforward; many of the properties of any objects or bindings in
the computation may be readily determined by simple inspection of a
corresponding bindable construct in the program. Having the high-
level data abstractions of "sequence" and "tree" made explicit makes
the task of the transformations very much simpler than in a lower
level language, for which a transformation system must include more
sophisticated analysis to determine which abstraction is actually
embodied in a particular use of a lower level structure [Standish et
al. 1976]. The fact that the transformations described here do not go
far beyond the representation of the abstractions supplied by Koala
does not mean that the more primitive linked record and array struc-
tures would have been a better starting point to choose, in order to
get a more general system: handling such structures by transformation
would be akin to attacking in similar fashion the control structures
of programs which are allowed free use of GOTOs and label-valued vari-

On the other hand, these properties of BSL also limit the transformation system. Two examples are its strong typing and its sharegroups. Maclan's strong typing renders impossible some desirable extensions to the transformations. Incorporate cannot apply to sequences of structs, for example, because BSL cannot define a sequence of repeated patterns of a mixture of types. A declarative form of the dynamic "ghost data structures" [Desjardins 1974] might have met this need.

A more basic limitation appears in multicopy's restriction that there be no selective updating of any members of the argument's sharegroup. This condition is much stronger than necessary; it would be sufficient if there were no updating of those objects bound to the argument, after they had become so bound. The otherwise useful notion of the sharegroup does not admit this finer distinction. Similarly it prevents us considering a change of representation during an object's lifetime, appropriate to its changed expectations of operations.

### 4.6.3.2 Influence of transformations on the languages' design

The detailed form of the shareability declaration in Koala and the need to carry some form of sharegroups into Maclan both come from considering how to write the transformations in order to manipulate programs across the whole language spectrum. (The transformation system aside, the sharegroup is unnecessary in Maclan.) Consider two possible forms for defining sharegroup membership. The one adopted in
BSL we may call "attributive": membership is an attribute of each declaration.

SHAREGROUP kennel OF dog;

CAPSULE pair = (STATE: SEQUENCE 2 OF dog SHAREABLE kennel;
  feed = PROC (d: dog SHAREABLE kennel);
  ...).

The alternative form is "enumerative": the sharegroup declaration has no name, but enumerates its members.

SHAREABLE (ALL OF pair, d OF pair.feed): dog;

CAPSULE pair = (STATE: SEQUENCE 2 OF dog;
  feed = PROC (d: dog);
  ...).

The two forms declare the same sharegroup membership but behave quite differently in program transformations. The majority of transformations concerned with shareability depend on being able to determine and manipulate the shareability of a construct readily, and not on being able to discover all members of a particular sharegroup. For this the attributive form is clearly superior, as may be seen if multicopy's removal of a construct from its sharegroup, and pushout's modification of its type (and hence the enumerative form's description of the shared construct to "FOLLOW OF pair.ALL", say) are considered.

It is not immediately clear which form is the better for human comprehension of programs. Other languages with related constructs [Euclid; Ada] have used an attributive form of declaration, and the almost universal form of defining type for variables is similarly a distributed one. The simplicity of transforming programs with this distributed form suggests that the form is also simpler for humans to
comprehend, given the relation between human understanding and transformations proposed by one school of natural language studies [Greene 1972]. There is however no evidence that the kind of transformation we have used is involved in natural language processing, where studies have described only much smaller transformations.

Otherwise the transformations influenced the design of the languages at a more abstract level. They encouraged the adoption of the principle of strong textual correspondence, for example, and the somewhat ungainly forms of object-expressions and operation calls.

4.6.4 Limitations of Xlan

Xlan's advantage over purely textual manipulation languages is that it has strongly typed data. (A transformation is prevented from replacing a BSL statement by an arbitrary expression, for example: this is an Xlan type error.) Its shortcoming is that the grammar in terms of which it parses and manipulates a program is context-free. An Xlan program therefore has no direct notion of the scope of declarations, and furthermore Xlan cannot prevent a replacement which is illegal because of its context (attempting to add FOLLOW to a non-reference-typed object expression, for instance).

Xlan's notions of scoping could be improved by adding data structures to allow ready manipulation of symbol tables. This was not done because it was outside the scope of our enquiry.

A superior transformation language might allow for the use of a more context-sensitive grammar. But however this problem is
approached the question must be faced of the consistency of the whole program (in particular, that of declarations and applications, in class and type) while it is being changed piecemeal. The context sensitivity of the definition will have to be suspended at certain times if the parts of a transformation are to continue to be done in sequential steps, because as remarked earlier the program may be internally inconsistent during this process. The price paid for working under the greater protection of a context-sensitive definition would therefore be added complexity of the transformation language. There is also a danger that the subject language grammar and the expression of the transformations would become too complex for ready comprehension. The form of a language to manipulate programs through such definitions is unexplored.
5 MULTIPLE REPRESENTATIONS

5.1 Introduction

5.2 Type-splitting
  5.2.1 How to split capsules
  5.2.2 When not to split capsules
  5.2.3 Applying transformations
  5.2.3.1 Capsule descriptions
  5.2.4 Discussion
  5.2.4.1 Necessity for capsule descriptions
  5.2.4.2 Overhanging transformations
  5.2.4.3 Descriptions and classes of transformation

5.3 Assignment compatibility between different representations
  5.3.1 Value manipulation and bundle interface operations
    5.3.1.1 Variable-length sequences with unshareable components
    5.3.1.2 Shareable components
    5.3.1.3 Fixed-length sequences
  5.3.2 Discussion
  5.3.2.1 Alternative mechanisms
  5.3.2.2 Variant forms of operations
  5.3.2.3 Choice of language for vmos and bios
  5.3.2.4 Particular transformations and the bios
  5.3.2.5 New object generation
  5.3.2.6 Trees and structs
  5.3.2.7 Conclusion
5.1 Introduction

In this chapter we turn to the interaction of the transformation scheme with the programming language, and in particular consider how to deal with the problem that the requirement of strong correspondence in our design is to some extent at odds with that of multiple representation. The transformational mechanism of "type-splitting" and the BSL mechanisms associated with a "bundle" of capsule definitions are the result. The transformations, formation of bundles, and type-splitting combine in what we term the "transformation system".

The bundle is the key to multiple representations. The transformation system ultimately produces a Maclan program in which each capsule in the initial Koala program has become a bundle. The Koala capsule may have several representations, each described by a Maclan capsule in that bundle. The bundle gives its member capsules a home and a common abstract capsule definition, but without generic types this is merely a convenience for the human reader. These capsules define distinct types, which BSL's strong typing normally renders incompatible, and ensures that every bindable construct in the program has a single type. Generic types provide the only exception to this. But generic types are constrained in ways which make feasible their implementation on the conventional machine. The major constraint is that they accept only those actual types defined by the capsules of one bundle. This relaxes strong typing sufficiently to allow us to define generic operations that implement copying operations between multiple representations of the one bundle, and thereby maintain for all representations these operations of their common abstract type.
The generic bundle operations are referred to as "value manipulation operations", or vmos. They define top- and total-copy forms of assignment, comparison and copying operations for parameters and yields of types from the bundle. They are needed even where the parameters to the operation are of the same type, and even where that type is the only capsule in the bundle, because in no part of the BSL spectrum but Koala do assignment, equality and copy generators exist as primitive operations for the structured objects which are the STATES of many abstract capsules.

The transformation system therefore encloses every capsule in a bundle. It does so before it applies the first transformation. This process also adds some operation declarations to the capsule, called "bundle interface operations", or "bios". The bundle has the same identifier as the capsule. It contains the original capsule and an ABSTRACT capsule which is a copy of it; it also contains generic bundle operations defining top- and total-copy forms of assignment, equality and copying operations (the vmos). The bodies of the vmos contain calls on the bios of their actual parameters. The forms of the vmos and bios depend on the state declaration of the capsule, as described later in this chapter.
5.2 Type-splitting

5.2.1 How to split capsules

To get multiple representations of an original capsule the number of capsules in its bundle must increase. An extra capsule is introduced into a bundle when one of its existing capsules is "split" in the course of applying any of the major transformations. Split capsules, like newly divided amoebae, are initially indistinguishable; but one is then given a unique new identifier and is subjected to the transformation exactly as if it had existed originally. The other remains untransformed and almost unchanged. Not all applications of transformations cause a type-split, as we call this process. Only a transformation which is applied to some proper subset of all the applied instances of a type will cause its capsule to split; those instances become applied instances of the newly created and transformed type.

The distinctive feature of a type-splitting transformation specification is not its form, but its arguments. It has a set of bindable constructs within the "environment" (dotted-name) parts of its arguments [section 4.2.1], where a non-splitting specification (of both structural and binding transformations) has only capsules. The environment parts may be bindable constructs (variables, parameters, components) or sharegroups; the set consists of the collection of all bindable constructs within individual arguments, plus all the members of any sharegroups.
It is a consequence of BSL's strong typing that all members of a sharegroup have the one type, in Maclan as well as in Koala. Different representations are different types in this respect as in any other, because otherwise the representation type of an object referred to by a Maclan reference variable of some sharegroup would not be known at compile time, and code generation for accessing its components or calling its self-operations would be impossible. A sharegroup is therefore indivisible when a type is split. The transformation system requires that where any argument's environment-part names a shareable bindable construct, all other members of its sharegroup must be in the set.

Consider an example of nested capsules with a sharegroup, and two variables of the capsule's type.

CAPSULE book = (STATE: SEQUENCE UPTO 25 OF edition SHAREABLE edlist;
SHAREGROUP edlist OF edition;
CAPSULE edition = (STATE: STRUCT (author: full name;
POSSIBLY etalia: name list;
title: string; ...);
language = FUNC (x: edition): tongue;
BEGIN ... END);
...);
VAR redbk: book; VAR bluebk: book;
Applying "evert(book)" would result in no type-splitting, only the transformation of capsule "book" as we have seen in Chapter 4. Applying "evert(redbk)", on the other hand, has the bindable construct redbk as argument, which is only a subset of the applied instances of "book"; the capsule is therefore split, one part is renamed and transformed, and redbk becomes an instance of the new type.
A transformation can split capsules to which it does not apply directly. From the same starting point as before, a split in "book" is caused by applying

\[ \text{pushout} \left( \text{YIELD OF redbk.edition.ordered_print} \right) \]

Notice that it is whether the argument's environment contains bindable constructs that determines whether the transformation splits a type; the transformation may be either binding or structural. The split here is between "redbk" and other instances of "book", whereas the transformation affects the new capsule's "edition.ordered_print"
operation. But because the argument names the whole capsule "edition" within "redbk", that capsule is not itself split within either book capsule.

BUNDLE book = [  
  CAPSULE ABSTRACT = ... (* exact copy of original *) ... ;  
  CAPSULE book = ... (* as above *) ... ;
]

CAPSULE book2 = (STATE: SEQUENCE UPTO 25  
  OF edition $ edition SHAREABLE edlist;  
  SHAREGROUP edlist OF edition $ edition;  
  BUNDLE edition = [ CAPSULE ABSTRACT = ( ... );  
    CAPSULE edition = (STATE: ... ;  
      language = FUNC (x: edition): REFTO (tongue);  
      BEGIN ... END; ... (* bios *) ... );  
    ... (* vmos *) ... ];  
  ... (* bios *) ... );

  ... (* vmos *) ... ];


BUNDLE book = [  
  CAPSULE ABSTRACT = ( ... );  
  CAPSULE book = ... (* as above *) ... ;
]
CAPSULE book2 = (STATE: SEQUENCE UPTO 25
OF edition$ edition2 SHAREABLE edlist;
SHAREGROUP edlist OF edition$ edition2;
BUNDLE edition = [CAPSULE ABSTRACT = ( ... );
CAPSULE edition = (STATE: ...
language = FUNC (x: edition: tongue;
BEGIN ... END; ... (* bios *) ... );

CAPSULE edition2 = (STATE: ...
language = FUNC (x: edition2: REFTO (tongue);
BEGIN ... END; ... (* bios *) ... );

... (* vmos of edition *) ... ];
... (* bios of book2 *) ... );

... (* vmos of book *) ... ];


Type-splitting does not "rename" a newly split capsule merely by changing its identifier, but as the splitting of the capsule "book2.edition" shows, certain applied instances of that identifier are changed also. Those that change with the capsule identifier are:
the argument set that caused the split; all instances within the capsule itself, except those that belong to a sharegroup that is of the other capsule's type; and all those that BSL requires to be of the new capsule's type. The last group includes any members of a sharegroup that has changed type, including those members declared in the "original" capsule. Thus splitting "edition" in "book2" gives the "language" operations in the two capsules the definitions


before the yield of the operation in edition2 is pushed out; but had "x" been shareable, both would have the same definition, namely

The sharegroup "edlist" in book2 is the argument of the transformation, and hence all of its member declarations, including the component type of the STATE sequence of book2, are renamed. The declaration in the capsule "book" is unaffected because it is a member of the unaltered sharegroup in that capsule.

A bindable construct of ANY shareability is no exception to strong typing, but can be bound to members of different sharegroups while these have a common type. It is therefore necessary to ensure that after a type is split, its pre-split instances of ANY shareability are not required to bind to objects of both post-split types. This is so because the split creates distinct textual occurrences of the ANY constructs of that type, with distinct post-split types (that is, loosely, ANY constructs are split along with their types).

It is a result of BSL's constraints on ANY shareability that the process of splitting a capsule described above also correctly handles its instances of ANY shareability. Only discrimination branch variables and the SELF of self-operations can have ANY shareability. Splitting a capsule creates distinct copies of its self-operations (as for all its operations); and by the nature of self-operation calls, an instance of either post-split type can be bound to the SELF of only those self-operations that are in its own type's capsule. In either post-split capsule the type of any operation's SELF is therefore only ever that of the capsule. A particular discrimination branch variable may bind to different objects at various times, but only to those objects yielded by the discriminated expression. The expression is of some particular union type which itself has no ANY shareability. The
discrimination branch variables are tightly linked to the discriminated expression, and transform along with it as a matter of course. The post-split type of the ANY variables is thereby fully determined.

It is for this reason that other bindable constructs, such as formal parameters, are not allowed to have ANY shareability. Unlike discrimination branches and self-operations, such constructs are neither tightly bound to one post-split type nor necessarily duplicated when the parameter's type is split. The latter is because unlike self-operations, operations with parameters and variables of the type being split can appear outside the textual enclosure of the type's defining capsule. As we have seen this problem does not arise with any other kind of shareability, because every other bindable construct is a member of a specific sharegroup and transformed with the other members of the sharegroup; and an unshareable construct is the sole member of a unique sharegroup in this respect as in all others.

5.2.2 When not to split capsules

The effect of applying a specified transformation can at times be had more economically than by actually transforming a capsule after possibly splitting it. Splitting and transformation can both be avoided if a capsule that is "completely equivalent" to the expected result is known to exist in the program already. ("Complete equivalence" of capsules means that they are semantically equivalent, in the sense already used in this work, and moreover make the same use of exactly the same data space.) The same effect as applying the transformation is then got by substituting the existing capsule.
identifier for that in applied instances of the type to be transformed. This procedure is generally cheaper than actually transforming the capsule.

"Descriptions" of capsules, comprising their transformational history, are used to characterize both existing capsules and those that would result from a specified transformation. Complete equivalence of capsules is therefore also defined as the equivalence of their descriptions. Thus a description should correspond to our intuitive notions of completely equivalent capsules, as we shall show in section 5.2.3.1 below.

Completely equivalent capsules arise by type-splitting and transformation. If one transformation is applied separately to several instances of the one type, or one set of transformations is applied to them in the same order, then the resulting capsules are completely equivalent. Depending on the choice of transformations the results of applying the set in different orders may also be so. For example, if "evert(redbk)" is applied to the original program in the preceding section, followed at some later time by "evert(bluebk)" (with no intervening transformations on these capsules), then the two types are completely equivalent and can be recombined. The same effect can be seen with the following set of transformations, in whichever order they occur:

- pushout (author OF redbk.edition.edition)
- nullfornonex by range(NIL) (others OF redbk.edition.edition)
- pushout (author OF bluebk.edition.edition)

The two transformations in either "edition" capsule affect different
components of its state, and are completely independent.

Note that completely equivalent capsules are not necessarily textually equal, even allowing for systematic renaming. The transformations

\[
\text{nullfornonex by range(NIL) (x),}
\text{nullfornonex by range(NIL) (y),}
\]
depending on the order of their application, produce different post-trans texts for "x <- y"; but the two posttrans programs are clearly completely equivalent. Capsule descriptions abstract away from the textual detail that would confuse such cases of complete equivalence.

5.2.3 Applying transformations

We now describe the process of applying a specified transformation, assuming that we can construct descriptions for existing capsules and for them as they will be after an intended transformation has been carried out; and that we can determine whether two descriptions are equivalent. The assumption is proved in the next section.

The first phase of the process extracts from the specification's arguments sets of arguments of the same type and similar environment parts, and thereafter these sets are treated separately. Choosing any argument from a set as its representative, a sequence of descriptions is built, one for each bundle scoping level in its environment part. Each description is associated with a "replacement set" of parse-trees in the program. Where only a subset of the corresponding capsule's bindable constructs is named, the replacement set consists of these
constructs. Where a capsule name or all of its bindable constructs are given, the replacement set is empty.

Consider for example a simple transformation specification such as


Descriptions of their hypothetical posttrans states are formed for "book", with the declaration of "redbk" as its replacement set; and for "edition", with an empty set. This posttrans description of book states that its only point of difference from the original "book" is that its nested capsule "edition" has a new description, and refers to it; the posttrans description for this "edition" is that the capsule has "pushout" applied to its "author" component. At this point these descriptions have been formed by simulating the effect of the transformation on the descriptions of the existing capsules, and no transformation of capsules has been done.

The second phase of the process is driven by the search for an existing capsule fitting any of the descriptions. It proceeds through the sequence of descriptions in order. If a capsule with an equivalent description to the current one is found, its identifier is used to replace all the instances of the old type-identifier, which are referred to in the current description's replacement set. The process is then complete. If no such capsule is found, the original is split and its new identifier replaces those in the replacement set. The search then continues within this newly created capsule for the next description in the sequence.
Where a description has no replacement set all applications of the capsule are replaced; but if there is also no existing capsule with a matching description the process continues within the original capsule without splitting it.

If the search has found no matching description at the end of the sequence of descriptions then the transformation is applied in the program to the original declaration of the last step, or to its split copy if there is a replacement set.

Even where a matching description has been found a small amount of transformation remains to be done to replacement set members (or to all applications of the capsule) that lie outside the textual bounds of the capsule. We refer to these constructs as "overhanging". Depending on the transformation, the program context of these type-identifiers (such as a variable declaration) may need to be transformed, and also some constructs associated with them (such as applications of the variable). Such piecemeal transformations apply parts of the whole transformation that are also applied in the full transformation of a capsule.

5.2.3.1 Capsule descriptions

A capsule's description depends on the history of transformations undergone by it and on the identity of the capsule it is ultimately derived from. The transformations include "inner" transformations: those applied to the capsule's inner capsules and selectively (i.e. to subsets of a type's instances) to bindable constructs it contains.
The following derivational properties of capsules are reflected in descriptions:

- the order of applying binding and inner transformations to different bindable constructs is insignificant;

- the order of applying inner transformations to different inner capsules is insignificant;

- the order of applying binding and inner transformations to the one bindable construct is significant;

- the order of applying structural transformations to a capsule is significant;

- the order of applying a structural transformation and any inner or binding transformation in the same capsule is sometimes significant.

In addition, a description of a capsule can exist before the capsule itself exists.

We use Xlan with two extensions to define descriptions with these properties. The extensions are a MAPPING type constructor and a SHARED modifier.

```plaintext
description =
  name: (own_id: (identifier | "program")
   environ: [SHARED description])
  hist: history;
```

with each capsule declaration is associated a description.

We use Xlan with two extensions to define descriptions with these properties. The extensions are a MAPPING type constructor and a SHARED modifier.
I derived; derived = basis: SHARED description derivation: (structural | bindings);

structural = structural_xform;

bindings = MAPPING (identifier | 'all')
  ONTO ('unchanged' | { xform: (binding_xform | inner)
    from_sharing: [sharegroup_identifier] } );

inner = description;

structural_xform = 'evert' | 'indices' | 'overlay' |
  'padout at' ('first' | 'last' | 'both ends') |
  'nullfornonex by range' '("(identifier | number) ")';

binding_xform = 'pushout' | 'multicopy' |
  'nullfornonex by structure' |
  'incorporate by' ('markers' | 'table').

With each capsule declaration is associated a description:

VAR descriptions = MAPPING capsuledec ONTO SHARED description;

SHARED allows a parse-tree to have more than one parent, but makes
ANCESTOR and CONTEXT operations on it meaningless. A MAPPING is an
updateable relation. It is updated by a SETMAP statement such as

SETMAP (descriptions, this_capsule, new_descript);

and the value of the relation for a particular (by identity) parse-
tree in its domain is given by an expression, such as

  descriptions (this_capsule).

The description of a capsule is either that it is an "original"
(i.e. present in the original program); or derived from some other
capsule by a structural transformation; or derived from some other
capsule by a number of binding or "inner" transformations on its con-
stituents. A "constituent" in this sense is any of the bindable or
scoping constructs (operations, bundles, capsules) defined in the capsule. The Xlan repetition constructor used for the binding transformations corresponds to the property that the order of binding transformations on one bindable construct is significant; the mapping constructor, to the property that transformations on different bindable constructs are independent. We choose to make the mappings complete by using the symbol "unchanged" for all untransformed constituents. The properties of "inner" transformations permit them to be treated like binding transformations.

For example, consider "edition" as previously defined, with two added variable declarations.

```
BUNDLE edition = [
  CAPSULE edition = (STATE:
    STRUCT (author: full name;
      POSSIBLY etalia: name_list;
      title: string);
    language = SELF FUNC: tongue;
    BEGIN ... END;
    ...);
  VAR ed1: edition $ edition;
]
```

We show a description with explicit selector labels for clarity. Round brackets and commas represent paired domain and range values in a mapping, and angle brackets contain all its pairs. If this capsule is in its original form its description is

```
(name: (own_id: "edition"
  environ: (shared description of bundle "edition")
  hist: "original").
```

Applying the binding transformation

```
pushout (author OF ed1)
```
produces the description

```
(name: (own_id: "unknown"
        environ: (shared description of bundle "edition"))
hist: (basis: (shared original edition description)
    derivation: < ('title', "unchanged")
    ('author', "pushout")
    ('etalia', "unchanged")
    ('language', "unchanged") > )).
```

Further applying a structural transformation, such as `evert (edl)`
will produce another description derived from this one:

```
(name: (own_id: "unknown"
        environ: (shared description of bundle "edition"))
hist: (shared transformed description above)
    derivation: "evert")).
```

The identifier "unknown" is used for descriptions that belong to a capsule as yet unknown.

It is clear from this example that all declarative levels also have descriptions. They are required as "environ" of some capsule descriptions. Their histories comprise only "inner" and unchanged derivations. The outermost program level has no description, being represented by a nonexistent "environ" in its constituents' descriptions.

Equivalence of descriptions is used in the process of applying a transformation which is described above. It depends on descriptions having equivalent histories and the same ultimate origin, as defined by the following Xlan function:
FUNCTION equivalent (dl, d2: description): boolean;
BEGIN
IF is remembered (dl, d2) THEN RETURN true
ELSE IF dl.hist IS original AND d2.hist IS original
THEN RETURN (dl.name.own id MATCHES d2.name.own_id
AND ((EXISTS dl.name.environ AND EXISTS d2.name.environ)
CAND equivalent (dl.name.environ QUA EXISTS,
  d2.name.environ QUA EXISTS)))
ELSE IF dl.hist IS derived AND d2.hist IS derived
THEN remember (dl, d2);
  IF equivalent (dl.hist QUA derived . basis,
    d2.hist QUA derived . basis)
AND equiv derivation (dl.hist QUA derived • derivation,
                   d2.hist QUA derived • derivation)
    THEN RETURN true
  ELSE forget (dl, d2);
    RETURN false
ENDIF
ELSE RETURN false
ENDIF ENDIF ENDIF
END.

The ultimate origin of a description is described by its
declaration's identifier and a particular environment of the program,
in turn described by a description of the same form. The program's
outermost declarative level has and needs no description.

Descriptions can be self-referencing by virtue of inner descrip-
tions having the outer as their environment, and possibly applying an
outer capsule in an "inner" transformation. The latter case makes it
necessary to use a "memory" in traversing connected descriptions to
decide equivalence; the technique is borrowed from the Algol 68
Report.

Descriptions with transformations must derive from the same
basis, if they are to be equivalent, and moreover must have equivalent
derivations from it. Derivations are equivalent only if they have the same structure (either "structural" or "bindings"). Structurally transformed derivations are equivalent if the same structural transformation has been applied; this is determined by textual matching. Derivations with binding and inner transformations are mappings; they are equivalent if and only if the range values (which are repetitions) for matching domain values have corresponding components which themselves either match textually or denote inner transformations with equivalent descriptions.

This form of description has the properties outlined above, and moreover requires only small amounts of maintenance when the corresponding program structures split and are transformed. It allows the process of applying transformations to search the program for an existing capsule completely equivalent to a transformation's result, without performing that transformation, because neither creating derived descriptions nor determining descriptions' equivalence refers to existing components of the program parse-tree. When a capsule is split one half keeps the original description, the other takes a copy, with shared basis and environment and copies through its history; the copy's inner constructs which have descriptions take copies of their originals', with the enclosing new descriptions for their environments. When a capsule is transformed it simultaneously updates its description to the value of the new "hypothetical" description which was unsuccessfully searched for; its inner constructs thereby have their environments maintained without further action.
5.2.4 Discussion

The motive for a system of descriptions of program constituents is to gain some understanding of the interactions between sequences of transformations applied to one program to achieve multiple representations. An additional practical motive is reducing the cost of applying transformations.

5.2.4.1 Necessity for capsule descriptions

In practical terms, multiplication of redundant, completely equivalent capsules cannot be avoided by specifying simultaneously all applications of a transformation to arguments of one type, as might be thought. Such a scheme would be inadequate because, as described earlier, applying a transformation may cause the splitting of types other than the one it transforms, and duplicates could still arise thereby. To avoid making any splits which would result in duplicating capsules we need to know which of these side-effect splits would have the same results from distinct arguments of the transformation. But this itself necessitates a scheme of capsule descriptions.

5.2.4.2 Overhanging transformations

The need to transform some "overhanging" applications of a capsule when substituting the identifier of an existing, completely equivalent capsule is an inevitable result of being able to transform primitive notions of the data spaces. It is not merely by choice that capsule interfaces and type behaviour are mutable when changing data
spaces; and therefrom results a leakage of changes within the capsule (whether effected by transformation or wholesale substitution) across its interface. The mutability exists because a primitive notion ("binding" in BSL’s case), which is the interfaces’ underlying mechanism, changes its characteristics across the language spectrum. Unlike the value manipulation operations, binding has no more primitive notions in terms of which it can be expressed to maintain a fixed interface. The overhanging transformations of the far side of the changed object generator, operation call, and binding assignment interfaces are the result.

5.2.4.3 Descriptions and classes of transformation

The system of descriptions clearly can accommodate any further transformations added to the set considered so far, provided they conform to the classification of being structural or binding. The classification is based on the comparison between pretrans and posttrans objects: a binding transformation preserves an object of the pretrans object’s type somewhere within the posttrans type, whereas a structural transformation does not. (Thus the two forms of nullfornonex have different classifications.) The mechanism for applying transformations therefore need consider the possibility of type-splitting only for its arguments’ environment parts; the prefix part of a binding transformation’s argument cannot cause a split.

The system of descriptions based on this classification has the properties required of it, as we have seen. But these properties are abstract ones which ignore all lesser differences between
transformations than their class. Properties of particular transformations are ignored, for instance:

- "inner" transformations are included in the form of binding transformations, but the order of applying any transformation to an inner capsule and a structural transformation to its enclosing capsule makes no difference (whereas in this scheme the descriptions are unequal);

- combinations of transformations producing completely equivalent results can produce very different descriptions; for instance, the effect of one transformation can be compensated for by a change in another's arguments, e.g. "pushout (x); evert (x,FOLLOW)" as against "evert (x); pushout (x)".

The solution of these problems would require more detailed descriptions and a more finely detailed model of the effects of transformations upon them. An extreme example of the specialization needed in such a case is the requirements of the incorporate transformation: where one component of some type has been incorporated, it would be desirable to avoid incorporating the operations of the component's type into the capsule for a second time when a second component of the same type is incorporated. The present mechanism contains no possibility of achieving this.

The elementary algebra of descriptions of data structure transformations presented here informally is thus seen to be incomplete. But it provides a simplifying framework in which to apply transformations, and is adequate to meet most of its practical aims.
5.3 Assignment compatibility between different representations

A major goal of this work is data independence. In the context of BSL and the transformation system this means

(i) choosing between alternative representations for one abstract type;

(ii) choosing different representations for different instances of a type;

(iii) maintaining all abstract type behaviour for all representations.

In particular this last includes "assignment" in abstract type behaviour as far as possible.

The transformation system allows different representations to be chosen. User-defined type behaviour is preserved by the transformations acting on its encapsulation in BSL capsules; implicitly defined behaviour is preserved by bundles and their generic value manipulation operations, allied to the transformations.

In a conventional simple language with name equivalence of types assignments between instances of different capsules, defining distinct types, are forbidden. Maclan has this property only for bindings, which are indeed strongly typed. (Although the binding assignment operation has disappeared from the language, the semantics of binding are demonstrated in parameter passing and the properties of references in particular.) But value assignment, equality and copying are not
primitives, and are defined as generic operations. Their parameters are less strongly typed and allow arguments to be of different types, constrained to belong to the same bundle.

For all types representing a common abstract type the value manipulation operations are preserved by definitions of generic operations in their common bundle. The \textit{vmos} call bundle interface operations in each representation type's capsule. The \textit{bios} of each capsule are automatically tailored to that capsule by the same transformations which cause it to differ from the abstract capsule, so that they have equivalent semantics for all representation capsules of the bundle. The \textit{vmos} therefore mean the same, in terms of the abstract type, for all combinations of argument types from their bundle.

5.3.1 Value manipulation and bundle interface operations

Every capsule is enclosed in a bundle before any other transformations take place. The bundle has the same identifier as the capsule. It contains: an abstract capsule, which is a textual copy of the original capsule with the symbol ABSTRACT replacing its identifier; another copy of the original capsule, enlarged to include bundle interface operations; and a group of generic definitions for the bundle's value manipulation operations. For sequence capsules the
form and number of the bios and the form of the vmos depend on only
two things: whether the components of the sequence of the abstract
capsule state are shareable, and whether the sequence is of fixed
length. The remaining variation is in the type and shareability of the
operations' yields and parameters, which share these attributes of the
components of the sequence.

5.3.1.1 Variable-length sequences with unshareable components

Value manipulation operations define top- and total-copy assign-
ment, equality, and copy generation by generic operations: ":=",
":-:", "TOPCOPY", "=", "=",
and "TOTALCOPY". The bundle interface
operations which the vmos call are readers and writers over the
abstract sequence, with the names "reread", "rewrite", "reader",
"total_writer", "top_writer", "more" and "cease_write". We can illus-
trate their interaction by the vmos and initial bios for a capsule
whose state is a variable-length sequence of unshareable components.

One of the vmos, for example, is

:= = PROC (left: dest FROM BUNDLE ANY;
right: source FROM BUNDLE ANY);
BEGIN
reread() OF right;
rewrite() OF left;
WHILE more() OF right DO
  total_writer (reader() OF right) OF left
ENDDO;
cease_write() OF left
END.

The initial bios are defined using an auxiliary traverser named
"auxil" which is added to the sequence's original set of traversers.
The type of value yielded by "reader" and accepted by "total_writer"
and "top_writer" is a reference to the base type of the sequence. (The call of "reader" is allowed as actual parameter to "total_writer" because all Maclan function yields have generated shareability.) A capsule originally defined by

CAPSULE stack = (STATE: SEQUENCE UPTO 20 OF eltmode TRAVERSED scanner; ...

for example, is enclosed in a bundle and itself becomes

CAPSULE stack = (STATE: SEQUENCE UPTO 20 OF eltmode TRAVERSED scanner, auxil;

reread = SELF PROC;
BEGIN RESET auxil END;

rewrite = SELF PROC;
BEGIN
IF NOT FINISHED scanner THEN error
("assignment to sequence with unfinished traverser")
ELSE RESET auxil ENDIF END;

more = SELF FUNC: boolean;
BEGIN more YIELD NOT FINISHED auxil END;

reader = SELF FUNC: REFTO (eltmode);
BEGIN
reader YIELD REFTO TOPCOPY ((# auxil #));
NEXT auxil END;

total writer = SELF PROC (arg: REFTO (eltmode));
BEGIN
IF IS EMPTY SELF THEN GROW AT FIRST WITH TOTALCOPY (FOLLOW (arg)) ELSE IF FINISHED auxil THEN GROW AFTER LAST WITH TOTALCOPY (FOLLOW (arg)) ELSE (# auxil #) := FOLLOW (arg);
NEXT auxil ENDIF END;

The bios are henceforth treated like any other operations of the capsule, being transformed and copied when the capsule is transformed and split. The user is forbidden to apply transformations explicitly to the bio interfaces; they therefore are immutable and in all capsules correspond to their calls in the vmos. The transformations of the bios' bodies, being meaning-preserving, leave them always implementing the functions shown here, whatever the representation of their capsule becomes.

It is also clear that the meanings of the assignment, comparison and copying operations applied here to individual components of the sequence are preserved, despite any possible transformations of their type, by the same vmo/bio mechanisms within the bundle which encloses the capsule defining their type. The definition of the vmos depends upon this essential property, which in turn requires that the capsules defining the types of the components of the distinct implementation
capsules themselves all be members of one bundle. This is the reason for the caveat mentioned above that the capsule defining the type of the sequence components should be declared outside the sequence capsule; for otherwise the splitting of the sequence capsule would produce distinct bundles (one in the original, another in the split copy) for the capsules defining the types of the sequence components. Such members of distinct bundles would be incompatible for the required value manipulation operations in terms of which the vmos of their owning capsules are expressed.

The other value manipulation operations are those for total-copy equality and generation, and top-copy assignment, equality and generation. Top-copy assignment is very similar to the foregoing total-copy assignment, with a difference only in the repeated statement. Where total-copy assignment had

\[
\text{total_writer (reader() OF right) OF left}
\]

top-copy assignment has

\[
\text{top_writer (reader() OF right) OF left.}
\]

A single "reader" serves both top- and total-copy operations by passing a top-copy of each component out of the capsule, which "total_writer" accepts and then total-copies. This gives the correct, recursive semantics to both top- and total-copy assignment. In both cases the "writer" updates existing components rather than simply replacing them, because although the components themselves are unshareable here, they may have shareable components at some remove by which the substitution could be detected.
The pairs of equality vmos and copy generation vmos for top- and total-copy are as similar as these two for assignment. Top-copy equality, for example, is:

\[
:- = \text{FUNC} \left( \text{left}: \text{lt FROM BUNDLE ANY}; \right.
\]
\[
\text{right}: \text{rt FROM BUNDLE ANY): boolean;}
\]
\[
\text{VAR eqlsofar: boolean := true;}
\]
\[
\text{BEGIN}
\]
\[
\text{reread() OF left; reread() OF right;}
\]
\[
\text{WHILE eqlsofar AND more() OF left}
\]
\[
\text{AND more() OF right DO}
\]
\[
eqlsofar :=
\]
\[
(FOLLOW \left( \text{reader() OF left} \right) :-: FOLLOW \left( \text{reader() OF right} \right)
\]
\[
\text{ENDDO;}
\]
\[
\text{:- YIELD eqlsofar AND}
\]
\[
\text{NOT (more() OF left OR more() OF right)}
\]
\[
\text{END.}
\]

As described in Chapter 3 the copy-generating vmos have a yield of generic type which takes its actual type from the context of the operation call. This type is used to declare a local variable to which the parameter is assigned, thereby effecting the copying operation via the corresponding assignment vmo. The yield of the operation is this variable, no further copying being necessary to make it acceptable in Maclan [section 3.3.4].

\[
\text{TOTALCOPY = FUNC (fr: source FROM BUNDLE ANY): dest FROM BUNDLE;}
\]
\[
\text{VAR result: dest;}
\]
\[
\text{BEGIN}
\]
\[
\text{result := fr;}
\]
\[
\text{TOTALCOPY YIELD result}
\]
\[
\text{END.}
\]
5.3.1.2 Shareable components

For sequences with shareable components the interfaces between vmos and bios differ from those for unshareable components in the type of the values passed out of the "reader" and into the "total_writer" and "top_writer" bios. The reference type gains a shareability qualification. Thus for example for a bundle with abstract state

```prolog
SEQUENCE OF amode SHAREABLE agroup
```

the type yielded by "reader" is

```prolog
REFTO (amode SHAREABLE agroup)
```

but the form of the definition of top-copy assignment is no different despite this change in type:

```prolog
:- = PROC (left: dest FROM BUNDLE ANY; right: source FROM BUNDLE ANY);
BEGIN
reread() OF right;
rewrite() OF left;
WHILE more() OF right DO
  top_write (reader() OF right) OF left
ENDDO;
cease_write() OF left
END.
```

The differences in treatment of the values passed from one capsule to another appear in the "writer" bios. "Total_writer" again has "FOLLOW (arg)" where the unshareable component version has it.

"Top_writer" also uses FOLLOW but expresses the semantics of top-copying: where components are shareable then they are shared. Thus:
5.3.1.3 Fixed-length sequences

The operations for fixed-length sequences can discard the complex control of the "reader" generator and the need for a post-"writer" removal of excess components ("cease_write"): both source and destination necessarily have the same known length. The "writer" bios also need not allow for an empty or short destination. Thus the bios are reduced to "reread", "rewrite", "reader", "total_writer", and "top_writer". All except "total_writer" and "top_writer" are the same as for the corresponding cases, with shareable or unshareable components, in variable-length sequences. For a "SEQUENCE 20 OF amode SHAREABLE agroup", for instance,

```
top_writer = SELF PROC (arg: REFTO (amode SHAREABLE agroup));
BEGIN
  IF IS EMPTY SELF
    THEN GROW AT FIRST WITH FOLLOW (arg)
  ELSE IF FINISHED auxil
    THEN GROW AFTER LAST WITH FOLLOW (arg)
  ELSE (# auxil #) <- arg;
  NEXT auxil
END IF ENDIF
END.
```

and the simpler control structure is exemplified by

```
top_writer = SELF PROC (arg: REFTO (amode SHAREABLE agroup));
BEGIN
  (# auxil #) <- FOLLOW (arg);
  NEXT auxil
END
```
:= = SELF PROC (left: dest FROM BUNDLE ANY;
right: source FROM BUNDLE ANY);
VAR i: 1 .. 20;
BEGIN
reread() OF right; rewrite() OF left;
i := 1;
WHILE i <= 20 DO
  total_writer (reader() OF right) OF left;
i := i + 1
ENDDO
END.

5.3.2 Discussion

The goal of preserving assignment compatibility with multi-
representation is thus achieved within the framework of transforma-
tions developed for single representations. The language mechanisms
of BSL have been expanded beyond those needed for single representa-
tions to include a non-primitive formulation of the operations of
value assignment, comparison and copy generation; a form of generic
typing for parameters; and explicit treatment of the overloading of
these operations. These are a considerable extension of the power of
the language at the machine-oriented end of its spectrum, compared to
other languages with a similar data space. The vmo/bio mechanism is
however cheaper than alternatives which use less powerful language
features.

The mechanism adopted has the important property of being intel-
lectually manageable. It separates related capsules cleanly: they
communicate only via the immutable vmos, while each capsule describes
in its bios only its relationship to the original abstract type, and
not to the other capsules.
There are some limitations to this mechanism. Struct types are not handled as cleanly as sequences, as we shall see below, and total-copy operations for sequence types are limited to objects with no cycles in their component object graph. Trees cannot be handled at all.

5.3.2.1 Alternative mechanisms

Alternatives to the vmo/bio mechanism are either inadequate in semantic power or increase the size of the program excessively.

Maclan's generic operations could be avoided by adopting an open-form solution to value manipulation operations. This would expand each value assignment, comparison or generation in place, so that it could be transformed with its constituents' capsules as an overhanging application. The treatment of struct-typed objects in this formulation is little harder than that of sequences, unlike our closed form [section 5.3.2.6]. But this solution is inadequate in that it cannot be applied to any recursively constructed type (whereas the closed-form solution adopted forbids only actual cycles in the object/component graph in total-copy operations). For recursive types some closed-form operation is evidently necessary. An open-form solution is also undesirable because of the extreme expansion of the program that it would cause, with the textual equivalent of the bodies of a vmo and a number of bios replacing each vmo call. Transformation of programs is only attractive if the result is intellectually manageable, and it must be somewhat better than such an element-by-element expansion into lower order primitives, as argued in Chapter 4.
Another possible solution is to have a set of vmos for each pair of representation capsules in the bundle. This would need neither generic parameter mechanism nor bundle interface operations. Overloading could be replaced by generating a distinct name for each pair's operations. But the number of operation definitions would depend on the square of the number of capsules in the bundle, rather than being linear as it is with the use of generics. Splitting a capsule would be more complex as it would mean also splitting every one of the vmo definitions in which the type appeared, their number rising with the number of other capsules.

5.3.2.2 Variant forms of operations

Some less important points than those above concern the forms chosen for the various operations. In the vmos and bios we have made distinctions between abstract capsules with shareable and with unshareable components, and between those of fixed length and those of variable length. Both distinctions are essential, the former to maintain the correct semantics for top-copy operations, the latter because GROW and SHRINK cannot be applied to fixed-length sequences. The accompanying reduction in the number of bios for fixed-length sequences is incidental, and made more noticeable by the arbitrary choice of controlling the repetition by an index variable rather than by using calls of "more".

A further structural variant that might be distinguished is that of "no shared descendent components" which is used in the auxiliary transformations [section 4.2.2.2, no. 2]. This distinction is not
strictly necessary since the semantics of unshared components already adequately covers this case. The only gain would be to allow a quite different formulation of the operations whereby "rewrite" would perform a MAKE_EMPTY operation and "writer" successive GROWS AFTER the "auxil" traverser, kept always at LAST; no "cease_write" would be needed. This might be more efficient for certain representations: e.g. a linked list representation has a relatively expensive implementation of LAST, an operation used heavily in the standard variable-length sequence vmos.

The possibility of such variant forms demonstrates the limits of the power of this transformational system, in that they cannot be achieved by automatic auxiliary transformation nor by explicitly applied major transformation. In the spirit of the system presented in this thesis the required transformations should be available to be applied explicitly; but there would then be a need to include vmos and their generic parameters in transformations generally, and to make what amount to structural transformations of particular procedural, rather than data, structures of a program.

5.3.2.3 Choice of language for vmos and bios

The essential properties of the vmos and bios that determine the choice of languages for their expression are that vmos are immutable and bios subject to subsequent transformation. The vmos therefore must utilize only the machine-oriented data space of Maclan, being that of the final concrete program, from the time of their introduction. Thus although the overload operation definitions constitute an
operational semantics for the value assignment, equality and copying operations, they must be expressed over a concrete data space and hence use reference values rather than the more abstract possibility of mutable bindings.

For the bios, on the other hand, it is desirable that their initial definitions be given as abstract a form as possible. It is feasible to do this because they are subsequently transformed along with their owning capsule, so that their initial form need not be Maclan-acceptable. It is desirable to have an abstract form to avoid constraining the possible representations of the capsule, still in an abstract form at the time that the bios are created. Using indexing rather than traversing, for instance, is avoided.

5.3.2.4 Particular transformations and the bios

Two transformations in particular might raise some concern because of their possible effects on the underlying assumptions of the vmo and bio mechanism. These are incorporate, because it deletes all explicit reference to a component type; and multicopy, which if applied to the components of a sequence affects their shareability and binding. The concern arises because the bundle interface is formulated in terms of the component shareability and type.

Incorporate is incompatible with this mechanism. Its components cease to exist as objects in their own right, and therefore the vmo mechanism for their own type's bundle cannot continue to operate on them. Neither can that of their owner's type. By their very
existence its bios are sufficient to forbid the transformation, by
violating its restrictions: they apply TOPCOPY and TOTALCOPY to the
components, and the vmos call the components' SELF operations whose
interfaces must not be transformed.

Multicopying the sequence's components, on the other hand,
presents no problems. The transformation causes the component type of
the posttrans capsule to lose its shareability, while the parameter
type of its "writer" operations and the yield of its "reader" naturally retain theirs. But the normal effect of the transformation is
adequate to deal with this: a binding to or from a shareable name,
whether it is to one within a reference value with shareable qualifi-
cation (as in the yield assignment of "reader") or from one obtained
by FOLLOWing such a reference value (as in the GROW and binding
assignment of "writer" bios), is transformed to TOPCOPY the source.
Auxiliary transformations may then apply to the binding assignment to
produce an equivalent top-copy assignment. Thus the statements within
a "writer" are transformed:

\[
\begin{align*}
\text{GROW ... WITH FOLLOW (arg);} \\
(# \text{ auxil } #) & \leftarrow \text{FOLLOW (arg);} \\
\Rightarrow \text{GROW ... WITH TOPCOPY (FOLLOW (arg));} \\
(# \text{ auxil } #) & \leftarrow \text{FOLLOW (arg);} \\
\end{align*}
\]

and so is the yield of "reader":

\[
\begin{align*}
\text{reader YIELD REFTO (# auxil #)} \\
\Rightarrow \text{reader YIELD REFTO TOPCOPY ((# auxil #))}. \\
\end{align*}
\]

Note that although multicopy cannot be applied if there is any value
assignment of its argument (or its ancestors) in the pretrans program,
the assignment VMOS remain necessary because of their use in defining
the copy-generating operations, whose use is permitted.

5.3.2.5 New object generation

The object generator NEW has been largely ignored in the preceding descriptions of structural transformations and type-splitting. Unlike the copying object generators, it is not included in the overload definitions of VMOS, although it clearly has as much to do with the generation of values as they do. It is evidently necessary for each type, i.e. each capsule, and not merely each bundle, to have some means of generating values because a new object must have some particular representation.

The transformation of NEW generators for a capsule is therefore done at the same time as the capsule, by mechanisms already partly described. A binding transformation such as pushout applied to a component of the capsule state causes all corresponding actual parameters to be correctly transformed, i.e. all NEW generators for this type have their parameter list transformed to match the transformed STATE declaration. A structural transformation transforms the generator as a whole.

Where there is type-splitting, selected object generators are included in the transformation replacement sets with all other sources of bindings to the transformation’s arguments. The overhanging transformations therefore include these generators as necessary.
Where an object generator is not the source of a binding it is transformed only with the capsule whose type name it has. If it is the source for a copying operation the generic vmos work as for any other type.

5.3.2.6 Trees and structs

Vmos and bios can be defined for capsules with structs in a similar fashion to those with sequences. We maintain the notion of passing one component object at a time from a "reader" to a "writer" kept in step with it, but the types and shareabilities of successive objects may differ from each other, making the interface types more complex.

For structs a reader and writer are locked in phase by using a state-indicating component, analogous to the "auxil" traverser, added to the state definition. The value of this component indicates that a particular component of the abstract struct definition is being passed, and the "reader" and "writer" vmos consist largely of a case-statement to select the appropriate component.

The interface type presents more problems. Because the interface between "reader" and "writer" must handle objects of all the types among the struct's components, as well as indicating their possible nonexistence, the type of this interface is an NREFTO qualified by an overlay union of all component types with appropriate shareabilities. The case-branches in the initial bios reflect the individual shareability and possibility of their corresponding component. Thus for
example the bundle for a capsule with abstract state

```plaintext
STRUCT (a: amode SHAREABLE agroup;  
b: bb;  
POSSIBLY c: cmode)
```

has capsules with bios such as:

```plaintext
reader = SELF FUNC:  
NREFTO (OVERLAY (amode SHAREABLE agroup,  
bb,  
cmode));
BEGIN  
CASE auxil OF  
WHEN xa THEN reader YIELD REFTO (a) ENDWHEN,  
WHEN xc THEN WITH c DISCRIMINATE  
WHEN exc: TOPCOPY EXISTS  
THEN reader YIELD REFTO (exc) ENDWHEN  
OTHERWISE reader YIELD NIL . . . .
```

Tree-structured capsules need only a simple interface type like sequence capsules, but their structure makes bios impossible. The BSL mechanisms are inadequate to maintain a pair of traversers locked together over two tree objects. A simple traverser cannot be made to traverse the whole of a tree because it cannot back up, nor can its states be saved and restored. The traverser parameter is able to traverse the whole tree, but the lack of coroutine capabilities in BSL means that component values cannot be passed out of and into the interiors of the nests of recursive calls which are needed to maintain the states of the traverser parameters. Assignment compatibility for multiple representation therefore cannot be achieved for tree capsules with the language framework studied in this thesis.
The powerful and clean mechanisms described in this chapter make possible multiple representations of a single abstraction within one program, preserving assignment compatibility between different representations of abstractions originally defined with sequences and structs. The mechanism of the bundle, its overloaded value manipulation operations, its abstract capsule and encapsulations of each representation with associated bundle interface operations, is textually strong. It provides a clean formulation both of the separation between an abstract data type and each of its representations (as distinct capsules) and of the logical and semantically significant association of the different representations of one abstraction (by their membership of one bundle). The common membership of capsules in a bundle is closely coupled to the assignment compatibility of their instances through the overloading of the value manipulation operations at the bundle level. The cleanness of this formulation under the application of transformations is ensured by the general property of preservation of operation interfaces, so that those transformations applied to derive a particular representation maintain the interfaces between the capsule which defines that representation, the bundle to which this capsule belongs, and (through the operations of the bundle) other members of the bundle.
6.1 Bounded stack definition

6.2 Bundle form

6.3 Linked list
6.3.1 Eversion
6.3.2 Pushout elements
6.3.3 Nullfornonex
6.3.4 Evaluation

6.4 Vector with top
6.4.1 Padout at start
6.4.2 Indices for traversers
6.4.3 Nullfornonex
6.4.4 Evaluation

6.5 Alternative transformation paths

6.6 Discussion
This chapter draws together the separate threads of capsule definition, bundle creation, type-splitting, the application of transformations, and the various transformations themselves, by looking at an extended example. A capsule defining a bounded stack is taken through the complete development of two different representations: the "conventional" representation as a vector with an associated index indicating the current top of the stack, and a representation as a linear linked list where the head of the list is the "top" element of the stack.

The bounded stack has become a standard example of the data abstraction literature. It is sufficient to demonstrate the effects of the elements of the program development system built up in the preceding chapters. It also has the advantage of brevity and simplicity, making it feasible to present here; even a small example grows quickly in size because the application of transformations tends to increase the amount of text in a capsule definition as the language used moves towards the machine end of the spectrum, and because the bulky bundle interface operations are added. We shall therefore show only parts of the bounded stack bundle at each stage of the transformation sequence.

The transformations are hand-simulated since not all have been implemented. Points of interest in programs are marked by comments such as "(* A *)"; lines changed from the preceding program are marked on the left-hand margin with "]."
6.1 Bounded stack definition

There are many possible definitions of the abstraction "a bounded stack of integer values" as a Koala capsule. Here we shall use a capsule whose state is a bounded sequence, where "push" and "pop" are defined to grow and shrink the sequence at its first component.

THE INITIAL KOALA DEFINITION

CAPSULE stack = (STATE: SEQUENCE UPTO 25 OF integer;
   pop = SELF FUNC: integer;
   BEGIN
   pop YIELD TOTALCOPY ((#FIRST#));
   SHRINK AT FIRST
   END;
   push = SELF PROC (n: integer);
   BEGIN
   GROW AT FIRST WITH TOTALCOPY (n)
   END);

VAR a: stack <- NEW stack (16,17,18);
VAR b: stack <- NEW stack (10,11);
VAR c: stack <- NEW stack (-5,-6);
BEGIN
a := b;
IF a = b
   THEN a.push(33) ENDIF
END.

By choice, no explicit notice is taken of stack overflow and underflow; these will be detected as erroneous sequence operations, in terms of the sequence growing beyond its declared bounds and attempting to access the first component of an empty sequence.
6.2 Bundle form

Forming the bundle which will encompass all derivatives of this capsule is a necessary preliminary to applying any transformations. The internal detail of the capsule is correspondingly expanded to include the bundle interface operations; they take the form appropriate to a bundle whose abstract state is a variable-length sequence of unshareable components.

BUNDLE FORMATION

BUNDLE stack [ (* vmos as in section 5.3.1.1 *)

... CAPSULE ABSTRACT = (STATE: SEQUENCE UPTO 25 OF integer;

    pop = SELF FUNC: integer;
    BEGIN
    pop YIELD TOTALCOPY ((# FIRST #));
    SHRINK AT FIRST
    END;

    push = SELF PROC (n: integer);
    BEGIN
    GROW AT FIRST WITH TOTALCOPY (n)
    END);

CAPSULE stack = (STATE: SEQUENCE UPTO 25 OF integer

] TRAVERSED auxil;

pop = SELF FUNC: integer;
BEGIN
pop YIELD TOTALCOPY ((# FIRST #));
SHRINK AT FIRST
END;

push = SELF PROC (n: integer);
BEGIN
GROW AT FIRST WITH TOTALCOPY (n)
END);
reader = SELF FUNC: REFTO (integer);
BEGIN
reader YIELD REFTO TOPCOPY ((# auxil #));
NEXT auxil
END (* reader *);

total writer = SELF PROC (arg: REFTO (integer));
BEGIN
IF IS EMPTY SELF
THEN GROW AT FIRST WITH TOTALCOPY (FOLLOW (arg))
ELSE IF FINISHED auxil
THEN GROW AFTER LAST WITH TOTALCOPY (FOLLOW (arg));
ELSE (# auxil #) := FOLLOW (arg);
NEXT auxil
ENDIF ENDIF;
END;

reset = SELF PROC;
BEGIN RESET auxil END;

more = SELF FUNC: boolean;
BEGIN more YIELD NOT FINISHED auxil END;

cease write = SELF PROC; ...;
]

VAR a: stack $ stack <- NEW stack $ stack (16,17,18);
VAR b: stack $ stack <- NEW stack $ stack (10,11);
VAR c: stack $ stack <- NEW stack $ stack (-5,-6);
BEGIN
a := b;
IF a = b
THEN a.push(33) ENDIF
END.

It can be seen that an auxiliary transformation applies immediately in the bios. In this case the sequence components have no shareable components at any remove (since they are the primitive type "integer") and hence the TOPCOPY operation in "reader" will be transformed to TOTALCOPY by the automatic auxiliary transformations.

In the succeeding steps of the example below, the application of any automatic transformations is indicated in the step that produces their triggering conditions by writing the "raw form" (produced by the
explicit transformative step) in a comment preceding the result of the auxiliary. In this fashion

reader YIELD REFTO (* topcopy ( (# auxil # )) *)
TOTALCOPY ( (# auxil # ))

would be written in place of the corresponding statement above, and only in subsequent programs would this be collapsed to the simpler form

reader YIELD REFTO TOTALCOPY ( (# auxil # )).

The binding assignments in the variable declarations are also the subject of auxiliary transformation, since their left-hand sides are (i) always bound and unshareable (auxiliary 1) and (ii) have no shared descendent components (auxiliary 2). Each becomes a total-copy assignment.

So far no data structure transformation has been applied. We shall proceed to develop different representations for the instances of the "stack" abstraction that are bound to the variables "a", "b", and "c", which entails splitting the "stack" capsule into distinct capsules to define these representations.

6.3 Linked list

The specification of the transformations to represent the objects bound to "a" as linked lists is as follows:
evert (a);
(which splits "stack" and everts the resulting "stack2");

pushout (succ OF stack.stack2.element,
          pred OF stack.stack2.edgelem,
          curr OF stack.stack2.edgelem,
          seq OF stack.stack2);

The last step must also be selective, since not all instances of
REFTO (element SHAREABLE eltgroup) have POSSIBLY:

nullfornonex by range (NIL) (stack.stack2.seq,
                          stack.stack2.element.succ,
                          stack.stack2.element.pred,
                          stack.stack2.edgelem.curr).

No type-splitting occurs in this instance, even though nullfornonex by
range is a structural transformation, because its arguments are of a
simple type not defined by a capsule.

6.3.1 Eversion

The transformation applied by "evert (a)" splits the "stack" cap-

sule, since "a" refers to only some instances of "stack" and there is

no other capsule in the bundle which has a description equivalent to

the one which would result from everting the "stack" capsule. The

full transformation is then applied to the newly created "stack2", and

overhanging applications of "a" are also transformed. Evert adds to

the capsule a type "edgelem" for traversers of the sequence, and vari-

ous operations for some of the primitive operations in closed form:

"index", "compute_make_empty", "compute_size", etc.
SPLIT AND EVERT

BUNDLE stack [  
(* vmos unchanged *)
CAPSULE ABSTRACT = (* unchanged *);

CAPSULE stack = (STATE: SEQUENCE UPTO 25 OF integer
TRAVESED auxil: ... );

] CAPSULE stack2 = (STATE: STRUCT (  
POSSIBLY seq: element SHAREABLE eltgroup;
auxil: edgelem);
CAPSULE element = (STATE: STRUCT (  
elem: integer;
POSSIBLY succ: element SHAREABLE eltgroup));

CAPSULE edgelem = (STATE: STRUCT (  
POSSIBLY curr: element SHAREABLE eltgroup;
POSSIBLY pred: element SHAREABLE eltgroup));

] SHAREGROUP eltgroup OF element;

pop = SELF FUNC : integer;
BEGIN
pop YIELD TOTALCOPY (elem OF seq);
(* shrink at first *)
WITH seq DISCRIM
  WHEN exseq: TOPCOPY EXISTS
  THEN seq <- succ OF exseq
  ENDDISCRIM
  OTHERWISE error ("shrink - finished traverser")
ENDDISCRIM
END;

push = SELF PROC (n: integer);
BEGIN
  (* grow at first with totalcopy (n) *)
  BEGIN
    VAR newel: element SHAREABLE eltgroup;
    elem OF newel (* <- totalcopy (n) *)
    := n;
    succ OF newel <- seq;
    seq <- newel
  END
END);

reader = SELF FUNC: REFTO (integer);
BEGIN
  reader YIELD REFTO (* topcopy *)
  TOTALCOPY (elem OF curr OF auxil);
(* next auxil *)
WITH curr OF auxil DISCRIM
WHEN excurr: EXISTS SHAREABLE eltgroup
THEN
  pred OF auxil <- excurr;
  curr OF auxil <- succ OF excurr
ENDWHEN
OTHERWISE
  error ("advancing finished traverser")
ENDDISCRIM
END (* reader *);

total writer = SELF PROC (arg: REFTO (integer));
BEGIN
IF compute_is_empty ()
  THEN (* grow at first - as in push *) ... ENDIF
ELSE IF (* finished auxil *) NOT EXISTS curr OF auxil
  THEN (* grow after last with totalcopy (follow (arg)) *)
    WITH curr OF compute_last() DISCRIM
    WHEN excurr: EXISTS SHAREABLE eltgroup
      THEN
        BEGIN VAR newel: element SHAREABLE eltgroup;
          elem OF newel := FOLLOW (arg);
        END
      ENDWHEN
    OTHERWISE
      error ("growth - finished traverser")
    ENDDISCRIM
  ELSE(* (# auxil #) := *)
    elem OF curr OF auxil := FOLLOW (arg);
  ENDIF
END (* total writer *);

reread = SELF PROC;
BEGIN (* reset auxil *) BEGIN
  curr OF auxil <- seq;
  pred OF auxil <- NONEX
END END;

more = SELF FUNC: boolean;
BEGIN (* not finished auxil *)
  more YIELD (* not not exists *) EXISTS curr OF auxil
END;
index = SELF FUNC (indexpr: integer): edgelem;
VAR counted: integer (* <- *) := 0;
VAR indextrav: edgelem;
BEGIN
curr OF indextrav <- seq;
pred OF indextrav <- NONEX;
IF indexpr < 1
THEN error ("illegal non-positive index")
ELSE
WHILE counted < (indexpr - 1)
AND EXISTS curr OF indextrav
DO counted := counted + 1;
pred OF indextrav
<- curr OF indextrav;
curr OF indextrav
<- succ OF curr OF indextrav
ENDDO;
IF NOT EXISTS curr OF indextrav
THEN error ("illegal index - too large")
ELSE index YIELD TOPCOPY (indextrav)
ENDIF ENDIF
END (*index*);
compute_last = SELF FUNC: edgelem;
BEGIN ... END;
compute_make_empty = SELF PROC;
BEGIN seq <- NONEX END;
compute_is_empty = SELF FUNC: boolean;
BEGIN compute_is_empty YIELD NOT EXISTS seq END;
compute_size = SELF FUNC: integer;
BEGIN ... END );
VAR a: stack $ stack2 (* <- *) := NEW stack $ stack (16,17,18);
VAR b: stack $ stack (* <- *) := NEW stack $ stack (10,11);
VAR c: stack $ stack (* <- *) := NEW stack $ stack (-5,-6);
BEGIN
a := b;
IF a = b
THEN a.push(33) ENDIF
END.

Despite the fact that "a" and "b" now have the distinct types "stack" and "stack2", the total-copying assignment "a := b" and comparison "a = b" remain legal because the types involved are members of the one bundle. They are calls of the bundle's vmos, which in turn
call the capsule's bios. The abstract meaning and interface of the bios of stack2 are seen to be unchanged by this transformation, as is expected for any transformation.

Note the form chosen to express assignment of a new value to instances of both "edgelem" type ("auxil", at point B) and "element" type ("newel", at point A). BSL permits the assignment to be expressed in a single statement: e.g.

(a) newel :- NEW element (TOTALCOPY (n), seq);

but we have chosen an expanded form:

(b) elem OF newel := n;
    succ OF newel <- seq.

The notational convenience of the first form would normally be preferred in a high-level language program for readability. It is relatively brief and clearly states an intention to update every component of the object bound to "newel". At this high level "NEW element(...)" is regarded simply as a denotation for a value of a structured type.

There are several arguments against the use of this form here. The first is that we intend to move the program towards the machine-oriented end of the spectrum. We expect the machine to implement the program by executing exactly the actions it specifies: the freedom which the typical high-level language acceptor has, to perform "any action or combination which causes the same effect" [Algol 68, 2.1.4.1(a)], is deliberately not given to a machine-oriented language. In this case the actions of two selective value-updating assignments (b) are simpler than those implied by (a), namely applying to the storage allocator for a new "edgelem" object; binding particular
values to each component; copying each component value to "auxil"; and discarding the newly created object.

The second reason for preferring the expanded form of the construction is that it is simpler when transformed further. The expressive complexity of the posttrans program affects how easily the programmer can understand it and decide where to apply further transformations, and also determines the transformational power needed to produce intelligent and intelligible results. It can be seen that the expanded form is simpler in this sense. A first instance is the auxiliary transformation which immediately produces "elem OF newel := n" from "elem OF newel <- TOTALCOPY (n)" above. Again, after the further transformation pushout is applied to both "seq" and "succ OF newel":

(unexpanded)
newel :- NEW element (TOTALCOPY (n),
    YIELDOF (WITH seq DISCRIM
        WHEN exseq: TOTALCOPY EXISTS
            THEN YIELD TOTALCOPY (exseq) ENDWHEN,
        OTHERWISE YIELD NONEX ENDDISCRIM));

(expanded)
elem OF newel := n;
WITH seq DISCRIM
    WHEN exseq: TOTALCOPY EXISTS
        THEN succ OF newel <- TOTALCOPY(exseq) ENDWHEN
    OTHERWISE succ OF newel <- NONEX ENDDISCRIM.

The YIELDOF construction in the single assignment form adds an extra layer of complexity for both human understanding and the program transformer, which must carry around the context of the YIELDOF when analyzing the branches of the discrimination it contains, for these to be further transformed sensibly.
Previously, where such a simpler equivalent to some construction was needed, we have used an auxiliary transformation to produce it. However, this substitution is in general too difficult without data flow analysis, because the unexpanded form expresses what is in effect a parallel assignment to the fields of the object bound to the left-hand side, as exemplified by

\[ \text{auxil} \leftarrow \text{NEW edgelem (pred OF auxil, curr OF auxil)} \].

Against the expansion of such assignments it can be argued that further transformations could be written that might take advantage of the knowledge that the multiple assignments are in fact inseparable. For instance, with a "packed" representation of objects with several components [Pascal, Ada] the simple and efficient method for total-copy assignment is to copy all components in a single operation, whereas assigning sequentially requires unpacking, preserving the other components, and reconstructing the partially updated packed value for each component assignment. Such transformations are however beyond the scope of this system.

The choice of substituting open or closed forms for the more complex operations here (for instance growth, shrinkage, advance) is not an arbitrary one. In the case of GROW and SHRINK Koala requires that the state of all traversers, other than any explicit traverser used in the specifying expression, be finished. The posttrans form of these statements must make this check for all the object's traversers, other than the one indicating the position for the statement. Because a closed form would need the identity of this traverser in order to exclude it from checking, and this implies an additional sharegroup,
we use an open form in which the transformation makes the decision on identities and inserts appropriate checking statements. An added sharegroup is undesirable because shareable variables are further from the machine data space and need additional work for their representation.

In this instance no traverser checking need be done, since the only traverser is "auxil" which has special status and is excluded from this requirement. To keep the transformation relatively simple this case is not distinguished from the general. Moreover the closed form is not the simpler form when further transformations are to be applied: where different actual parameters may be transformed differently the open form may be transformed to reflect this, but the closed form must always enforce a common type of parameter on all its calls, and enforces the common binding convention of passing parameters.

6.3.2 Pushout elements

The pushout transformations are specified simultaneously in this example, but could equally well have been sequential. As implemented, pushout has exactly the same effect on the program text either way.

Pushout introduces references and operations to create and follow them. Possibility is preserved, and where a reference is possibly unbound it must be discriminated before it is followed.
PUSHOUT ALL ELEMENTS OF STACK2

BUNDLE stack [
(* vmos and abstract unchanged *)
CAPSULE stack = ( ... );
]

CAPSULE stack2 = (STATE: STRUCT (POSSIBLY seq: REFTO (element SHAREABLE eltgroup); auxil: edgelem);
CAPSULE element = (STATE: STRUCT (elem: integer;
POSSIBLY succ:
REFTO (element SHAREABLE eltgroup)));

CAPSULE edgelem = (STATE: STRUCT (POSSIBLY curr: REFTO (element SHAREABLE eltgroup);
POSSIBLY pred: REFTO (element SHAREABLE eltgroup)));
SHAREGROUP eltgroup OF element;

pop = SELF FUNC: integer;
BEGIN
| pop YIELD TOTALCOPY (* elem OF seq *)
| (YIELDOF (WITH seq DISCRIM
| WHEN exseq : TOTALCOPY EXISTS THEN
| YIELD FOLLOW (exseq)
| ENDDISCRIM)
| OTHERWISE error ("illegal follow")
| ENDDISCRIM);
(* shrink at first *)
WITH seq DISCRIM
| WHEN exseq: TOTALCOPY EXISTS
| THEN (* seq <- succ OF exseq *)
| WITH succ OF FOLLOW(exseq) DISCRIM
| WHEN exsucc : TOTALCOPY EXISTS THEN
| seq <- (* REFTO FOLLOW (exsucc) *)
| TOTALCOPY(exsucc)
| ENDDISCRIM
| OTHERWISE seq <- NONEX
| ENDDISCRIM
ENDWHEN
 OTHERWISE error ("shrink - finished traverser")
ENDDISCRIM
END;

push = SELF PROC (n: integer);
BEGIN
(* grow at first with totalcopy (n) *)
BEGIN
VAR newel: element SHAREABLE eltgroup;
elem OF newel := n;
(* succ OF newel <- seq *)
WITH seq DISCRIM
WHEN exseq : TOTALCOPY EXISTS THEN
  succ OF newel <-
  (* REFTO FOLLOW(exseq) *)
  TOTALCOPY(exseq)
ENDWHEN
OTHERWISE succ OF newel <- NONEX
ENDDISCRIM;
seq <- REFTO newel
END END;

reader = SELF FUNC: REFTO (integer);
BEGIN
  reader YIELD REFTO TOTALCOPY (elem OF YIELDOF (WITH curr OF auxil DISCRIM
    WHEN excurr: TOTALCOPY EXISTS THEN
      YIELD FOLLOW(excurr) ENDWHEN
    OTHERWISE error ("illegal follow")
    ENDDISCRIM));
  (* next auxil *)
  WITH curr OF auxil DISCRIM
    WHEN excurr: TOTALCOPY EXISTS THEN
      pred OF auxil <- (* REFTO FOLLOW(excurr) *)
      TOTALCOPY (excurr);
    WITH succ OF FOLLOW(excurr) DISCRIM
      WHEN exsucc: TOTALCOPY EXISTS THEN
        curr OF auxil <- (* REFTO FOLLOW(exsucc) *)
        TOTALCOPY (exsucc)
      ENDWHEN
    OTHERWISE curr OF auxil <- NONEX
    ENDDISCRIM
  ENDWHEN
Otherwise error ("advancing finished traverser")
END DISCRIM END (* reader *);

total writer = SELF PROC (arg: REFTO (integer));
BEGIN ...
  (* succ OF newel <- succ OF excurr *)
  WITH succ OF FOLLOW(excurr) DISCRIM
  WHEN exsucc: TOTALCOPY EXISTS THEN
    succ OF newel <-
    TOTALCOPY(exsucc)
  ENDWHEN
  OTHERWISE succ OF newel <- NONEX
  ENDDISCRIM;
  (* succ OF excurr <- newel *)
  succ OF FOLLOW(excurr) <- REFTO newel ...
END (* total_writer *);
reread = SELF PROC;
BEGIN BEGIN
  (* curr OF auxil <- seq *)
  WITH seq DISCRIM
  WHEN exseq: TOTALCOPY EXISTS THEN
    curr OF auxil <- TOTALCOPY (exseq)
  ENDWHEN
  OTHERWISE curr OF auxil <- NONEX
  ENDDISCRIM;
  pred OF auxil <- NONEX
END END;

more = SELF FUNC: boolean;
BEGIN more YIELD EXISTS curr OF auxil END;

index = SELF FUNC (indexpr: integer): edgelem;
BEGIN ... END;

compute_make_empty = SELF PROC;
BEGIN seq <- NONEX END;

(* other compute_ operations *)
)

VAR a: stack $ stack2 := NEW stack $ stack (16,17,18);
VAR b: stack $ stack := NEW stack $ stack (10,11);
VAR c: stack $ stack := NEW stack $ stack (-5,-6);
BEGIN
a := b;
IF a = b
  THEN a.push(33) ENDIF
END.

No further type-splitting is involved in this step, since the
transformation is specified via the capsule identifier "element".

Pushout is independent of the order in which its arguments are
transformed. At point H (previously "succ OF newel <- succ OF
excurr") it discovers and treats separately the left- and right-hand
sides of the assignment. The order of transforming the two sides
makes no difference to the resulting form of the statement, just as on
the larger scale neither does the order of applying the transformation
to any of its arguments. However the transformation does not contain
the analysis of the special case where both left- and right-hand sides are being pushed out simultaneously, and therefore does not transform this case into the more succinct form

\[
\text{succ OF newel} \leftarrow \text{TOTALCOPY (succ OF excurr)}.
\]

(Neither this case nor the discriminating form given in the example can be reduced to a total-copying assignment at this stage because the left-hand side (and in this case the right also) are possibly unbound.)

Pushout introduces TOTALCOPY into the binding of the EXISTS-case in both pre-existing discriminations and those it introduces (point F; points G and H). Where the pretrans discrimination branch required a binding to the "element" object found to exist, and therefore had a shareable variable initialized by binding, the posttrans branch in general requires a copy of the reference to the "element" object, and therefore has an unshareable variable initialized by a copy. The posttrans branch variable cannot be shared in general because it may be updated to correspond to a pretrans binding assignment. It would then also update the discriminated reference value, thus failing to preserve the program's meaning. That the copying is unnecessary in this particular case cannot be determined without data flow analysis.

Point D demonstrates that the transformation follows discrimination bindings related to its parameters ("seq" at this point) and transforms them also. The discrimination binding construct is of the same type as the discriminant object expression (REFTO(element)) but is always bound, so that it can itself be followed without being checked for existence.
Auxiliary transformations in this transformation step include expressing the yield of "index" as a total-copy of "indextrav", where previously it was a top-copy. This applies because the shared "element" components of the "edgelem" have been replaced by unshared components of a type which has no shared components at any remove, the primitive reference type "REFTO (element ...)".

The point raised previously concerning the choice of alternative forms for expressing equivalent operations is exemplified in a different way at C and E. The question here is: "How 'large' a construction should be brought inside an introduced discrimination?" In pushout, when transforming the argument's binding assignment contexts, we discriminated on the whole assignment statement. Here we have several options - the smallest enclosing statement, the enclosing expression, or the smallest expression necessary - and have chosen the last. Compare it with what the first would produce at C:

```
WITH seq DISCRIM
  WHEN exseq: TOTALCOPY EXISTS
    THEN pop YIELD elem OF FOLLOW (exseq) ENDWHEN
  OTHERWISE error ...
```

and what the choice of the "enclosing expression", which in this case is the argument of TOTALCOPY, would produce:

```
pop YIELD TOTALCOPY
  (YIELDOF WITH seq DISCRIM
    WHEN exseq: TOTALCOPY EXISTS
      THEN YIELD elem OF FOLLOW (exseq) ...)
```

(Neither this version nor the one chosen admits the yield-propagation auxiliary.)
Choosing to use a wider context for the discrimination makes the definition of a transformation more complex, because in general there are more cases uncovered thereby. But on the other hand, the wider the boundaries of discrimination, YIELDOF, and parametric constructs are, the more opportunity there will be in general for auxiliary transformations to improve the program. The "smallest expression" case demonstrated here is probably too restrictive, and pragmatically an appropriate context is perhaps the largest expression of which the transformed construct is a part. In this case this would have produced

\[
pop \text{ YIELD YIELDOF (WITH } \text{ seq DISCRIM}
\]
\[
\text{ WHEN exseq: TOTALCOPY EXISTS}
\]
\[
\text{ THEN yield TOTALCOPY (elem OF FOLLOW (exseq)) ...}
\]

from which yield-propagation produces

\[
\text{WITH seq DISCRIM}
\]
\[
\text{ WHEN exseq: TOTALCOPY EXISTS}
\]
\[
\text{ THEN pop YIELD TOTALCOPY (elem ... .}
\]

Another good example of the effect of context width is provided by pushout. Were pushout to take only the right hand side of a binding assignment as the context to be transformed (rather than the whole
statement) the effect of "pushout (y); pushout (x)" on

\[ x \leftarrow y \]

(where \( x \) and \( y \) are shareable and possibly unbound) would be

\[ x \leftarrow \text{YIELDOF}\ (\text{WITH}\ \text{YIELDOF}\ (\text{WITH}\ y\ \text{DISCRIM}
\text{WHEN}\ \text{exy}:\ \text{TOTALCOPY}\ \text{EXISTS}
\text{THEN}\ \text{YIELD}\ \text{FOLLOW}\ (\text{exy})\ \text{ENDWHEN}
\text{OTHERWISE}\ \text{YIELD}\ \text{NONEX}\ \text{ENDDISCRIM})
\text{DISCRIM}
\text{WHEN}\ \text{exrhs}:\ \text{TOTALCOPY}\ \text{EXISTS}
\text{THEN}\ \text{YIELD}\ \text{REFTO}\ \text{exrhs}\ \text{ENDWHEN}
\text{OTHERWISE}\ \text{YIELD}\ \text{NONEX}\ \text{ENDDISCRIM}).\]

6.3.3 **Nullfornonex**

The possibly unbound constructs in this example are now all of type "\( \text{REFTO (element SHAREABLE eltgroup)} \)". All their "POSSIBLY" marks are removed by applying nullfornonex (NIL), extending the range of values of the type of the objects bound (\( \text{REFTO(...)} \)) to include the value NIL which indicates the logical nonexistence of the binding (\( \text{NREFTO(...)} \)). The auxiliary transformation called conditional flattening has several applications here; only the posttrans forms are shown.

**ALL NULLFORNONEX ON POSSIBLY REFTOS**

**BUNDLE stack [**

\(*\ \text{vano and abstract unchanged *})

**CAPSULE stack = ( ... );**

**CAPSULE stack2 = (STATE: STRUCT (**

\] seq: \( \text{NREFTO (element SHAREABLE eltgroup)} \);

auxil: edgelem);
CAPSULE element = (STATE: STRUCT (elem: integer; succ: NREFTO (element SHAREABLE eltgroup)));  

CAPSULE edgelem = (STATE: STRUCT (curr: NREFTO (element SHAREABLE eltgroup); pred: NREFTO (element SHAREABLE eltgroup)));  

SHAREGROUP eltgroup OF element;  

pop = SELF FUNC: integer; 
BEGIN 
pop YIELD TOTALCOPY (elem OF YIELDOF (IF seq <> NIL THEN YIELD FOLLOW (* TOTALCOPY *) seq) ELSE error ("illegal follow")) ENDIF);  
(* shrink at first *) IF seq <> NIL THEN (* seq <- succ OF exseq *) IF succ OF FOLLOW (seq) <> NIL THEN (* seq <- TOTALCOPY (TOTALCOPY (succ OF FOLLOW (seq))) *) seq := succ OF FOLLOW (seq) ELSE seq := NIL ENDIF ELSE error ("shrink - finished traverser") ENDIF END;  

push = SELF PROC (n: integer); 
BEGIN (* grow at first with totalcopy(n) *) BEGIN VAR newel: element SHAREABLE eltgroup; elem OF newel := n; (* succ OF newel <- seq *) (* conditional flattening *) succ OF newel := seq; seq (* <- REFTO newel *) := REFTO newel END END;  

reader = SELF FUNC: REFTO (integer); 
BEGIN reader YIELD TOTALCOPY (elem OF YIELDOF (IF curr OF auxil <> NIL THEN YIELD (* FOLLOW (TOTALCOPY (curr ... *)) FOLLOW (curr OF auxil) ELSE error ("illegal follow")) ENDIF));
IF curr OF auxil <> NIL
THEN
  pred OF auxil := curr OF auxil;
  IF succ OF FOLLOW(curr OF auxil) <> NIL
  THEN curr OF auxil := succ OF FOLLOW(curr OF auxil)
  ELSE curr OF auxil := NIL
ENDIF
ELSE error ("advancing finished traverser")
ENDIF END (* reader *)

total writer = SELF PROC (arg: REFTO (integer));
BEGIN ... END;

reread = SELF PROC;
BEGIN BEGIN
  curr OF auxil := seq;
  pred OF auxil := NIL
END END (* reread *);

more = SELF FUNC: boolean;
BEGIN
  more YIELD curr OF auxil <> NIL
END (* more *);

index = SELF FUNC (indexpr: integer): edgelem;
BEGIN
  pred OF indextrav := curr OF indextrav;
  BEGIN VAR hold succ : NREFTO (element SHAREABLE eltgroup)
    := succ OF YIELDOF
    BEGIN IF curr OF indextrav <> NIL
      THEN YIELD FOLLOW(curr OF indextrav)
      ELSE error("selection from nonexistent")
    ENDIF;
    (... (* curr of indextrav <- totalcopy (hold succ) *)
    (* conditional flattening *)
    curr OF indextrav := hold succ
  END
  ...
END (* index *)
compute_is_empty = SELF FUNC: boolean;
| BEGIN (* compute_is_empty YIELD NOT EXISTS seq *)
| compute_is_empty YIELD (* NOT seq <> NIL *)
| seq = NIL
| END;

(* other compute_operations *)
)
]

VAR a: stack$ := NEW stack $ stack (16,17,18);
VAR b: stack $ stack := NEW stack $ stack (10,11);
VAR c: stack $ stack := NEW stack $ stack (-5,-6);
BEGIN
a := b;
IF a = b
 THEN a.push(33) ENDIF
END.

The discrimination of existence on the transformation’s arguments becomes, as at L, a simple test within the range of values for the range-extending value NIL. Where a discrimination branch variable is applied solely in object contexts it disappears in the posttrans conditional branch. This is true of every case here. If the discriminated expression has no side-effects then it is substituted for every application of the branch variable. If the branch variable is declared as a copy then the expression is enclosed in a corresponding copier before substitution. If the expression may have side-effects it requires the more complex form of substitution with a holding variable seen at O. In either case the auxiliary transformations may remove redundant copying (L) and flatten the conditional (N).

In many cases the original run-time error detection semantics can be preserved throughout the transformation process. Note the error messages in "pop" (J,K), which correspond to stack underflow, for instance. Because the condition of popping an empty stack was not explicitly trapped in the abstract capsule but left to the implicitly
detected error of detraversing or shrinking at a finished traverser, the error message is couched in terms of that former condition, even though there is no longer any sequence or traverser. (The detraversing error was left implicit by evert and so appears only as an error in selecting from following a nonexistent reference.) One error that disappeared as soon as the sequence was everted, however, is that of exceeding the bound on the size of the sequence (see section 6.3.4 below).

A few if-statements remain that may be simplified only by more sophisticated analysis (K and M). The assignments in their branches exclude them from conditional flattening, even though it is apparent to human analysis that they could be safely simplified in the same fashion.

6.3.4 Evaluation

Evaluation of the capsule that results from this series of transformations has several aspects, sufficiency and efficiency among them. Most important is the question of whether the transformations have preserved equivalence: does the capsule implement the same abstraction defined by the original? The language level achieved at this point is also noteworthy: since the aim of applying the transformations is to derive an implementation of the abstraction, the capsule should be at the Maclan end of the language spectrum. The efficiency of the resulting capsule can be compared with a hand-written Maclan equivalent to show how well the transformations used have captured the programming knowledge implied by the hand-
implementation of the abstraction, and how well the apparatus for applying a series of transformations integrates the separate pieces of programming knowledge the transformations embody.

It can be verified by inspection that the transformed capsule implements the abstraction defined by the original stack capsule. The sequence of integer values defined at any time by the "elem" components of successive "elements", starting with that referenced by "seq", succeeded by that referenced by "next" of the current, and terminated by a NIL reference value, is operated upon by the transformed "pop" and "push" in a way exactly equivalent to that in which the "SEQUENCE OF integer" responds to the original "pop" and "push" operations. A formal proof is not attempted here, but is in principle possible. The stack operations are not the only ones that must be equivalent for the two capsules: the bundle interface operations must also have identical effect. This is evidently so, upon inspection.

However, exceeding the bounds of the bounded stack produces differing results in the two capsules. Both capsules detect stack underflow but report different errors. The representation would report an attempt to follow NIL, whereas the abstraction (implicitly) detects a selection from a nonexistent binding. In that both detect an error we can say that their behaviour is equivalent. When the bounded stack of the original abstraction overflows there is a similar implicit detection of the error. In the linked-list representation, however, no bound on the size of the stack is imposed either explicitly or implicitly. In this case we have what might be called "partial equivalence", by analogy with "partial correctness": for uses of
the capsule which keep the abstract version free of error the concrete
version behaves equivalently, but no guarantee of the behaviour of the
concrete is given for operations which are an error for the abstract.
Maclan is capable of expressing the required detection of abstract
effects. In particular the property of checking the upper bound could
have been incorporated by a more detailed form of even which included
some measure of of the sequence's size which could be checked against
the bound on every grow operation.

The issue of whether behaviour under error conditions forms part
of the semantics of a programming language (and hence must be mirrored
exactly in its implementation) is not pursued here, and no consistent
approach to error detection has been taken in the transformations. We
conclude that "stack2" is sufficiently equivalent to the abstract
stack for the purpose of implementation.

The language in which the capsule is expressed may be established
by inspection (manually, or by an Xlan procedure). Maclan is most
easily defined, with respect to the Broad Spectrum, by exclusion: in
particular, it has only manifest- and singly-free-sized data types, no
dynamic binding, no sharing except by reference, no POSSIBLY, no
discrimination, no unions, etc. These conditions are fully met by the
definition of "stack2" derived above. Hence this capsule is pure
Maclan; and because it is equivalent to the abstract stack it is a
"representation" of it.

Disregarding for the moment the bundle interface operations, how
do the basic "push" and "pop" operations compare in efficiency with
their hand-coded equivalents? Writing "push" directly in Maclan to operate on the same state declaration, for instance, gives

```maclan
push = SELF PROC (n: integer);
BEGIN
  seq := REFTO NEW element (TOTALCOPY (n), TOTALCOPY (seq))
END.
```

Comparison of the efficiency of this operation and that derived by transformation can be made by considering the actions of the Maclan machine that are caused by evaluating them. The machine operations specified by the hand-written version are: the creation of a new "element" object, with copies of the parameter "n" and "seq" as its initial component values; the creation of a reference to the object; and assignment of the reference value to "seq". In contrast, the derived form of this procedure declares a variable "newel", which implicitly creates a new "element" object of DONTCARE value; copies of "n" and "seq" are then assigned to this object's components; a reference to the object is generated and assigned to "seq"; and the variable "newel" is discarded when control leaves the procedure "push".

The version derived by transformation contains no redundant operations that may be eliminated in a simple fashion. The only substantial difference between the two versions that might affect efficiency is the creation of the variable "newel" as opposed to the use of an anonymous new object generator. Without a knowledge of the storage allocation mechanisms and the relative costs of anonymous object generation and named variable creation the difference in efficiency between the two versions cannot be calculated.
The presence of an explicitly named variable in the transformed version of this operation may be traced back to the engineering decision to use separate assignments to components of new object components, in order to make further transformations simpler. This ease of transformation costs very little in terms of the efficiency of the final result.

Now consider "pop". Here a hand-coded version appears considerably simpler:

```
pop = SELF FUNC : integer;
BEGIN
  IF seq = NIL
    THEN error ("stack underflow")
  ELSE pop YIELD TOTALCOPY (elem OF FOLLOW(seq))
  ENDIF;
  seq := succ OF FOLLOW(seq)
END.
```

Detecting the error of stack underflow and selecting the value to be returned are essentially the same operations in both versions. Removing the popped element from the list is two degrees simpler in the hand-coded version. One difference is the redundant detection of whether "seq" is NIL: the effects of the preceding statement must be analyzed, and the meaning of a call of "error" known (i.e. that the computation aborts), before it can be deduced that, for this statement to be evaluated, "seq" cannot be NIL. This is a higher order of performance in program optimization than we are expecting here.

The second difference is the continuing separation of the assignment to "seq" into two parts, depending on the value of "succ OF FOLLOW(seq)". Here the crudeness of the auxiliary transformation expressed in Xlan shows: because the assignment is to "seq", which is
a constituent of the conditional expression, conditional flattening to produce a simple assignment does not apply. Thus due to the inadequacy of the short-range and simple-minded auxiliary transformations this operation continues to display its transformational history, at some cost to efficiency.

The forms of the bundle interface operations are highly dependent on the definition of their bundle value manipulation operation interfaces, which are unalterable. Hand-coding the bios cannot change these definitions. The derived forms of the bios show the same problems as are displayed by "pop" (i.e. some redundant testing of error conditions and unnecessary discrimination of simple assignments) but are otherwise as good as hand-coded versions.
6.4 Vector with top

Three transformational steps are required to represent instances of "stack" bound to "b" in the common form of a vector (fixed-length sequence) with an index variable indicating the stack top. Two structural transformations are needed, followed by a binding transformation:

- padout at start (b);
- indices for traversers (b);
- nullfornonex by range (0) (b.auxil, b.limit, b.writer.growaux).

The last transformation's arguments refer to the various index variables which are used on the sequence.

6.4.1 Padout at start

The application of "padout at start(b)" causes another split in the capsule "stack", producing "stack3". Padout makes this capsule's state into a sequence of fixed size, and translates GROW, SHRINK and MAKE_EMPTY operations into movements of an added traverser "limit" within the new sequence. The elements of the sequence are copied back and forth to preserve their order and compactness under the posttrans GROW and SHRINK operations. The auxiliary variable "growaux" is introduced as an unprotected traverser for use in moving elements of the sequence.
BUNDLE stack [  
CAPSULE stack = ( ... );  
CAPSULE stack2 = ( ... );  
] CAPSULE stack3 = (STATE: SEQUENCE 25 OF integer  
TRAVERSED auxil, limit (LAST);  

pop = SELF FUNC: integer;  
BEGIN  
  pop YIELD (* totalcopy (# first #) *)  
  TOTALCOPY (# limit #);  
  (* shrink at first *)  
  IF FINISHED limit  
  THEN error ("shrink - finished traverser")  
  ELSE NEXT limit  
  ENDIF  
END;  

push = SELF PROC (n: integer);  
BEGIN  
  (* grow at first with totalcopy (n) *)  
  IF FINISHED limit  
  THEN MAKECOINCIDE limit WITH LAST  
  ELSE PREVIOUS limit;  
  IF FINISHED limit  
  THEN error ("bound exceeded")  
  ENDIF ENDIF;  
  (# limit #) (* <- topcopy(n) *) := n  
END;  

reader = SELF FUNC: REFTO (integer);  
BEGIN  
reader YIELD REFTO (* TOPCOPY *) TOTALCOPY ((# auxil #));  
NEXT auxil  
END (* reader *);  

total writer = SELF PROC (arg: REFTO (integer));  
BEGIN  
  IF compute_is_empty()  
  THEN (* grow at first with totalcopy (follow (arg))*  
  IF FINISHED limit  
  THEN MAKECOINCIDE limit WITH LAST  
  ELSE PREVIOUS limit;  
  IF FINISHED limit  
  THEN error("bound exceeded")  
  ENDIF ENDIF;  
  (# limit #) (* <- totalcopy (follow (arg)) *)  
  := FOLLOW (arg)
ELSE IF FINISHED auxil
| THEN (* grow after last with ... *)
| PREVIOUS limit;
| IF FINISHED limit
| THEN error ("sequence bound exceeded") ENDIF;
| BEGIN VAR growaux: UNPROTECTED TRAVERSER OVER SELF;
| MAKECOINCIDE growaux WITH limit;
| WHILE NOT growaux COINCIDES LAST DO (* Q *)
| (# growaux #) := (# growaux BY next #);
| NEXT growaux
| ENDDO;
| (# LAST #) := FOLLOW (arg)
| END
| ELSE (# auxil #) := FOLLOW (arg);
| NEXT auxil
| ENDDO ENDDIF ENDIF
| END (* total_writer *);

reread = SELF PROC;
| BEGIN (* reset auxil *)
| MAKECOINCIDE auxil WITH limit
| END;

more = SELF FUNC: boolean;
| BEGIN more YIELD NOT FINISHED auxil END;

compute_is_empty = SELF FUNC: boolean;
| BEGIN compute_is_empty YIELD FINISHED limit END;

VAR a: stack $ stack2 := NEW stack $ stack (16,17,18);
VAR b: stack $ stack3 := NEW stack $ stack (10,11);
VAR c: stack $ stack := NEW stack $ stack (-5,-6);
BEGIN
| a := b;
| IF a = b
| THEN a.push(33) ENDIF
| END.

The transformation recognizes the "GROW AT FIRST" operations of
"total_writer" and "push" at P as needing no movement of existing
sequence components, but only copying of the new value to the position
indicated by the updated "limit". In this form of the "pop" and
"push" operations we can see the germs of the relative simplicity of
the bounded stack representation being developed: that changing the
stack's size at its "top" (i.e. at FIRST of the abstract sequence) is much simpler than within its depths.

"Total_writer" is called repeatedly by the vmo for total-copying assignment (e.g. "b := a"). It can be seen that for each component that is copied after the first, all previously written components must be re-written to make room at the end for the new component (point Q). This is a result of the particular form taken by the definition of the interfacing operations, viz. that the order of writing elements is the "natural" order of the sequence from FIRST to LAST. This inefficiency is regrettable. However, the performance of the representations derived from the various forms of padout is so variable for slight differences in the use of size-changing operations that some such mismatch is inevitable (compare "GROW AFTER LAST" with "GROW AT FIRST" in the cases of padout at start and padout at finish, for example).

The previously implicit checks on the bounds of the sequence are now made explicit in the transformed GROW operations (and also for shrinking at a finished traverser, as in "pop"). Without these checks detecting these errors would still be done implicitly, but the error messages would not refer to the originally defined instance of the sequence abstraction.

6.4.2 Indices for traversers

All the sequence's traversers are replaced by possibly unbound index variables of subrange type. The bounds of the subrange are naturally the size-bounds of the sequence; the value of an existing
index variable represents the point within the sequence indicated by
the corresponding traverser. A nonexistent binding of an index
represents the FINISHED state of the traverser. The growth in the
capsule's code size caused by this transformation is evident: it
results from explicit checking for the existence of the index variable
before almost any operation is applied. This corresponds to checking
for the non-finished state of the traverser, treating the finished
state distinctly and sometimes as an error.

INDICES FOR TRAVERSERS

BUNDLE stack [
  CAPSULE stack = ( ... );
  CAPSULE stack2 = ( ... );
] CAPSULE stack3 = (STATE:
  STRUCT (seq : SEQUENCE 25 OF integer;
    POSSIBLY auxil: 1 .. 25;
    POSSIBLY limit: 1 .. 25 := 25);
  pop = SELF FUNC: integer;
  BEGIN
    pop YIELD TOTALCOPY (* (# limit #) *)
    (YIELDOF (
      WITH limit DISCRIM
      WHEN exlimit: TOTALCOPY EXISTS
      THEN YIELD [exlimit] OF seq ENDWHEN
      OTHERWISE error ("detraversing finished")
      ENDDISCRIM)
    );

    (* shrink at first *)
    WITH limit DISCRIM
    WHEN EXISTS THEN
      (* next limit *)
      WITH limit DISCRIM
      WHEN exlimit: EXISTS THEN
        IF exlimit = 25
          THEN limit <- NONEX
        ELSE exlimit := exlimit + 1
        ENDIF ENDWHEN
      OTHERWISE error ("advancing finished traverser")
      ENDDISCRIM ENDWHEN
    OTHERWISE error ("shrink - finished traverser")
]
push = SELF PROC (n: integer);
BEGIN
(* grow at first with totalcopy (n) *)
WITH limit DISCRIM
WHEN EXISTS THEN
(* previous limit *)
WITH limit DISCRIM
WHEN exlimit: EXISTS THEN
IF exlimit = 1
THEN limit <- NONEX
ELSE exlimit := exlimit - 1
ENDIF ENDWHEN
OTHERWISE error ("advancing finished")
ENDDISCRIM;
WITH limit DISCRIM
WHEN EXISTS THEN ENDWHEN
OTHERWISE error ("bound exceeded")
ENDDISCRIM
ENDWHEN
OTHERWISE (* makecoincide limit with last *)
limit <- 25
ENDDISCRIM;
(* (# limit #: n *)
WITH limit DISCRIM
WHEN exlimit: TOTALCOPY EXISTS THEN
[exlimit] OF seq := n
ENDWHEN
OTHERWISE error ("detraversing finished")
ENDDISCRIM
END;
reader = SELF FUNC: REFTO (integer);
BEGIN
reader YIELD REFTO TOTALCOPY
(YIELDOF (WITH aux11 DISCRIM
WHEN exaux11: TOTALCOPY EXISTS
THEN YIELD [exaux11] OF seq ENDWHEN
OTHERWISE error ("detraversing finished")
ENDDISCRIM);
The constants "25" and "1" appear often in this capsule to denote the index values for the upper and lower bounds of the sequence. An index variable representing a traverser becomes nonexistent when it is advanced by NEXT from the upper value (25) or by PREVIOUS from the lower (1), where the correspondingly advanced traverser became FINISHED.
Whether or not TOTALCOPY is used to declare the numerous discrimination branch variables is noteworthy (points R and S). Where the purpose of the discrimination is to update the object possibly bound to the yield of the discrimination expression, the branch variable must share that object and is declared with its sharegroup or an implicit ANY (point S). To minimize introduced sharing this is not done in cases where the discrimination is not for the purpose of updating; instead the branch variable is declared as a copy (point R).

The numerous explicit checks for existence have many apparent redundancies amongst them. To remove them would require more powerful data flow analysis than is within the scope of this work.

6.4.3 Nullfornonex

Possibly unbound variables of any subrange type can extend the range of their type to indicate their unbound (nonexistent) state by a specific value, as was done to the reference variables in the linked-list example above. The type here is a subrange of integers whose range may be extended by picking any previously excluded integer value. We choose zero to keep the range as small as possible, for clarity.

nullfornonex by range (0) (b.auxil, b.limit, b.writer.growaux) produces
BUNDLE stack [ 
CAPSULE stack = ( ... ); 

CAPSULE stack2 = ( ... ); 

CAPSULE stack3 = (STATE: 
  STRUCT (seq: SEQUENCE 25 OF integer; 
  auxil: 1 .. 25; 
  limit: 1 .. 25 := 25); 

tp = SELF FUNC: integer; 
BEGIN 
  pop YIELD TOTALCOPY (* (# limit #) *) 
  (YIELDOF (IF limit < 0 
    THEN YIELD [limit] OF seq 
    ELSE error ("detraversing finished") 
    ENDIF 
    )); 

  (* shrink at first *) 
  IF limit = 0 
    THEN error ("shrink - finished traverser") 
  ELSE (* next limit *) 
    IF limit <> 0 
      THEN IF limit = 25 
        THEN limit := 0 
        ELSE limit := limit + 1 
        ENDIF 
      ELSE error ("advancing finished traverser") 
      ENDIF ENDIF 
END; 

push = SELF proc (n: integer); 
BEGIN 
  (* grow at first with totalcopy (n) *) 
  IF limit = 0 
    THEN (* makecoincide limit with last *) 
      limit := 25 
  ELSE (* previous limit *) 
    IF limit <> 0 
      THEN IF limit = 1 
        THEN limit := 25 
        ELSE limit := limit - 1 
        ENDIF 
      ELSE error ("advancing finished") 
      ENDIF; 
    IF limit = 0 
      THEN error ("bound exceeded") 
    ENDIF ENDIF;
(* limit := n *)
IF limit <> 0
THEN [limit] OF seq := n
ELSE error ("detraversing finished")
ENDIF
END;

reader = SELF FUNC: REFTO (integer);
BEGIN
reader YIELD REFTO TOTALCOPY (
YIELDOF (IF auxil <> 0
THEN YIELD [(TOTALCOPY *) auxil] OF seq
ELSE error ("detraversing finished")
ENDIF);
(* next auxil *)
IF auxil <> 0
THEN IF auxil = 25
THEN auxil := 0
ELSE auxil := auxil + 1
ENDIF
ELSE error ("advancing finished traverser")
ENDIF
END (* reader *);

total writer = SELF PROC (arg: REFTO ((integer));
BEGIN ...
IF limit = 0
THEN error ("sequence bound exceeded") ENDIF;
BEGIN VAR growaux: 0 .. 25;
(* makecoincide growaux with limit *)
growaux (* <- TOTALCOPY *) := limit;
WHILE NOT (* growaux coincides last *)
YIELDOF (IF growaux <> 0
THEN YIELD growaux = 25
ELSE YIELD false
ENDIF) DO
(* (# growaux #) := (# growaux by next #) *)
IF growaux <> 0
THEN [growaux] OF seq :=
YIELDOF (IF growaux <> 0
THEN IF growaux = 25
THEN error ("detrav finished")
ELSE YIELD [growaux + 1] OF seq ENDIF
ELSE error ("advancing finished trav")
ENDIF)
ELSE error ("detraversing finished")
ENDIF;
END (* total_writer *);
reread = SELF PROC;
   BEGIN (* reset auxil *)
   (* auxil <- totalcopy (limit) *)
   (* via discrimination and conditional flattening *)
   auxil := limit
END;

more = SELF FUNC: boolean;
   BEGIN (* yield not finished auxil *)
   more YIELD auxil <> 0 END;

compute_is_empty = SELF FUNC: boolean;
   BEGIN (* yield finished limit *)
   compute_is_empty YIELD limit = 0 END;

VAR a: stack $ stack2 := NEW stack $ stack (16,17,18);
VAR b: stack $ stack3 := NEW stack $ stack (10,11);
VAR c: stack $ stack := NEW stack $ stack (-5,-6);
BEGIN
   a := b;
   IF a = b
      THEN a.push(33) ENDIF
END.

In this case the conditional flattening that applied previously
is unable to work in several places. Both points U (which derives
from "NEXT auxil" where auxil is not finished) and T (from "PREVIOUS
limit") could be simplified only by going beyond the textual form of
the two branches of the conditional to analyze their expressions'
arithmetic properties and produce "auxil := (auxil + 1) MOD 26" at
point U, for instance. Text-matching is thus unable to conflate the
results of successive transformations as it did in the previous exam-
ple.
6.4.4 Evaluation

The capsule derived by this series of three transformations is expressed in Maclan and hence is a representation of the bounded stack abstraction, assuming the transformations to be correct. Unlike the linked-list representation, this "vector plus top index" representation is bounded in the same way as the abstraction. As before its error messages may differ for a particular illegal operation.

The structure defined by the state of the capsule is the same as could be expected of a hand-written version of the capsule in Maclan. The operations achieve the correct effects on this structure, but also specify many redundant tests and much unnecessary discrimination of like cases. The redundant tests for the correct state of the index variables before each transformed traverser operation can only be absorbed by using more powerful analytical methods; but the reason they arise in the first place should be noted. The operations on traversers that are introduced by the padout transformation (e.g. in "GROW AFTER auxil ...") have been written with implicit knowledge of the relationships of the traversers' indicated positions in the sequence, when they may possibly be finished, etc. But this knowledge is not explicitly expressed in the text of those operations, unlike the knowledge about existence of bindings made textually explicit in the branches of a discrimination statement, for instance. Since all later transformations operate solely on knowledge gleaned from the textual program structure, they are unable to recover the knowledge that was lost and must include checks at points that are in fact secure.
The inefficiency of growing at other than the first position of the sequence (in "total_writer") has been mentioned above. This inefficiency is unavoidable, since the way in which "total_writer" is called is fixed by the interfaces of the bundle's vmos and cannot be modified to fit different requirements arising out of capsule transformations. An interface mechanism that allowed for greater modification as a result of structural transformations than does the present vmo/bio system would perhaps improve this aspect of the derived representation. No such mechanism is known.

6.5 Alternative transformation paths

We choose to represent objects bound to the variable "c" and all those generated by the three object generators "NEW stack (...)" in the same way as "a", but derive the representation somewhat differently. The application of transformations for "a" was

\[
\begin{align*}
evert (a); \\
pushout & (\text{succ OF stack.stack2.element,} \\
& \text{pred OF stack.stack2.edgelem,} \\
& \text{curr OF stack.stack2.edgelem,} \\
& \text{seq OF stack.stack2}); \\
nullfornonex & \text{ by range (NIL) (stack.stack2.seq,} \\
& \text{stack.stack2.element.succ,} \\
& \text{stack.stack2.edgelem.pred,} \\
& \text{stack.stack2.edgelem.curr).}
\end{align*}
\]
The final "stack2" capsule's description has the history:

```
hist: (basis: El
derivation:
< ("seq", ("pushout",
   inner: NREL1))
("element", (inner: (basis: EL1
derivation:
< ("succ", ("pushout"
inner: NREL2))

("edgelem", (inner: (basis: EEl
derivation:
< ("pred", ("pushout"
inner: NREL3))
("curr", ("pushout"
inner: NREL4))

> ))))

> ).
```

Here "El", "EL1", "EEl" and "NREL1" etc. denote descriptions. These are the description of the everted stack (El), and those of its "element" (EL1) and "edgelem" (EEl) capsules. The "NRELn" are descriptions, equal but not shared, of the types defined by occurrences of "NREFTO (element SHAREABLE eltgroup)". They take the form

```
hist: (basis: RELn
derivation: 'nullfornonex by range' (NIL))
```

where "RELn" is a description of a type defined by "REFTO (element SHAREABLE eltgroup)".

The transformations for "c" on the other hand will be specified in pushout and nullfornonex pairs, and their arguments all identify "stack" rather than merely some instances of it:

```
evert (stack.stack);
pushout (seq OF stack.stack);
nullfornonex by range (NIL) (stack.stack.seq);
pushout (succ OF stack.stack.element);
nullfornonex by range (NIL) (stack.stack.element.succ);
etc.
```
The application of these transformations affects the capsule defining "stack" and the new object generators of the variable declarations, until the last is about to be applied:

nullfornonex by range (NIL) (stack.stack.element.pred).

At this point the history of the capsule's description is

\[
\text{hist: (basis: } E2 \quad \text{derivation:}
\begin{align*}
&< ('\text{seq}', ('\text{pushout}' \\
&\quad \text{inner: NREL5})) \\
&('\text{element}', (inner: (basis: EL2 \\
&\quad \text{derivation:}
\begin{align*}
&< ('\text{succ}', ('\text{pushout}')) \\
&> )))
\end{align*}
\end{align*}
\begin{align*}
&('\text{edgelem}', (inner: (basis: EE2 \\
&\quad \text{derivation:}
\begin{align*}
&< ('\text{pred}', ('\text{pushout}' \\
&\quad \text{inner: NREL6})) \\
&('\text{curr}', ('\text{pushout}' \\
&\quad \text{inner: NREL7})) \\
&> )))
\end{align*}
\end{align*}
\]

The "NRELn" are descriptions equivalent to those of the same names above; similarly, "E2", "EE2" and "EL2" are equivalent to "El", "ELl" and "EEl" respectively.

Applying the last transformation to this description produces a description which is equivalent to that for "stack2". The transformation is therefore not actually applied to the capsule "stack", but the type of "c" is changed to be that defined by the capsule fitting the description. The transformation is however applied to the overhanging new object generators that lie outside the capsule. The object declarations that result are:
VAR a: stack $ stack2 :=
  NEW stack $ stack2 (REFTO NEW element (16,
    REFTO NEW element (17,
      REFTO NEW element (18, NIL)),
    DONTCARE));

VAR b: stack $ stack3 :=
  NEW stack $ stack2 (REFTO NEW element (10,
    REFTO NEW element (11, NIL)),
  DONTCARE);

VAR c: stack $ stack2 :=
  NEW stack $ stack2 (REFTO NEW element (-5,
    REFTO NEW element (-6, NIL)),
  DONTCARE).

6.6 Discussion

These examples show that this small number of transformations embodies central parts of our knowledge of data structuring. We have used them to define two significantly different representations of a simple abstract data type. The BSL programming language supports both representations within the one program, and the transformation system has enabled us finely to select the points of application.

Applications of the principles of representation described in section 4.4.10 are seen in both examples. The variable-sized sequence is made regular by eversion, or invariant by padding out. The variable, possibly unbound, shared bindings resulting from evert are made invariant and structure is represented by values, by the combination of pushout and nullfornonex. The traversers remaining after padout are constrained variable bindings, which indices represent by values of a fixed object.
The quality of the final implementations is as much a result of the subject language's design as of the transformations'. The linked-list representation is almost as good as its hand-coded equivalent because BSL's appropriate language forms enable the transformations' relatively simple analyses to integrate their distinct parcels of data structuring knowledge. The second example integrates its final code less well. As we have noted earlier this appears to demonstrate a failing in the language forms available: that is, the lack of textual forms in BSL to express in the program the padout transformation's knowledge about the state of traversers. This failing is made apparent by the decision to restrict the system to simple text-recognizing transformations: the superior analytic power of transformation systems like that of Kibler et al. [1977] would readily simplify this redundancy-laden code. Note however the simplicity of this code compared to that which would be necessary if traversers were not constrained by the language forms to indicate only one object's components. But it is clear from this example that the treatment of traversers has not been as firm as it should be, and for human comprehension also we should, in using traversers, adhere as closely to the strong-correspondence principle as we do when using POSSIBLY bindings and unions.

A second conclusion about language design is that there are good reasons for distinguishing some capsules as unbundled because of the role that they play in defining implementation capsules. The bundle as described in Chapters 4 and 5 contains capsules of two classes, with distinct roles: the "abstraction" and "implementation" capsules.
A third class of capsule is also present, exemplified by "element" and "edgelem" in the linked-list implementation capsule. These capsules are used to define an implementation capsule, but themselves possess no explicitly defined operations and have not been formed into bundles in the example. They are never split, and hence do not require the value manipulation interface that enbundling them would give. Thus lack of splitting might be taken as a rule of thumb to indicate that the formation of a bundle was not necessary. But more importantly, we observe that these capsules contain no operations, and no structural transformations are applied to them. They are used only to name data templates, with no operations but component selection. Whereas the capsule is an information-hiding, abstracting mechanism, these have nothing to hide and no abstraction distinct from themselves. A textual distinction should be made for which TEMPLATE might be an appropriate symbol, as in

```
TEMPLATE element = STRUCT (POSSIBLY succ: element; elem: integer).
```

Designing the library of transformations applied here has much in common with designing the parts of any complex software system, and is approached similarly by first considering specifications of the parts and then their realization. The distinctive feature of the library as a software system is that the interaction between its parts is very weak. We have expressed the specifications of the transformations informally, as in section 4.2.2.1; for example, "replace a binding to an object by a reference to it", for pushout. The realizations, expressed as Xlan procedures, must fill in a great amount of detail
missing from the specification. The techniques adopted for this we refer to as "transformation engineering". Some of these techniques have been discussed earlier in this chapter, but some more general points drawn from these examples follow.

It is desirable that the number of transformations should be small, avoiding variants which express no substantial differences to the user. This requires a realization to cover variations in the subject language's forms that could otherwise have been handled by distinct transformations. Hence, for example, the one form of pushout must work whether the subject binding is marked POSSIBLY or not, despite the differing treatments required by these variant forms.

The abstract specification is typically couched in terms of an effect on a binding's formal type or a capsule's state, thus determining one portion of the realization. But the transformation's effects on the program's related operations are given much greater freedom. As described earlier the width of contexts considered by the transformation is important in governing the complexity of the realization and the amenability of the result to further transformation. If contexts are too small, the effects of a series of transformations remain unintegrated.

The only guidance in this area is pragmatic, the ability to use larger contexts in transformations being restricted by the difficulties of writing them at all. The difficulties are due to the diffusion of significant meaning in the parse-tree, even in constructs with strong correspondence; and to our weak perceptions of the interactions
between the independently written transformations when they are applied to overlapping contexts. No more than an informal discipline can be imposed to assist in the process, one rule being to avoid generating constructions with boundary interfaces that are narrow and hard to penetrate.

Examples of this rule's application are in the many uses of open form, in-line substitution for operations where possible and simple; adding closed form operations and the YIELDOF construction should be avoided, or a YIELDOF context enlarged as far as possible. Open forms are shown in evert's treatment of RESET or NEXT applied to a traverser, and it is clear that open forms of operations avoid the hard boundary imposed by immutable parameter passing to a closed-form operation definition. Nevertheless, we have expressed some operations such as "index", "compute_make_empty", etc. in closed form. They are either too lengthy to make in-line forms practicable, or their interface is so simple as not to interfere with later transformations.

Enlarging the context of YIELDOF in realizing transformations was discussed earlier; some ability to enlarge it after the fact of major transformation is given by the related auxiliary transformation called yield propagation.

It is apparent that this rule of wide contexts was applied insufficiently in writing the transformations: it should have led to setting the appropriate context for the pushout transformation of a constituent of a selecting object expression to the whole expression or further, rather than the limited extent used here [section 6.3.2].
The principle is shown at its best in extending the context of transformation for pushout of the right hand side of a binding assignment, to the whole of that assignment statement.
7 CONCLUSION

7.1 Summary

7.2 Conclusions
7.2.1 Descriptions in data structure refinement
7.2.2 Is transformation really necessary?
7.2.3 The transformations
7.2.4 Interaction between language and transformations
7.2.5 Abstract type framework

7.3 Future work
7.3.1 Extending BSL
7.3.2 Adding transformations
7.3.3 Transforming other languages
7.3.4 A fully automatic system
7.3.5 Program proofs

7.4 Conclusion

Whereas the methodology of structured programming is directly supported by the procedure and module structures of languages in Algol, Simula and some languages family, transformation as a tool for program development is not supported by any language. It is justified by a number of examples, including the formally established relationships between recursion and iteration [Sussman and Reitman 1979] and arrays and the programs using them [Steele et al., 1979]; the informal appeal to transformational notions used in developing specific programs [Dijkstra 1976; Pritchard 1979]; and some recent examples of informal transformations applied to formalized data type definitions [Harman and Boy 1979]. Areas in which a more precise treatment has been missing are the use of transformation as a constructive tool for programming; the application of transformations to programs with complex data structures; and the consideration of alternative representations.
The practical application of a programming methodology rests on the support given by the programming language on which it operates. Where the programming language lends only weak support, the difficulties of applying the methodology will reduce its benefits in organizing the programmer's thought processes; where the language lends strong support the use of the methodology goes naturally with use of the language. Working under this paradigm we have developed an object-oriented broad spectrum language that supports a mechanically aided methodology of data structure refinement by transformation.

Whereas the methodology of structured programming is directly supported by the procedure and module structures of languages in the Algol, Simula and capsule languages family, transformation as a tool for program development is not supported by any language. It is founded on a number of examples, including the formally established relationships between recursion and iteration [Burstall and Darlington 1975], and arrays and the programs using them [Standish et al. 1976]; the informal appeals to transformative notions made in developing specific programs [Dijkstra 1976; Pritchard 1979]; and some recent samples of informal transformations applied to formalized data type definitions [Pchtsch and Broy 1979]. Areas in which a more precise treatment has been missing are the use of transformation as a constructive tool for programmers; the application of transformations to programs with complex data structures; and the consideration of alternative representations.
Our Broad Spectrum Language makes good these omissions with sufficient data structure transformations to demonstrate that it is possible to develop programs by transformative stepwise refinement. The contribution of this work lies in the language mechanisms that remove references from the high-level end of the spectrum, and those that express multiple representations of an abstraction at the other; the identification of two major classes of data structure transformation; and the demonstration of a transformation system which integrates all of these, usefully applies some important parts of our knowledge of data representation, and achieves data independence for the Broad Spectrum Language.

7.1 Summary

The Broad Spectrum Language applies the models of program structure and the relationship between program and data space denoted by the terms "strong typing", "data encapsulation" and "dynamic object generation" over the whole spectrum. At the high-level end of the spectrum the language is referenceless. It provides dynamic binding of names to objects and high-level constructors for defining abstract data types. The constructors are for selectable cartesian products, discriminated unions and tree- and sequence-structured types; an abstract data type is defined by an unparametrized information-hiding capsule which defines procedures and functions to apply to instances of the capsule type. The binding of objects to names is potentially fully dynamic, with qualifications on the freedom of aliasing and nonexistent bindings determined by the declaration of sharegroups and
traversers tied to single instances of structures, and the marking of those names that are permitted to be possibly unbound.

The other end of the language spectrum has fixed binding of names to objects, simpler data structures and references, with a weakened form of typing provided by undiscriminated union constructors. Its data structures and bindings are close to those of the conventional Pascal-like virtual machine. The exceptional features of this part of the spectrum are its bundle encapsulation mechanism and its non-primitive treatment of value manipulation operations like assignment. The bundle contains a number of low-level capsules that each implement the same abstraction and are, by the vmo mechanism, assignment-compatible with each other. Generic types play a key role in this.

The transformations presented constitute the elements of a structured description of general principles of representation. Although few in number they are sufficient to demonstrate the system and show the value of removing references from the high-level language. They are divided into three groups: those that restructure data types, those that modify the relations between one object and another, and those that refine the resulting program constructs. The first two groups, the structural and binding transformations, are manually applied; the last group applies automatically. The structural transformations include introducing simple recursive types to represent dynamically variable structures, enclosing variable-length sequences in ones of fixed length, and annihilating levels of structure. The binding transformations take advantage of the distinctions introduced by the range of reference-replacing language constructs to
represent shared objects by copying and nonexistent ones by tagging, as an alternative to representing sharing, nonexistence and dynamic variation by manipulating reference values.

We define the transformations by procedures in an algorithmic programming language Xlan. This language manipulates the text of programs of the Broad Spectrum Language structured as parse-trees of an extended BNF grammar of that language. Detailed definitions of parts of the transformations show the substantial amount of detailed programming semantic knowledge they contain.

The transformation system permits the user to apply transformations to a program selectively, and allows alternative capsules to be derived from a single parent by the mechanism we have called typesplitting. Thus the one-to-one mapping between an abstract data type definition and its implementation in a particular program is relaxed, making it possible to generate appropriate representations as desired for objects that are of the same type but are known to have different patterns of behaviour. An associated system of capsule descriptions allows the transformation system's user to consider small groups of variables for representation independently with reduced risk of unnecessarily duplicating capsules.

7.2 Conclusions

The programming language, transformations and descriptions presented here constitute the small kernel of a complete data structure transformative refinement system. Parts of it are sufficiently
complete to support firm conclusions, while others can provide a basis for further extension. In the former category are elements of the transformation system, arguments for the use of transformations, the distinction between the major and auxiliary transformations, and the abstract data type framework over the whole spectrum. The abstract elements of the Broad Spectrum Language have been summed up already, in Chapter 3.

7.2.1 Descriptions in data structure refinement

Without applying capsule descriptions, automatic type-splitting would lead to a danger of duplicated capsule definitions. Descriptions provide a consolidating force to counter the expansive tendencies of type-splitting, enabling the user of a transformational program development system to concentrate on the representation of one variable or sharegroup at a time. Since a representation that duplicates a previously developed one will not introduce a new capsule definition, concerns about the final program's having in a bundle many capsules describing the same representation do not intrude into the development of a representation for one set of instances of a type.

7.2.2 Is transformation really necessary?

Transformations are essential for representing bindings and in type-splitting, as well as for the representation of sets described in Chapter 2. The representation of bindings cannot be internalized within the capsule defining the variable's or parameter's type,
because the bindings to different instances of the type may be represented differently. This requires that the program construct defining the binding be transformed, as seen in the set example and demonstrated in the representation of bindings by references and by multiple copies.

Transformations are also necessary to derive multiple representations from a single abstraction. This cannot be done by immediate constituent refinement without a conceptual extension for the purpose. Moreover by using transformations we not only have a methodology of refinement by transformation, but a library of representation techniques in the transformations that we have described.

7.2.3 The transformations

A library of transformations is an expression of some of our knowledge about programming and programs in a formalized and directly applicable form. To convey that knowledge to the human reader the transformations must identify their subject knowledge at a suitably abstract level, which has been done here by raising the subject language level to include explicit forms of expression for these abstractions, particularly those of binding. As well as making a mechanized program development assistant possible, these abstractions and their equivalences expressed in a transformation library have a value in providing a notation for comprehending and communicating relationships between a definition of data structures and binding abstractions and their implementation.
The transformation library distinguishes a set of auxiliary transformations from the major data structuring ones. The former apply automatically as optimizing transformations typically many times to each application of a major transformation. The need for a combination of these automatically invoked small-window optimizing transformations to be part of a system of data structuring transformations has been observed recently elsewhere [Partsch and Broy 1979]. In our case the constructions to which the auxiliary transformations apply are themselves the result of applications of the major transformations, and generally not present in the original program. It is therefore apparent that in a sense they are unnecessary: the knowledge that they contain could be absorbed into the major transformations by enlarging the size of the contexts treated and the number of specific cases of constructs distinguished. Doing this would eliminate the need for the automatic applicative mechanism in the transformation system. But the knowledge in the library as a whole is more simply and elegantly formulated as two sets of transformations of different character, than as a smaller number of transformations which all repeat some common auxiliary knowledge.

Furthermore we distinguish "structural" and "binding" classes among the major transformations. They identify significantly different aspects of representing high-level data structures, namely the representation of properties specific to various abstract type constructors and the representation of type behaviour primitives.

Xlan is able to express the conventional context-sensitive aspects of semantics straightforwardly because it allows for
distinctions between left- and right-hand values; is able to express the complex scoping rules of a data encapsulation language; and enables the transformation procedures to relate closely the logically closely coupled transformation of a data type declaration and its instances, which are only loosely coupled in the subject program text. It is also capable of expressing the automatic auxiliary transformations which are simple examples of the optimizing transformations found in the literature, but is limited in this area by the lack of automatic inference or reasoning facilities.

Confidence in the equivalence between a derived representation and the high-level program follows from our confidence in the correctness of each transformation in the library, and is independent of the programmer's care or acumen. Applying the transformations he chooses can affect only the efficiency of the program, not its correctness. Our confidence in the transformations has the same grounds as our confidence in repeatedly used programs of similar size. We have not formally proved any of the transformations to be correct. Proofs would appear to be obtainable, certainly for the auxiliary transformations, and probably for the major transformations, but both are beyond the scope of this work. We have relied on inspection and informal reasoning to support our confidence in the transformations' correctness; had we implemented them all and thus been able to use them extensively our confidence would have been increased.
7.2.4 Interaction between language and transformations

The simple analysis in the transformations is only possible, and successful in producing quality representations, because of certain forms in the subject language. These are strong typing, other strong correspondence, aliasing controlled by sharegroup, object-tied traverser variables, and capsules.

Including the abstractions of forms of bindings in the Broad Spectrum Language makes it possible to recognize the class of binding transformations as being on an equal footing with the structural transformations. These transformations that introduce references, change sharing to copying, weaken the typing of unions, introduce null values, etc. are as important in the development of data structure representations as those transformations that make major structural changes in state descriptions of data types.

Extending the notion of strong typing to other aspects of the Broad Spectrum Language was called the principle of strong correspondence. This made it possible to express the binding and structural transformations, and some useful auxiliary transformations, in Xlan without needing an automatic inference system in that language because the syntactic forms corresponded so closely to the semantics. In the parts of the language where the strong correspondence principle was less strictly applied the transformations were visibly less effective.

The operations that interface assignments of implementation capsules are not of a privileged kind but are transformed in common with other operations of their capsules. They therefore need no special
mechanisms or notice in the transformation procedures. This however is paid for when executing the resulting program: many calls of the generator and acceptor operations for an assignment between instances of similar or even identical representation are necessary. It is not possible to implement these cases more directly because the overloading of assignment is implicit; expanding them in line is undesirable, as shown in Chapter 5.

7.2.5 Abstract type framework

A distinctive feature of the transformative methodology is the bundle/capsule/vmo/bio language framework that allows multiple implementations of a capsule to be derived within one program. Depending on its position in the language spectrum and in its bundle, the capsule can specify both abstract data structure definition and implementation. The bundle collects capsules that represent one abstract definition, and its value manipulation operations and bundle interface operations relate each capsule to others in the bundle.

The bundle presents a notion of a "data type" that consists of an abstract definition and possibly many implementations. The preservation of assignment compatibility between different capsules in the bundle indicates that although they define different storage templates and specific operations at the machine level, the capsules are all logically of the one abstract type. Their simultaneous presence in the program contrasts with the notion of a data type as a single instance of one implementation paired with the type definition, as
found in the capsule languages, Modula 2, or Ada.

The view of a data type presented by the BSL bundle is that it consists of an equivalence class characterized by its abstraction and containing all possible implementations. A particular program chooses a subset of the equivalence class for each of its abstract types. This notion is not shown within a single program in the capsule languages, where its effect is obtained only by the extra-language means of rewriting the program or relinking the program with different library modules. The "schemes" of Mitchell and Wegbreit [1977] contain a related concept of abstract data type and many implementation capsules in one program, but there the relationship between an abstraction and its implementations does not explicitly define the abstraction and does not include assignment compatibility between implementations: the bundle is a stronger encapsulation of an abstract type's representations than the scheme.

A further point arises from this abstract framework, concerning the interaction between the language design and the transformation system. We have observed in Chapter 5 and shown in Chapter 6 that the bundle interface operations that support the framework of interfacing capsules to their bundle need no special treatment from the transformations. This is a measure of the framework's robustness: we would be much less satisfied if mechanisms to tie bundle members into a single type required exceptional treatment from the derivation operators.
7.3 Future work

7.3.1 Extending BSL

The language BSL as used in this thesis is a mini-language. Obvious extensions would be to add further type constructors at the abstract end, and at the concrete end to approach the machine more closely. Adding abstract constructors such as sets and mappings would be necessary to make Koala's abstractions more useful, and could be done without difficulty.

Adding more concrete data types to Maclan presents more of a problem. A language approaching a linear, untyped machine storage model cannot easily maintain the object-oriented model at present shared by Koala and Maclan, but it would be interesting to see how much closer an extended BSL could go. For a full range of representations we need a language of a different kind again, addressing machine-specific fine structure and non-linearities: word size, access to bit-fields, bytes, and multiple words, memory interleaving, paging, and implementation. For the first extension we would suggest a broadening of the spectrum of BSL, adding another well-defined data space beyond Maclan; but BSL could not accommodate this last set of structures in anything like its present form.

An extension in a different direction is to consider allowing objects to change representation during program execution, either dynamically or under some extended form of scoping for representations. Dynamic change of representation appears frequently in ma-
ally written programs and is a significant aspect of practical knowledge concerning program equivalence. But a strongly typed language gives a single type to a variable throughout its scope, and the transformation system depends on strong typing. Completely dynamic changing could not be efficiently done in the present Maclan's model of the machine data space, because in general it would require the object's type to be defined as a union of the representations and the object to be discriminated at run time for every application of its operations. But if the object's representation were kept constant in fixed regions of the program smaller than the object's textual scope, such a static variation in representation would have an efficient realization. The transformation system and the transformations themselves would need to modify their concept of a type to accommodate this. Further work in programming language design is needed here to tie a representation to something other than a name denoting an abstract entity, while maintaining the security of strong typing.

7.3.2 Adding transformations

The transformation system can readily accommodate more transformations. It is only necessary that they fit the classification of structural, binding, or auxiliary. Some general principles for designing new transformations have been discussed in section 6.6. Given the necessary BSL extensions we would expect both "packed" and "sparse" representations to be achieved by new structural transformations; a form of evert for trees is possible without language extensions, and again this would be a structural transformation.
We have achieved significant results in deriving data structure representations, using a relatively small library of transformations. This was possible because representations were specified generatively by combinations of transformations. The library necessary to represent the larger range of abstractions and representations mentioned could therefore be expected to be manageably small.

However the power of any additional transformations is limited by the language in which they are expressed, Xlan. It is apparent from the transformations studied here (as demonstrated in Chapter 6) and from the classes of transformations achieved in the work described in Chapter 2 that more powerful transformations require more powerful reasoning. They need to manipulate attributes of program constituents, logical properties, and arithmetic properties, and for this Xlan is inadequate in its present form. Any further work in this area would require that Xlan be considerably extended or replaced by a more powerful language.

7.3.3 Transforming other languages

It would clearly be useful to be able to apply these data structure transformations to programs in other languages. But as they stand, the transformations cannot be applied to any programming language other than BSL, for two reasons.

Firstly, the transformations are expressed directly as operations on the syntactic forms of that language, i.e. on its programs' surface structures. Transformations applicable to more than one language
would need to operate at the semantic level, on the programs' deep structures, expressed in some common notation. At present few programming languages possess formally described semantics, and many different notations are employed. But given some common basis, the transformations described here could be re-expressed and as well as being applied to represent Koala sequences as Maclan sequences (or as structs and reference variables), might be applied to represent Pascal files as Pascal arrays (or as records and pointer variables).

Secondly, the direction of the transformations is from the abstract to the concrete. The problem of transforming between different concrete representations is a separate one, which would require identifying an abstraction in the first concrete form before our transformations could be applied to derive the second concrete form from this abstraction. In languages without a sufficient range of abstractions the description of an abstract type can only be in the form of one of its representations, and such languages therefore need transformations of the concrete to concrete class. Most common languages are in this category; in the Algol family only the Pascal file is sufficiently abstract to provide an opportunity for transformations of our kind, as mentioned above. A language with sufficient abstract types would also need sufficient representation types, but there exist few languages with such a broad spectrum.
7.3.4 A fully automatic system

Incorporating this library of transformations into a fully automatic system would be possible in principle. It would require the analysis of the subject program to determine which representations would be possible, and general analysis of what performance in space and time would be expected of each. The former could be done by considering all possible transformation sequences for all bindable constructs in the program, eliminating sequences that violated a transformation's restrictions or did not tend towards a representation. Analysis of performance would require detailed consideration of the various forms of representation and of their interacting operations defined by vmos, and as shown by Low [1974], we would have to add externally collected information on frequencies of data values and specific space and time costing factors for a given program. But because a type's instances could be partitioned more finely than in Low's system (by sharegroup rather than by their use of assignments) we would expect such an automated system to be more expensive than his. Some other heuristic to find the best combination of representations would also be needed, because there are more interactions between representations.

7.3.5 Program proofs

Another area for further study is the effect of such transformations on program proving techniques. If a library of transformations could be proved to be correct then a proof of an abstract program
would also be sufficient to prove any of its implementations that were derived with that library. This would greatly simplify the task of proving concrete programs.

Even without a formal proof of the transformations themselves, they might usefully be extended so that a proof developed for the abstract program could be transformed in parallel with transformations of the program. It could then be checked that the resulting text was both a valid proof and corresponded to the posttrans program; a proof of the representation would thereby be obtained much more easily than by hand, although not completely automatically.

7.4 Conclusion

In summary, data structure refinement by transformation and data independence with strong typing extend our tools for developing data structures in programs. It is apparent that our combination of programming language forms and relatively simple transformations is an effective way of capturing and applying knowledge about programming.
APPENDIX 1: SYNTAX DESCRIPTION

The notation used for syntax description in Chapter 3 and as the basis of Xlan type definitions in Chapter 4 is a further extension of the Extended BNF described by Wirth [1977c]. The extension is in the description of repetition, and a different meaning is attached to an instance of simple repetition. A frequent occurrence in the syntax of programming languages is of the form "a list of Xs separated by Ys", which is expressed as a pair of repetition (curly) brackets enclosing two expressions separated by a slash. The first expression describes the repeated construct, the second the construct lying between each pair of instances of the first. The change to the meaning of a repetition is in the minimum number of occurrences specified by an unadorned curly bracket: here that minimum is one, rather than zero. A repetition of zero or more occurrences is described by a curly bracket pair with trailing asterisk. Square brackets denote an optionality, round brackets are for grouping only, a vertical bar separates alternatives (and has lower priority than textual concatenation which denotes actual concatenation). Single quotes surround literals and are written twice within literals. Identifiers may contain underscore characters. An example of the use of this syntax description is a description of itself, as follows.
syntax = { production / ";" } . . ;
production = identifier "=" expression ;
expression = { term / "|" } ;
term = { factor } ;
factor = identifier | literal | 
'(" expression ")' | 
'[" expression "]' | 
'{" expression [ "/" expression ] "}' [ "*" ] ;
literal = """" { character } """".

The lexical question of the distinction between, say, an eight-character identifier and two adjacent four-character identifier factors is not answered by this description. Blanks and new-line characters are ignored in the syntax description except within literals and to the extent that they distinguish a pair of textually adjacent identifiers from a long singleton identifier.
This is the complete syntax of the Broad Spectrum Language. The method of description is that defined in Appendix 1, with the added feature of distinguishing productions and constituents for the various languages in the broad spectrum.

Constructs common to Koala and Maclan are given no distinguishing mark. Koala constructs that do not appear in Maclan (except within ABSTRACT capsule definitions) are marked with a following "@K"; similarly those of Maclan with "@M". Constructs from the intermediate portion of the spectrum that appear in neither Maclan nor Koala are marked "@I". The grammar of the language at a particular part of the spectrum can be derived from the one given here by textually deleting constructs with inappropriate marks, and simultaneously deleting any alternation symbol ( '|' ) associated with the deleted constructs.

Where the non-terminal being defined in a production is marked, the whole production may be deleted. With the "variable_declaration" syntax, for example, the Koala form is obtained by deleting " | := " from the definition:

(for Koala)
variable_declaration = 'var' simple_formalpara [ '<=' expression ].

Comments are strings of any characters between '(*' and '*)'.
(a) Declarations

program = "begin" block "end";

block = { inblockdec ";" }* compound_statement;

inblockdec = commondec | variable_declaration;

commondec = operation_declaration | capsule_declaration |
bundle_declaration | share_declaration;

incapsuledec = commondec | state_declaration;

variable declaration = "var" simple_formalpara 
[ ("<-@K | :-@M) expression ];

capsule declaration = "capsule" (identifier | abstract@M) 
"=" ("{ incapsuledec / ';' }" "") block;

operation declaration = identifier "="
[ "self" [all_shareability]@K ]
[ "traversing" ]@K ( "proc" | "func" )
[ formal_trav_pack ]@K [ accepting_formalpara_pack ]
[ ":" [possibly]@K accepting_formaltype ]";" block;

share_declaration = "sharegroup" identifier "of" formaltype;

state_declaration = "state" ":" (structdec | seqdec |
treedec @K | formaltype);

simple_formalpara = [possibly]@K id_list
":=" simple_formaltype ["=" expression]@M;

accepting_formalpara = [possibly]@K id_list
":=" accepting_formaltype [="=" expression]@M;

formaltype = type identifier | subrange | union @K |
unprotrav @I T reftype @M | enumeration |
overlayunion @M;

simple_formaltype = formaltype [ simple_shareability ];

accepting_formaltype = formaltype [ accepting_shareability ];

simple_formalpara_pack = "(" { simple_formalpara / ";" } ");

accepting_formalpara_pack = "(" {accepting_formalpara / ";" } ");

simple_shareability = "shareable" sharegroup_identifier;

accepting_shareability @K = simple_shareability | "any";

all_shareability @K = accepting_shareability | "gen";
\[ \text{reftype } @M = (\text{`refto` | `nrefto`) `(` `simple_formaltype ``)`) \; \]
\[ \text{subrange} = \text{constant } `'..` constant \; \]
\[ \text{unprotrav } @I = \text{`unprotected` `traversal` `over`} \; \text{object_expression} \; \]
\[ \text{union } @K = \text{`union` `(` `identifier` `:` `simple_formaltype` `/` `,` `)` ``)`} \; \]
\[ \text{id_list} = \{ \text{identifier} / `,` \} \; \]
\[ \text{enumeration} = `( `id_list `)` `;` \]
\[ \text{overlayunion } @M = \text{`overlay` `(` `formaltype` `/` `,` `)` `;)`} \; \]
\[ \text{structdec} = \text{`struct` `simple_formalpara_pack`} \; \]
\[ \text{seqdec} = \text{`sequence` `[`upto`]@K `integer_expression`] `of` `simple_formaltype` `(` `traversed` `trav_declist` `)` `@K` \; \]
\[ \text{treedec } @K = \text{`tree` `[`upto` `integer_expression]}
\text{`of` `simple_formaltype` `articulated` `id_list`}
\text{`(` `traversed` `trav_declist` `;` `parameter` `id_list`)`;}
\]
\[ \text{trav_declist} = \{ \text{identifier}
\text{`(` `traverser` `expression` I `finished` `)` I `formal_trav_pack` = `[` id_list `)` `;` `block`\}
\]
\[ \text{bundle declaration } @M = \text{`bundle` `identifier` =}
\text{`(` `inbundledec` `/` `,` `)` `);` \]
\[ \text{inbundledec } @M = \text{capsule_declaration} I \overloadopdec \; \]
\[ \text{overloadopdec } @M = (\text{`:=` I `:-` I `topcopy` I `totalcopy` I `:=` I `:-:` I `:-:`) `=` (`proc` I `func`) `generic_para_pack` `[` `;` `block` `;` `generict_formaltype` `;` `identifier` `from bundle` `[`any`]` `;` `generic_para_pack` `@M = `[` `identifier` `:` `generict_formaltype` `/` `,` `)` `;` `)`) \; \]

(b) Statements
\[ \text{statement} = \text{assignment} I \text{compound_statement} I \text{repetition} I \text{conditional} I \text{procedure_call} I \text{case_statement} I \text{shrink} @K I \text{grow} @K I \text{trav_update} @K I \text{discrimination} @K I \text{make_empty} @K I \text{yield_statement} \; \]
statement_list = { statement / ';' }* ;

compound_statement = "begin" { inblockdec ";'" }* @M
statement_list "end" ;

assignment = object_expression assignment_operator expression ;

assignment_operator = ':=' | ':=' | '<-' @K | 'yield' ;

repetition = "while" boolean_expression "do"
statement_list "enddo" ;

conditional = "if" boolean_expression "then" statement_list
[ "else" statement_list ] "endif" ;

procedure_call = operation_call [ 'of' object_expression ] ;

case_statement = "case" expression "in" { case_branch / ',' } [ "otherwise" statement_list ] "endcase" ;

case_branch = "when" constant "then" statement_list "endwhen" ;

shrink @K = "shrink" ("at" traverser_expr
[ "via" articulation_identifier ] | object_expression "at" integer_expression ) ;

grow @K = "grow" ( ("at" | "after") traverser_expr | object_expression ("at" | "after") integer_expression )
"with" expression [ "via" articulation_identifier ] ;

trav_update @K = simple_trav_update | artic Advance |
make coincide ;

discrimination @K = union discrimination |
existence_discrimination ;

make_empty @K = "make_empty" object_expression ;

yield_statement = "yield" expression ;

simple_trav_update @K = ( "reset" | "next" | "reroot" |
"finish" | "previous"@I ) simple_trav_expr ;

artic Advance @K = "advance" simple_trav_expr "by"
articulation_identifier ;

make_coincide @K = "make coincide"
( traverser_identifier@K | object_expression@I )
"with" traverser_expr ;
union discrimination @K = 'with' object_expression 'discriminate'
   { discrim_branch / ',' } [ 'otherwise' ~ statement_list ]
   'enddiscrim' ;

discrim_branch @K = 'when' [ identifier ':' ]
   [ 'topcopy' | 'totalcopy' ] discriminant_identifier
   [ accepting_shareability ] 'then' statement_list 'endwhen' ;

existence discrimination @K = 'with' object_expression
   'discriminate' 'when' [ identifier ':' ] [ 'topcopy' | 'totalcopy' ]
   'exists' [ accepting_shareability ]
   'then' statement_list 'endwhen'
   [ 'otherwise' statement_list ] 'enddiscrim' ;

c) Expressions

expression =
   [ discriminant_identifier ':' ] @K ( binary_expr | factor ) ;

binary_expr = factor operator factor | coincides @K ;

operator = relational_operator | 'and' | 'cand' | 'or' | 'cor' | '*' | '/' | 'mod' | 'div' | '+' | '-' ;

relational_operator = '=' | '<>' | '<' | '>' | '<=' | '>'=
   | ':=-:' | '<->' @K ;

factor = constant | object_expression | brackexpr |
   notfact | finished_query @K | size_query @K | existence @K |
   isempty @K | reffer @M ;

constant = enumeration_identifier | number | 'nonex' @K |
   string_denotation 'nil' @M | 'dontcare' @M ;

number = { digit } [ '.' { digit } ] ;

string_denotation = "" { character }* '"' ;

coincides @K = traverser_identifier 'coincides' traverser_expr ;
brackexpr = '( expression ')' ;

notfact = 'not' factor ;

finished_query @K = 'finished' traverser_expr ;

size_query @K = 'size' object_expression ;

existence @K = 'exists' object_expression ;
isempty @K = `is_empty` object_expression;

reffer @M = `refto` expression;

traverser_expr = (`first` | `last` | `root` | traverser_identifier)
{ `by` (`next` | articulation_identifier | `previous`@I) }*
[`of` object_expression];

simple_trav_expr = traverser_identifier `of` object_expression;

object_expression = selection | 
{ selection `of`}* ( selection | object_head )

object_head = `self` | new-generator | copier |
yield_expr | follower @M;

selection = (applied_index | selector_identifier |
function_call | detravexpr @K ) [`qua` formaltype] @M;

selector_identifier = field_identifier | variable_identifier |
formal_parameter_identifier;

applied_index = `[@ integer_expression `]`;

detravexpr @K = `(#` traverser_expr `#)`;

new_generator = `new` type_identifier actual_para_pack;

copier = (`topcopy` | `totalcopy`) (`expression `);

yield_expr = `yieldof` `(`` statement_list ``)`;

follower @M = `follow` `(`` reference_expression ``)`;

function_call = operation_call;

operation_call = [ type_identifier `$` ] operation_identifier
[trav_para_pack] @K actual_para_pack;

actual_para_pack = `(` `expression / `, `)* `)

trav_para_pack @K = `(` `[` trav_para / `,` ]` `)``

trav_para @K = traverser_expr | relative_trav;

relative_trav @K = `unchanged` | `reroot` |
{ `advancing` articulation_identifier }.
APPENDIX 3: XLAN DESCRIPTION OF BSL

This appendix contains the Xlan rule definitions used for the examples of transformation procedures in Chapter 4. It differs from the complete BSL syntax of Appendix 2 in a number of ways. The lexical differences are that the special symbols that would otherwise take the lexical form of an identifier are stipped with a preceding '%', and the names of rules are abbreviated.

The rules describe a subset of BSL omitting trees, subranges, unions and bundles. The rule for "object_expression" (objexpr) is modified for easier manipulation in Xlan, but produces the same text strings. The rule for "formalpara" allows only a single identifier, rather than a list. The context-sensitive shareability distinctions ("simple_", "accepting_") have been omitted.

actualparapack = `(` params: {expression / `,`}0 `)`;
advance = simpladvance;
applyindex = `][ indexpr : expression `];
asgrator = `<-` | `:-` | `:=` | `%yield`;
assignment = lhs: objexpr rator: asgrator rhs: expression;
binaryexpr = leftrand: factor rator: operator rightrand: factor;
block = declarations: [decs: {inblockdec / `;`} `;`] body: compoundstmt;
brackexpr = `(` braced: expression `)`;
bundledec = `%bundle` identifier `=` `[`
buncontent: {inbundledec / `;`} `]`;
bytrav = `%at` travex: travexpr;
capsuledec = "%capsule" capid: (identifier | "%abstract") =
(" capcontent: { incapsuledec / ';' } ")
;
cmpndobjex = tl: selnstep hdr: possofobjexpr;
coinctravs = "%makecoincide" lhs: identifier "%with" rhs: travexpr;
commondec = opndec | capsuledec | sharedec | bundledec;
compoundstmnt = "%begin" statlist: stmtlist "%end";
condl = "%if" cond: expression "%then" truebr: stmtlist
elsie: [ "%else" falsebr: stmtlist ] "%endif";
constant = "%true" | "%false" | number | "%nonex" | "%nil" ;
copier = copykind: ( "%totalcopy" | "%topcopy" )
(" copied: expression ")
;
detravexpr = "(#" traved: travexpr ")";
emptier = "%makeempty" vessel: objexpr;
existobj = "%exists" existential: objexpr;
expression = binaryexpr | factor;
factor = constant | objexpr | brackexpr | notfact | finishedtrav
| sizeobj | existobj | isempty | reffer;
finishedtrav = "%finished" finis: travexpr;
follower = "%follow" '( follexpr: factor )
;
formallist = {formalpara / ';'};
formalpara = fpposs: ["%possibly"] fpid: identifier ":":
ftype: formaltype;
formalparapack = "(" formals: formallist ")";
formaltype = defn: ( identifier | reftype)
shareability: [ "%shareable" shgrp: identifier ];
functionyield = "::" fpposs: ["%possibly"] ftype: formaltype;
grow = "%grow" grower: (indextoobj | travgrow)
"%with" wither: expression;
identlist = {identifier};
idlist = {identifier / ",", };
inblockdec = commondec | vardec;
inbundledec = capsuledec | overloadopdec;
incapsuledec = commondec | statedec | selfopndec;
indextoobj = obj: expression `%at` index: expression;
isempty = `%isempty` queried: objexpr;
newgen = `%new` newtype: identifier newfields: actualparapack;
normalhead = `%proc` | `%func`;
notfact = `%not` knotted: factor;
objexpr = objhead | cmpndobjex;
objhead = `%self` | newgen | copier | follower;
ofobjexpr = `%of` hdo: objexpr;
operator = `<->` | `:-:` | `=` | `<=` | `<` | `>=` | `>` | `%and`
| `%cand` | `%or` | `%cor` | `*` | `%div` | `%mod`
| `+` | `-`;
opncall = capsule: [typeid: identifier `$`]
calledop: identifier paras: actualparapack;
opndec = opnid: identifier `=`
head : (normalhead | traversinghead)
params: [ formalparapack ] funcyield : [functionyield] `;`
body: block;
possofobjexpr = [ ofobjexpr ];
proccall = objexpr;
program = `%begin` block `.`;
reffer = `%refto` reffexpr: expression;
reftype = kind: (`%refto` | `%nrefto`) (` to: formaltype `)`;
repet = `%while` whilcond: expression `%do`
whildo: stmtntlist `%enddo`;
selfhead = `%self` hasany: [`%any`] (`%proc` | `%func`);
selfopndec = opnid: identifier `=`
head: selfhead
params: [formalparapack] funcyield: [functionyield] `;`
body: block;
selnstep = step: (applyindex | identifier | opncall | detravexpr)
  queer: [ "%qua" explicitype: formaltype ];
seqdec = "%sequence" upto: ["%upto"] limit: [number] "%of"
  base: formaltype traversed: ["%traversed" traversers: idlist];
sharedec = "%sharegroup" grpid: identifier
  "%of" grptype: formaltype;
shrink = "%shrink" shrinker: (indextoobj | bytrav);
simpladvance = dirn: ( "%reset" | "%next" ) trav: travexpr;
sizeobj = "%size" sizeable: objexpr;
state = ( structdec | seqdec | formaltype);
stmntlist = {statement / ';'}O;
structdec = "%struct" fields: formalparapack ;
travexpr = traverser: ( "%first" | "%last" | identifier)
  articed: ["%by" artics: { ("%next" | identifier) / "%by" } ];
traversinghead = "%traversing" ("%proc"|"%func")
  ["travparams: idlist "];
travexpr = termtrav: termtravexpr hdr: possofobjexpr;
travgrow = posn: ( "%after" | "%before" | "%at" ) trav: travexpr;
vardec = "%var" varcore: formalpara
  initval: [ how: asgrator what: expression].
BIBLIOGRAPHY

ADA
Ichbiah et al. 1979.

ALGOL 60
Naur 1963.

ALGOL 68
Branquart et al. 1971
King 1975
Lindsey and van der Meulen 1973
Peck 1971
van Wijngaarden et al. 1976.

ALLEN, F.E. and COCKE, J. (1972)

ALPHARD
Wulf et al. 1976a
Wulf et al. 1976b.

ARSAC, J.J. (1979)
"Syntactic Source to Source Transforms and Program Manipulation",

BALZER, R.M. (1967)
pp. 535-44.

BAUER, F.L. (1979)
"Program development by stepwise transformations - the project CIP",
in F.L. Bauer and M. Broy (eds), Program Construction
(Lecture Notes in Computer Science vol. 69; Springer-Verlag, Berlin), pp. 236-66.

"Correctness of Data Representations: Pointers in High Level Languages",
Conf. on Data, pp. 115-19.

"The Composition of Semantics in Algol 68", Comm. ACM, vol. 14,

BRINCH HANSEN, P. (1973)


CLU
Liskov 1976
Liskov et al. 1977
Liskov and Zilles 1974
Scheifler 1977.


DEMBINSKI, P. and SCHWARTZ, R. (1977)
"The Taming of the Pointer", SIGPLAN Notices, vol. 12, no. 7
(July 1977), pp. 60-74.

DESJARDINS, P. (1974)
"Dynamic Data Structure Mapping", Software Pract. & Exp., vol. 4,

DIJKSTRA, E.W. (1972)
"Notes on Structured Programming", in O.-J. Dahl et al. (1972),
pp. 1-82.

A Discipline of Programming (Prentice-Hall, Englewood Cliffs).

DIJKSTRA, E.W. (1977)
Panel Discussion, Seminar on Programming Language Systems, Austra­
lian National University, February 1977, quoted in J.M. Bennett,
"One View of Computer Science Teaching", Australian Comp. Science

D’IMPERIO, M.E. (1969)
"Data Structures and their Representation in Storage", Annual

EARLEY, J. (1971)
14, no. 10 (October 1971), pp. 617-27.

EARLEY, J. (1973)
An Overview of the VERS-2 Project: High-Level Languages in
Automatic Programming (Memorandum ERL-M416, Electronics Research
Laboratory, Computer Science Division, University of California,

ELI
Bros gol 1971
Holloway et al. 1974
Wegbreit 1971
Wegbreit 1974a
Wegbreit 1974b.

ELLIS, D.J. (1974)
Semantics of data structures and references (MAC TR-134, M.I.T.,
Cambridge, Mass.) (available as NTIS PB 236 594).

EUCLID
Lampson et al. 1977
Welsh et al. 1977.


HORNING, J.J. (1976)  
"Some Desirable Properties of Data Abstraction Facilities",  
Conf. on Data, pp. 60-2.

"Preliminary ADA Reference Manual" and "Rationale for the  
Design of the ADA Programming Language", SIGPLAN Notices,  
vol. 14, no. 6 (June 1979), pts A and B.

JENSEN, K. and WIRTH, N. (1975)  
Pascal User Manual and Report (2nd ed.; Lecture Notes in  
Computer Science vol. 18; Springer-Verlag, Berlin).

JOHNSON, C.W. (1978)  
"References and/or efficiency in high level languages", Proc.  

JOHNSON, R.T. and MORRIS, J.B. (1976)  
"Abstract Data Types in the MODEL Programming Language", Conf. on  
Data, pp. 36-46.

"A Language Extension for Controlling Access to Shared Data",  
277-85.

JONES, N.D. and MUCHNIK, S.S. (1978)  
TEMPO: A Unified Treatment of Binding Time and Parameter Passing  
Concepts in Programming Languages (Lecture Notes in Computer  
Science vol. 66; Springer-Verlag, Berlin).

"Program Manipulation via an Efficient Production System",  

KIEBURTZ, R.B. (1976)  
"Programming without Pointer Variables", Conf. on Data,  
pp. 95-107.

KING, P.R. (ed.) (1975)  
Proceedings of an International Conference on Algol 68  
Implementation (Utilitas Mathematica Publishing Inc., University  
of Winnipeg, Manitoba).

KNUTH, D.E. (1973)  
The Art of Computer Programming, vol. 1: Fundamental Alg-  
orithms (2nd ed.; Addison-Wesley, Reading, Mass.).

"Structured Programming with goto Statements", Computing  
Surveys, vol. 6, no. 4 (December 1974), pp. 261-301.

KYNE, A.P. (1978)


LINDSEY, C.H. and MEULEN, S.G. van der (1973)
Informal Introduction to ALGOL 68 (revised and reprinted; North-Holland, Amsterdam).

LISKOV, B. (1976)
An Introduction to CLU (Computation Structures Group Memo­randum 136, M.I.T. Lab. for Computer Science, Cambridge, Mass.).


LISKOV, B. and ZILLES, S. (1975)

LOGRIPPO, L. (1975)

LOVEMAN, D.B. (1977)

Automatic Coding: Choice of Data Structures (Stanford AI Project Memorandum AIM-242, Stanford, Calif., August 1974) (available as NTIS AD/A-000 500).
LOW, J.R. (1978)

LOW, J. and ROVNER, P. (1976)


MEALY, G.H. (1967)

MEALY, G.H. (1974)

MESA
Geschke and Mitchell 1975
Geschke et al. 1977.

MEULEN, S.G. van der (1977)
"ALGOL 68 Might-have-beens", SIGPLAN Notices, vol. 12, no. 6 (June 1977), pp. 1-18.

MITCHELL, J.G. and WEBREIT, B. (1977)

MODEL
Johnson and Morris 1976
Morris 1976.

MODULA
Wirth 1977a
Wirth 1977b.

MODULA 2
Wirth 1978.

MORAN, T.P. (1973)

MORRIS, J.B. (1976)
Synopsis of Data Type Abstraction in Programming Languages (NTIS LA-UR-76-1750).
NAUR, P. (ed.) (1963)  

ORGANICK, E.I. (1973)  
Computer System Organization: The B5700/B6700 Series  

PARNAS, D.L. (1972)  

PARTSCH, H. and BROY, M. (1979)  
"Examples for change of types and object structures", in F.L. Bauer and M. Broy (eds), Program Construction (Lecture Notes in Computer Science vol. 69; Springer-Verlag, Berlin), pp. 421-45.

PASCAL  
Jensen and Wirth 1975  
Welsh et al. 1977  
Wirth 1971b.

PECK, J.E.L. (ed.) (1971)  
ALGOL 68 Implementation (North-Holland, Amsterdam).

PERLIS, A.J. (1967)  

Machine Oriented Higher Level Languages (North-Holland, Amsterdam).

PRATT, T.W. (1975)  
Programming Languages: Design and Implementation (Prentice-Hall, Englewood Cliffs).

PRITCHARD, P. (1979)  


SAIL  

SCHEIFLER, R.W. (1977)  
SCHWARTZ, J.T. (1975a)

SCHWARTZ, J.T. (1975b)

SCHWARTZ, J.T. (1975c)

SETL
Schwartz 1975c.

SHAW, M. (1976)
"Research Directions in Abstract Data Structures", Conf. on Data, pp. 66-8.

SIMULA 67
Dahl et al. 1970
Dahl et al. 1972

SPITZEN, J. and WEGBREIT, B. (1975)

STANDISH, T.A. (1967)


STANTON, R.B. (1976)

WALK, K. (1971)

WEGBREIT, B. (1971)

WEGBREIT, B. (1974a)
WEGBREIT, B. (1974b) 

WEGBREIT, B. (1976) 

WEGBREIT, B. and SPITZEN, J.M. (1976) 


WIRTH, N. (1971a) 

WIRTH, N. (1971b) 


WIRTH, N. (1977a) 

WIRTH, N. (1977b) 

WIRTH, N. (1977c) 

WIRTH, N. (1978) 

WISE, D.S. (1976) 
WULF, W.A. (1972)  

WULF, W.A., LONDON, R.L. and SHAW, M. (1976a)  

WULF, W.A., LONDON, R.L. and SHAW, M. (1976b)  
Abstraction and Verification in Alphard: Introduction to Language and Methodology (Information Sciences Institute Technical Report, University of Southern California, Los Angeles) (available as NTIS ADA 028365).