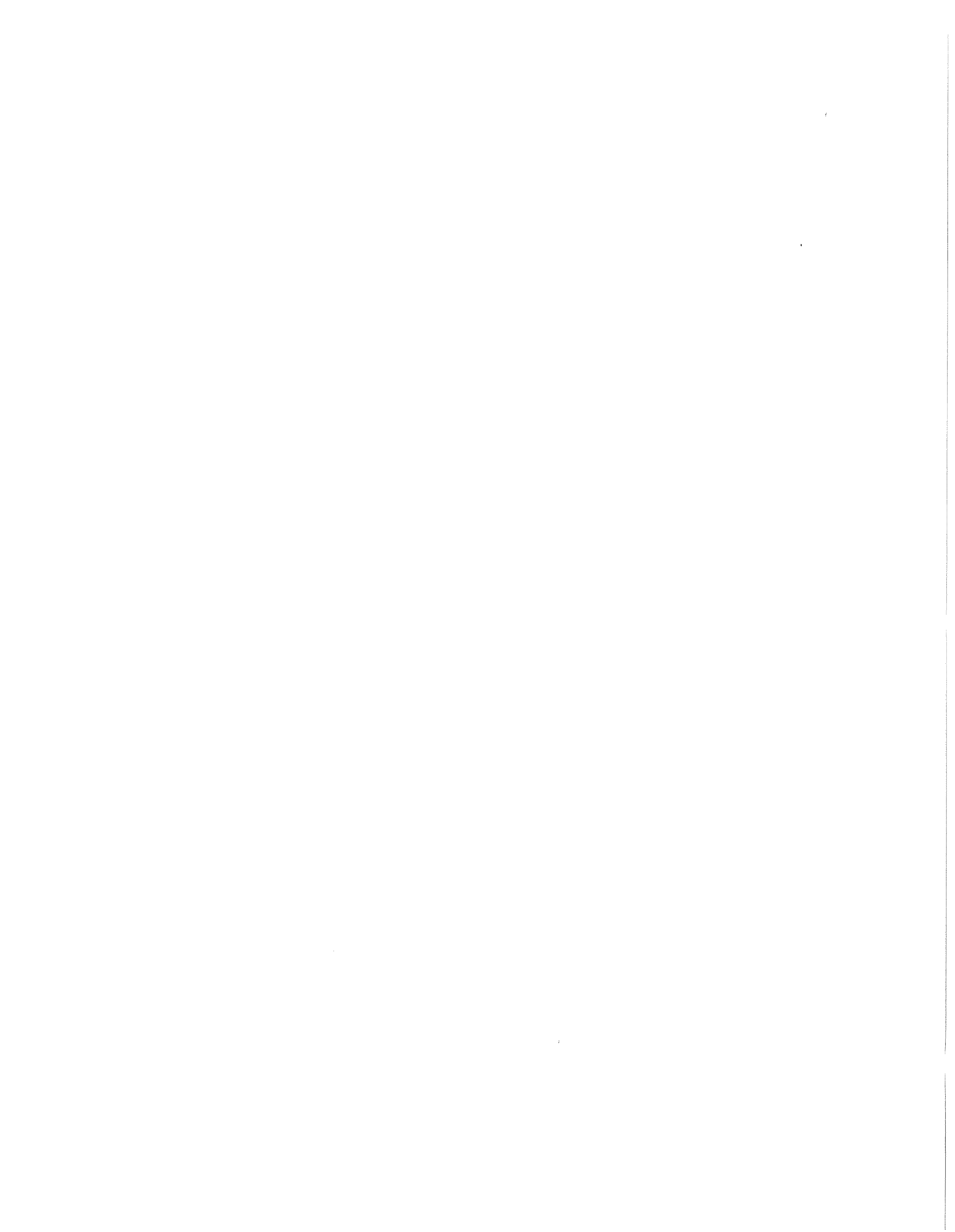
# E:
# A Persistent Systems Implementation Language

by

Joel Edward Richardson

Computer Sciences Technical Report #868
August 1989



# E:

# A PERSISTENT SYSTEMS IMPLEMENTATION LANGUAGE

by

JOEL EDWARD RICHARDSON

A Thesis submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1989

# ABSTRACT

This thesis presents the design and implementation of the E programming language. E is an extension of C++ designed for building systems that manage persistent objects, e.g. a database management system. Several aspects of this programming domain cause difficulty in conventional languages. For example, one must usually write the system code without knowing the types of entities to be manipulated. In addition, the entities themselves are persistent, outlasting the program that creates them. E addresses these and other problems through a judicious choice of language constructs that significantly ease the programmer's task. Being based on C++, E provides classes and inheritance. It then adds generator classes for defining generic container types, iterators for processing streams of values, and a persistent storage class for declaring a database as a collection of language objects. Through a series of refinements to an example program, we illustrate each of these language features.

One important benefit of having persistence in a language is that I/O is transparent to the programmer; a central problem in the implementation of such a language, therefore, is in designing techniques to manage I/O automatically and efficiently. This thesis presents a new technique called Compiled Item Faulting (CIF) that addresses the I/O problem. CIF combines static analysis and a minimum of run-time support to produce E programs that can access physical storage efficiently.

Finally, we present the results of an initial performance study. These results demonstrate that CIF can be very effective in producing high quality code. The study also points out certain areas where CIF's effectiveness is more limited. In the conclusions, we suggest several avenues for further performance improvements.

# ACKNOWLEDGEMENTS

It has been my great honor and good fortune to have had Mike Carey as an advisor these past five years. I believe that he is a model of what a Ph.D. advisor should be: imaginative, compassionate, exacting. He has my deepest appreciation and respect.

David DeWitt has also been central in helping me to complete this degree. He opened many doors that mere mortals could not, and he always had time to listen.

Thanks also go to the other members of my committee: Raghu Ramakrishnan, John Beetem, and especially, Marvin Solomon. Several discussions with Prof. Solomon contributed greatly to improving the design of E. I am grateful for his insightful comments, both as a member of the committee, and as one of the first E users.

Special thanks go to Larry Rowe for his early critiques (and encouragement) and for suggesting a unified design for the syntax of iterate loops.

Thanks also to the whole EXODUS group (past and present): Beau Shekita, Goetz Graefe, Dan Schuh, Todd Proebsting, Mike Zwilling, Scott Vandenberg, Srinivas, David Haight, Dave Martin, Dan Lieuwen, and Paul Bober. They have provided a fun, stimulating environment in which to discuss and develop ideas. I would like especially to thank Dan Schuh. He implemented iterators and generators in the E compiler, fixed innumerable bugs, and generally kept things running that otherwise wouldn't.

Sheryl Pomraning has, for many years, not only helped to make the CS Department run, she has made it a brighter place to work. Thanks, Sheryl.

Many friends have also helped to make Madison home over the years. Thank you Toby and Jane, Sandy and Gene, Lee and Bob, Tom and Denise, and of course, Mike and Carol.

I would like to thank my parents for years of guidance, encouragement and love. They never forced me to choose one path or another, but they always insisted that I give my best effort.

My appreciation and love go also to Betty and Lyman Farrar. Thank you both for the lesson of "moments."

Finally, I would like to thank my wife, Helen. While she was given the life of a "dissertator's widow," she gave in return her loving support, ready encouragement, deep understanding, and a beautiful daughter. Without Helen, this work would not have been completed; without Emily, this work might have been completed sooner, but life would have been far less delightful. Thanks also to P., L., I., E., M., and L. for being ever-ready with suggestions and offers to help.

# TABLE OF CONTENTS

ABSTRACT .......... ii
ACKNOWLEDGEMENTS .......... iii
TABLE OF CONTENTS .......... iv

**Chapter 1:** INTRODUCTION .......... 1

**1.1** EXTENSIBLE DATABASE SYSTEMS .......... 2
**1.2** EXODUS .......... 2
**1.3** THE E LANGUAGE .......... 3
**1.4** OUTLINE OF THESIS .......... 4

**Chapter 2:** A SURVEY OF RELATED WORK .......... 6

**2.1** DATABASE PROGRAMMING LANGUAGES .......... 6
**2.1.1** Pascal/R .......... 7
**2.1.2** Rigel .......... 7
**2.1.3** Plain .......... 8
**2.1.4** Theseus .......... 9
**2.1.5** Other DBPLs .......... 9
**2.2** CONCEPTUAL MODELLING LANGUAGES .......... 9
**2.2.1** Taxis .......... 9
**2.2.2** DIAL .......... 10
**2.2.3** Galileo .......... 11
**2.3** PERSISTENT LANGUAGES .......... 12
**2.3.1** PS-Algol .......... 12
**2.3.2** Napier88 .......... 13
**2.3.3** Avalon/C++ .......... 13
**2.3.4** O++ .......... 14
**2.4** OBJECT-ORIENTED DATABASE SYSTEMS .......... 14
**2.4.1** GemStone .......... 14
**2.4.2** Vbase .......... 14
**2.4.3** Orion .......... 15
**2.4.4** $O_2$ .......... 15
**2.5** RELATIONSHIP TO E .......... 16

**Chapter 3:** THE DESIGN OF E .......... 17

**3.1** C++ REVIEW .......... 17
**3.1.1** Classes .......... 17
**3.1.2** An Example .......... 18

3.1.3 Inheritance .......... 22
3.2 ITERATORS .......... 23
3.2.1 Iterators in E .......... 23
3.2.2 Flow of Control .......... 25
3.2.2.1 Advance .......... 25
3.2.2.2 Break .......... 26
3.2.3 A Recursive Iterator Example .......... 27
3.3 GENERATOR CLASSES .......... 29
3.3.1 Parameters to a Generator Class .......... 29
3.3.1.1 Class Parameters .......... 29
3.3.1.2 Constraints on Class Parameters .......... 30
3.3.1.3 Function Parameters .......... 31
3.3.1.4 Constant Parameters .......... 32
3.3.2 Class Name Scoping .......... 33
3.3.3 Nested Instantiations .......... 33
3.4 DB TYPES AND PERSISTENCE .......... 34
3.4.1 Database Types .......... 35
3.4.2 Persistent Objects .......... 35
3.4.3 Collections .......... 36
3.4.3.1 Creating Objects in a Collection .......... 37
3.4.3.2 Scanning Collections .......... 37
3.4.3.3 Destroying Objects and Collections .......... 38
3.4.4 The Binary Tree Example Revisited .......... 38
3.4.5 Implementing a Database Index .......... 42
3.5 TWO LANGUAGE DESIGN ISSUES .......... 43
3.5.1 Orthogonality .......... 43
3.5.2 Persistent Handles .......... 44

Chapter 4: COMPILER ORGANIZATION .......... 46

4.1 ARCHITECTURE OF THE COMPILER .......... 46
4.2 PROCESSING DECLARATIONS .......... 48
4.2.1 Representation of Objects and Pointers .......... 48
4.2.2 Type Declarations .......... 50
4.2.3 Data Declarations .......... 52
4.3 GENERATING CODE .......... 54
4.3.1 Two Machine Models .......... 54
4.3.1.1 A Persistent Virtual Memory .......... 54
4.3.1.2 A Load/Store Machine .......... 55
4.3.2 The Storage Manager Interface .......... 56
4.3.3 Overview of Code Generation .......... 58
4.4 OTHER IMPLEMENTATION ISSUES .......... 60
4.4.1 Constructors and Destructors .......... 60
4.4.2 Virtual Functions .......... 62

Chapter 5: CODE GENERATION .......... 64

5.1 PHASE I: INITIAL PIN SCHEDULING .......... 64
5.1.1 Identifying Common Subexpressions .......... 64
5.1.2 Items .......... 67
5.1.3 Initial Pin Scheduling .......... 69
5.1.3.1 Detecting Items to Pin .......... 70
5.1.3.2 Deciding Where to Pin Items .......... 72
5.2 PHASE IV: TRANSFORMING THE SYNTAX TREE .......... 74
5.2.1 The Functions genSprigs( ) and mungeTree( ) .......... 76
5.2.2 Generating Code Sprigs .......... 77
5.2.2.1 The Pinning Sprig .......... 78
5.2.2.2 The Unpinning Sprig .......... 79
5.2.2.3 The Reading Sprig .......... 79
5.2.2.4 The Writing Sprig .......... 80
5.2.3 Grafting the Sprigs .......... 80

Chapter 6: COMPILED ITEM FAULTING .......... 87

6.1 OVERVIEW .......... 87
6.1.1 Considerations .......... 87
6.1.2 Compiled Item Faulting .......... 89
6.2 IMPLEMENTATION .......... 91
6.2.1 Phase I Revisited .......... 91
6.2.2 Phase II: Ensuring Path Safety .......... 92
6.2.2.1 The Role of Alias Analysis .......... 93
6.2.2.2 The Current Implementation .......... 93
6.2.2.3 Handling Array Elements .......... 97
6.2.3 Phase III: Propagation and Coalescing .......... 97
6.2.3.1 Coalescing .......... 98
6.2.3.2 Propagation .......... 100
6.2.4 Phase IV Revisited .......... 105
6.2.4.1 Entering a Pinning Region .......... 106
6.2.4.2 Grafting Pin and Unpin Operations .......... 106
6.3 COPING WITH FINITE BUFFER SPACE .......... 107

Chapter 7: AN INITIAL PERFORMANCE STUDY .......... 110

7.1 ORGANIZATION .......... 110
7.2 THE EXPERIMENTS .......... 111
7.2.1 Test I: Graph Traversal .......... 111
7.2.2 Test II: Tree Traversal .......... 115
7.2.3 Test III: Relation Scan .......... 116
7.3 SUMMARY .......... 119

Chapter 8: CONCLUSIONS .......... 120

8.1 THESIS SUMMARY .......... 120
8.2 RETROSPECTIVE AND FUTURE WORK .......... 121

8.2.1 Language Design ..... 121
8.2.1.1 Db Types Versus Non-db Types ..... 121
8.2.1.2 Strings and Other Variable Size Types ..... 122
8.2.1.3 Generators and Inheritance ..... 122
8.2.2 Compiler Implementation ..... 123
8.2.2.1 Alias Analysis and Other Optimizations ..... 123
8.2.2.2 Other Performance Enhancements ..... 123
8.2.2.3 A Hybrid Approach ..... 124
8.2.2.4 Generators ..... 124
8.2.3 Programming Environment Support ..... 125
8.2.3.1 Classes as Objects ..... 125
8.2.3.2 Schema Evolution ..... 126
8.2.3.3 Debugging Support ..... 126
8.3 CONCLUSION ..... 127

**Chapter 9:** REFERENCES ..... 128

# CHAPTER 1

# INTRODUCTION

In the 1970's, the relational data model was a major focus of research in the database community. Today, relational database technology is well understood, a large number of relational systems are available in the market place, and they support the majority of traditional business applications relatively well. One of the foremost database problems of the 1980's and beyond is how to support classes of applications that are not well served by relational systems. For example, computer-aided design systems, scientific and statistical packages, image and voice data, and large, data-intensive AI applications all place demands on database systems that exceed the capabilities of relational technology. Such application classes differ from business-oriented systems in a variety of ways, including their data modeling needs, the types of operations of interest, and the storage structures and access methods required for their operations to be efficient.

This thesis presents the design and implementation of the E programming language. E was developed in the context of work on extensible database systems, an area of research that addresses the problems stated above. The original design goal of E was to provide a language targeted specifically for implementing database management systems (DBMSs) [Rich87]. That is, we wanted to provide programming constructs that would give the database implementor (DBI) high leverage in solving the particular problems of building a DBMS. The resulting language design [Rich89a] is suitable for building not only DBMSs, but persistent systems[1] in general.

One of the main features of the E language is the provision of typed, persistent objects, and one of the main implementation challenges has been the generation of code for expressions that manipulate these objects. There have been several versions of the E compiler and several different implementations of persistence. Version 1.0 of the compiler [Rich89b] was a demonstration of feasibility as well as a learning vehicle. Two alternate lines of development have evolved from that version. This thesis describes the implementation of version 2.2 which not only improves greatly over version 1.0 on the quality of generated code, but also provides a framework that allows further improvements to be made easily. This framework, called *compiled item-faulting*, along with the language design itself, are the two main contributions of this thesis. Before discussing the specific problems that E addresses, let us first develop the context out of which it grew.

---

[1]By "persistent system", we mean a software system that maintains and manipulates objects whose lifetimes may extend beyond any given program run.

## 1.1. EXTENSIBLE DATABASE SYSTEMS

In the mid-1980's, several research groups in the database community began to explore ideas for building "extensible database systems" [Bato88, Care89, Daya85, Rowe87, Schw86]. Although different groups have different notions of what "extensible" means, a common theme (broadly stated) is the desire to support more flexibility than is provided by traditional DBMSs for customizing the database system to the user's application. For example, one usually cannot add new attribute types to a commercial database system; in an extensible DBMS, it should be easy to augment the collection of "base" types in the system with new, user-defined types. Another direction for extensibility is the support of new data models and the new operations and index structures that they require. At the same time, we wish to avoid the severe performance penalty associated with providing such flexibility as a layer on top of a (nonextensible) relational system [Care85].

A range of different approaches has appeared in the literature. For example, POSTGRES [Rowe87], the successor to INGRES [Ston76], is an attempt to provide for the needs of next generation applications through extensions to the relational model; the POSTGRES data model includes procedures as a data type, abstract data types for attributes, triggers, and rule processing. The STARBURST system [Schw86] also retains the relational model, but provides a system architecture that may be extended by adding "attachments" [Lind87] in places with well-defined interfaces. The PROBE project [Daya85, Daya87], while departing from the relational model, is similar to POSTGRES in attempting to provide for all users' needs within a single system. PROBE offers an object-oriented data model and includes support for temporal and spatial data and for recursive query processing. Finally, GENESIS [Bato88] is an attempt to formalize database structures and processing in a variant of the functional data model [Ship81]. GENESIS provides extensibility in that modules defined under its data model may be easily plugged together.

## 1.2. EXODUS

The EXODUS Project at the University of Wisconsin has been exploring a toolkit approach to building and extending database systems. The driving philosophy for our research has been a belief that no one system is likely to meet the needs of all potential applications [Care86b]. Unlike the projects described above, EXODUS is not itself a database system but a set of powerful software tools to be used in building such systems. The bulk of the work to date has been to examine the factors that make DBMSs particularly difficult systems to build and to provide tools targeted to address those factors. The first operational DBMS built with the EXODUS tools was a small relational system that we demonstrated at SIGMOD '88 in Chicago. The latest effort to validate the EXODUS approach has become a research project in its own right. EXTRA, an advanced, object-oriented data model, and EXCESS, its associated query language, have recently been defined [Care88]. Current work includes defining a formal algebra for EXCESS queries and mapping EXTRA data structures into E constructs.

The first component of EXODUS to be designed and built was the EXODUS Storage Manager [Care86a]. The Storage Manager provides four main abstractions: objects, files, transactions, and buffer groups. We shall describe the first two very briefly here; more details will be given in Chapter 5. An *object* is an uninterpreted string of bytes of arbitrary size. A client of the Storage Manager manipulates data stored in an object by reading and writing subranges of bytes. A client may also grow or shrink an object. A Storage Manager *file* is a collection of objects. Unlike conventional operating system files, these files do not themselves store byte-addressible data; rather, they are provided as a mechanism both for object grouping and for efficient storage allocation. By providing a simple and uniform (yet powerful) abstraction, the Storage Manager is able to support a wide range of physical storage needs.

While some new systems built with EXODUS may be hard-wired for a particular application, others will be general purpose DBMSs, the latter being based on advanced data models having formal algebras. (EXTRA is one example.) Another EXODUS component, the Optimizer Generator [Grae87a, Grae87b], allows the DBI to produce a query optimizer tailored for a new algebra from a high-level description. This component generated the optimizer used in the SIGMOD '88 demonstration, and it will soon be used to generate an optimizer for the EXCESS query language.

## 1.3. THE E LANGUAGE

The third major component of EXODUS is the E language and its compiler. As we said earlier, the original motivation for E was to provide programming constructs tailored for implementing a DBMS. The traditional difficulty in building such a system derives from several factors. First, the DBI must write code whose primary task is to manipulate data on secondary storage. A significant portion of the total system code is therefore devoted to interacting with the storage layer, e.g. calling the buffer manager to read a record, and with transaction management, e.g. calling the lock manager to set a lock. Another difficulty for the DBI is that the code to implement operators (e.g. hash join) and access methods (e.g. B+trees) must be written independently of any data types on which they might operate. The DBI cannot know, for example, that a user will eventually want to build an index over a set of polygons, keyed on area. The need for generic access methods is particularly critical in an extensible system, since one of the stated goals is to be able to add new types easily. Finally, a DBMS must convert queries posed by end users into a form that the system can execute. This translation is greatly simplified if the basic operators can be written in a uniformly composable manner.

These and other considerations have led to the current language design. E is an extension of C++ [Stro86] providing generator classes, iterators, and persistent objects. C++ provided a good starting point with its class structuring features and its expanding popularity as a systems programming language. Generator (or generic) classes were added for their utility both in defining database container types, such as sets and indices, as well as in

expressing generic operators, such as select and join. Iterators were added as a useful programming construct in general, and as a mechanism for structuring database queries in particular. Both generators and iterators in E were inspired by those in CLU [Lisk77]. Persistence — the ability of a language object to survive from one program run to the next — was added because it is an essential attribute of database objects. The provision of persistent objects has several major benefits for the programmer [AtkM83]. First, mapping the application to the language becomes conceptually easier since there is no longer a semantic gap between the objects in the language and the objects in the database. Second, this mapping is also easier to implement, since we can manipulate the database objects naturally, via expressions in the language.

E thus represents a synthesis of ideas and advances from both the programming language and database communities. While E is certainly not the first language to provide some form of persistence, it is distinguished from its predecessors in being a systems-level implementation language rather than a modelling or prototyping language. Also, as we have mentioned, the implementation of E provides a new mechanism for managing I/O at run-time.

## 1.4. OUTLINE OF THESIS

Over the past decade, there has been considerable interest in the synthesis of database systems and programming languages. From the very early work on Pascal/R [Schm77] to the most recent of the persistent language designs, the area has remained very active. Chapter 2 reviews of some of the more important language designs to emerge from this research.

Chapter 3 presents the E language design in detail. Beginning with a review of C++, this chapter then describes the main language additions: iterators, generators, and persistence. Iterative refinement of a binary tree example serves to motivate as well as to illustrate each new feature.

We devote the next three chapters to describing the compiler implementation. Chapter 4 discusses the overall organization of the compiler and the handling of various kinds of declarations (e.g. of persistent objects). In Chapters 5 and 6, we describe code generation. Since a language that features persistent objects also hides I/O from the programmer, the system is responsible for making I/O efficient. Chapter 5 presents the basic concepts and organization of the code generator, while Chapter 6 introduces compiled item faulting (CIF), our mechanism for improving the performance of E programs.

In Chapter 7, we present the results of a small performance study. We ran these experiments as a preliminary evaluation of the quality of code generated by the E compiler. The results show that CIF is a viable approach to managing I/O in a persistent system, and they suggest ways that conventional compiler-optimization techniques can be applied to further improve performance.

Chapter 8 presents our conclusions. We evaluate E's successes and failures, both as a language design and as an implementation, and we give our recommendations for future work.

# CHAPTER 2

# A SURVEY OF RELATED WORK

In this chapter, we review some of the more significant work to emerge from efforts to integrate programming languages and database systems. We discuss representatives from several different categories: database languages, in which constructs from a particular data model (usually relational) have been merged with a programming language language (e.g. Pascal); conceptual modelling languages, in which very high level data descriptions subsume much of the semantic integrity checking usually performed by procedures; persistent languages, in which the complete type system of a base language is made available for defining and manipulating persistent objects; and object-oriented database systems (OODBs), in which modern, object-oriented type systems have replaced the relational model. The order of languages in the survey also corresponds roughly to the chronological order in which they appeared. While the survey is by no means exhaustive, it should give the reader a feeling for the range of designs that have appeared over the last decade. (An excellent comparison of database programming languages may be found in [AtkM87].)

## 2.1. DATABASE PROGRAMMING LANGUAGES

The majority of work in this area occurred in the late 1970's and early 1980's in response to the problems encountered in database application programming.[2] Prior to that time, application programs were traditionally written in an embedded language, i.e. one in which the data definition and query language of a DBMS is available inside a traditional programming language (e.g. embedded SQL [Date82, pp.145+]). There are several problems with the embedded language approach which may be summed up with the term "impedance mismatch" [Cope84]. For example, the programmer is burdened with notational awkwardness, since special symbols are required to flag the DBMS statements to a preprocessor. A more severe problem is that the type system available to the application programmer is effectively the lowest common denominator between the DBMS and the host language: integers, reals, and characters. And finally, the correspondence of types in an application program to the schema in the database is not well defined. Thus, one is forced to maintain this integrity manually.

---

[2]The term "application program" is quite generic and refers to any program written as an extra layer between the DBMS and the user. Application programs can be simple (e.g. a menu of "canned queries") or can be quite complicated (e.g. a forms-based interface).

### 2.1.1. Pascal/R

As we shall see, the typical database programming language (DBPL) combines the concepts of an existing data model with those of an existing programming language. For virtually all DBPLs, the chosen data model is the relational model, and Pascal is the most common starting language. This approach was pioneered by Pascal/R [Schm77], the first of the "integrated DBPLs" [AtkM87].

Pascal/R extends the type system of Pascal with two type constructors: **relation** and **database**, where a relation is of some record (tuple) type, and a database is a collection of named relations. In addition, a **relation** type declaration also specifies the primary key by naming the appropriate field(s) in the constituent record-type. Basic operators on relations include union, difference, and replacement, all based on primary keys; e.g. *R1 :- R2;* deletes each tuple in R1 whose primary key appears in R2.

Pascal/R provides both low- and high-level access to relations. The **foreach** loop, with an optional selection predicate, allows tuple-at-a-time processing. The general relation constructor, an expression whose value is a relation, provides the full power of the relational calculus, e.g. one Pascal/R statement can express any QUEL query.[3]

From today's standpoint, Pascal/R suffers several drawbacks. The first — actually an artifact of Pascal — is the limited ability to define abstract types. A more serious problem is that databases are mutually exclusive, and a program may access only one database at a time. Finally, iterators are restricted to operating on relations only, although the concept is useful in many other contexts. The next language addresses these and other issues.

### 2.1.2. Rigel

The Rigel language [Rowe79] was developed shortly after Pascal/R, and it benefited from newer developments in programming languages. Like Pascal/R, Rigel provides **relation** as a type constructor, where a relation consists of records with named fields, a subset of which forms a primary key. And likewise, Rigel also provides an expression syntax that includes the relational calculus, such that a single expression can implement a complex query.

Rigel improves on Pascal/R in several key areas, however. First, it incorporates the module concept as developed in [Wirt77]. In addition to providing a form of abstraction, modules provide a database structuring mechanism: each module may declare its own relation(s), and each program must import the modules (i.e. the part

[3] One additional feature listed in [AtkM87] is the ability to access a relation as an associative array, indexed on primary key. However, [Schm77] made no mention of this. The Plain language [Wass79] *does* have this capability.

of the database) that it needs.[4] Rigel provides another kind of abstraction with the **view** type constructor. In combination with the visibility rules of modules, this mechanism allows the definition of secure, alternative interfaces to the database. Finally, iterators[5] were generalized to a CLU-like [Lisk77] form in which the programmer may define arbitrary iterator procedures.

One restriction that Rigel shares with virtually all other database programming languages is in the type of entity that may be stored in a relation:

> In [no] case may any field of a relation tuple type be a pointer, union type, relation type, file type, view type, nor a structure of a dynamic size. [Rowe81, p.8]

Such restrictions are motivated, in part, by the implementation difficulties in trying to provide them, but more fundamentally by the semantics of the relational model. As E does not involve the relational model, it is not semantically bound by these restrictions; moreover, part of the interesting research has been in solving the associated implementation problems.

### 2.1.3. Plain

The language Plain [Wass79] is another Pascal extension based on the relational model. It was designed "to support the construction of interactive information systems", and it includes not only database oriented features, but also those tailored for user I/O, including extensive string handling and pattern manipulation. Like both Pascal/R and Rigel, Plain includes the type constructor **relation**. The language allows tuple-at-a-time processing through a form of iterator loop. Unlike Pascal/R or Rigel, however, Plain's extended syntax includes algebraic operators rather than calculus expressions. Thus, a Plain query is decomposed into a series of selects, projects, and joins.[6] The stated motivations for including an algebra instead of a calculus include consistency with the procedural nature of Pascal and ease of compilation. Finally, Plain is similar to Rigel in allowing the programmer to define abstract data types and in the structuring of the database such that each program imports the relations it needs.

Plain has several interesting features that distinguish it from other languages in its class. One is the provision for associative access to a relation. For example, assuming that DEPT is a relation of department tuples and that the department name is the primary key, then the statement

*print( DEPT[ "toy" ].floor );*

prints the floor number of the toy department. Another feature, a **marking**, provides a convenient and efficient mechanism for taking snapshots of relations and for holding temporary results of queries.

---

[4]As later examples will show, E takes a similar approach.

[5]Rigel uses the term "generator".

[6]Only **join** is actually an operator in the language. Plain's **where** clause and attribute lists implement, respectively, select and project.

### 2.1.4. Theseus

One more effort seeking to integrate database concepts with an existing programming language (Euclid [Lamp77]) is Theseus [Shop79]. Like all the above languages, Theseus provides a **relation** type constructor with associated primitive operations. Although these operations are not strictly those of the relational algebra, the paper demonstrates that Theseus is relationally complete.

One feature which does distinguish this language is that the elements stored in relations are **a-sets** rather than simple records. An a-set is a set of (name, value) pairs, and behaves very much like a property list in LISP. However, since names are declared and are not first-class objects, name expressions can be statically type-checked. A-sets are intended to subsume records, parameter lists, and messages with a single mechanism and to "move the relational database in the direction of ... 'knowledge bases.'" [Shop79, p.495]. Whether these goals were actually achieved is unclear; the language design was incomplete and the paper discussed a number of possible extensions.

### 2.1.5. Other DBPLs

The above set of DBPLs is by no means exhaustive. Ada has been used as a starting point for database extensions in at least two ways: AdaRel [Horo83] was proposed as a relational extension, with a flavor similar to the Pascal-based extensions. Adaplex [Smit83] is a synthesis of Ada with the DAPLEX functional data model [Ship81]. An approach quite different from any of the DBPLs mentioned so far was taken in Aldat [Merr85], which extended the relational algebra with limited procedural constructs.

## 2.2. CONCEPTUAL MODELLING LANGUAGES

The DBPLs of the previous section have one important feature in common: each is a synthesis of an existing programming language with *relational* database concepts. The languages described in this section are all original designs (as opposed to extensions of existing languages), and they provide, at once, more flexible data models and more structured programming environments.

### 2.2.1. Taxis

One of the more distinctive languages to emerge from database programming efforts is TAXIS [Mylo80, Nixo87], for it was the first DBPL to provide inheritance as a modelling tool [AtkM87]. The use of inheritance is, in fact, quite pervasive in TAXIS; not only are the users' data arranged in a class hierarchy, but *all* other components of the database system (e.g. transactions) are part of the *same* hierarchy. The objects in a user's database are instances of user-defined classes; every instance of a class has all of the properties named in the class definition. Classes are themselves instances of metaclasses, allowing one to define class properties (e.g. current number of instances).

The TAXIS kernel comprises a set of predefined metaclasses arranged in a hierarchy. To create a database application, one defines new classes and class instances. To define the schema, one creates instances (e.g. DEPT) of the predefined metaclass VARIABLE-CLASS[7]; the definition of DEPT specifies the properties (attributes) that all department instances will have. To write an application program, one defines an instance (e.g. HIRE-EMP) of the metaclass TRANSACTION-CLASS. This means that each program is actually a class definition; the "attributes" of a transaction instance include the actions to be performed, a parameter list, local variables, and the return value. An instance of this class is an executing program, called an ***execution instance***. Finally, an attribute of a variable class object may be a transaction class, effectively providing support for virtual fields.[8]

TAXIS provides class hierarchies via the IS-A construct, a specialization hierarchy with multiple, structural inheritance. Thus one may define the classes STUDENT and EMPLOYEE and then define STU-EMP such that every STU-EMP object has all the properties of both students and employees. One important point to note about classes and the IS-A hierarchy is the semantics of extents, i.e. the sets of existing objects of a given type. In TAXIS, each class has one implicitly associated extent, and the IS-A relationship implies extensional containment as well as structural inheritance. That is, each STU-EMP instance is included in both the STUDENT and the EMPLOYEE extents.

### 2.2.2. DIAL

The language DIAL [Hamm80] is, like TAXIS, an attempt to facilitate the construction of database applications through the use of very high level constructs within a tightly structured environment. One major emphasis in DIAL is data description. In particular, many of the integrity constraints associated with an application are expressed declaratively; the use of procedures is definitely secondary. The second emphasis is on user interaction. DIAL provides a very high level, forms-oriented interface for use by all applications. Forms look very similar to normal entities — they are objects with attributes — and the programmer may "code" rather complex interactions with a user by declaring properties for the form, including, for example, user prompts and integrity constraints on input values. Again, the goal is to subsume with declarations a large part of the procedural programming involved in developing an application.

The data model provided by DIAL comprises entities and entity classes, where all entities in a given class share the same structure (attributes). New classes may be derived from existing ones via specialization. As in TAXIS, the notion of a class encompasses both type information and physical extent. Furthermore, membership in

---

[7]Of the several predefined metaclasses in TAXIS, VARIABLE-CLASS is closest to the usual concept of a relation. A presentation of the full range of features in TAXIS is beyond the scope of this survey.

[8]A virtual field (or attribute) is one whose value is computed at the time of access.

a derived class may be defined to be automatic, based on satisfying a user-defined predicate; e.g. a PERSON entity automatically becomes an ADULT (a derived class of PERSON) when the age becomes 18. The DIAL run-time system is responsible for ensuring that this PERSON now also appears in the set of ADULTs. Another interesting result of this design is that an entity may acquire (or lose) attributes as it dynamically satisfies (or fails) the predicates defining various derived classes; in the above example, a PERSON becoming an ADULT might acquire the new attribute *job title*. Another implication is that type checking often cannot be done at compile time, since class membership can change dynamically.

DIAL provides highly tailored constructs for manipulating classes and entities. For example, entities may be created and updated only within specially declared procedures whose actions are limited to a small set of predefined operations. The designers argue that most database application programming follows "common and frequently recurring patterns," and that the language should therefore address these needs with "problem-specific rather than general structures" [Hamm80, p.76]. The authors claim that in one application, the DIAL program was no more than 25% of the size of an equivalent program written in a conventional language.

### 2.2.3. Galileo

Classifying Galileo [Alba85] is slightly problematic. It is one of the more recent and certainly one of the most ambitious of the very high level DBPLs; however, due to its treatment of environments (to be discussed), Galileo can also be considered a persistent language. Intended as an interactive, conceptual modelling language, Galileo combines a number of advanced features. Like DIAL, Galileo is based on a semantic data model, and one goal is to provide a very powerful data description facility such that many semantic constraints of the application may be expressed declaratively. Galileo also provides abstract data types, subtypes with multiple (structural) inheritance, strong typing, and type inferencing.

As in Theseus, entities are viewed as sets of (name, value) pairs, although the semantics of names and values differ. Data in a Galileo database are grouped into classes[9], where a class is a "modifiable sequence" of entities, all of the same type. One interesting note is that, unlike TAXIS and DIAL, Galileo, in theory at least, separates the concept of type from that of class (physical extent); however, the syntax for defining classes forces one also to define a new type for each class.

One novel aspect of Galileo is its treatment of *environments*, which are mappings between names and definitions. Environments are used as a modularization mechanism, and the language provides numerous environment operators affording very fine control over the visibility of names. Using this mechanism, one can

---

[9]The term "class" has become one of the most overloaded words in the PL area (second only to "object"). A Galileo class is a set of instances; a C++ class is a type.

construct a database, provide alternate views, and grow the database incrementally. Environments can be nested, and the user gains access by "entering" the desired environment. Finally, there is a unique global environment in which all declared values automatically persist. Since there is no restriction on what an environment may contain, any value — e.g. simple values, relations, and other environments — may persist. As of this writing, the implementation of environments is still at the prototype stage and is based on a workspace load/save model [Orsi89].

## 2.3. PERSISTENT LANGUAGES

The languages in this section, inspired partially by the DBPLs, specifically explore and develop the concept of persistence. Of particular interest is the principle that persistence of a data object should be independent of the object's type and of how the object is used [AtkM83]. Such a feature is called *orthogonal persistence*.

### 2.3.1. PS-Algol

The first language to explore fully orthogonal persistence was PS-Algol [AtkM83]. Having observed that 30% of the code in a typical program simply moves data between disk and memory,[10] the authors extended an existing language, S-Algol [Morr82], with certain predefined procedures and a new run-time system allowing arbitrary structures to be preserved indefinitely. To preserve an object, one opens a database (named with a string value). The structure returned is called a "persistent name environment." The user then calls a procedure which inserts into this environment a pair comprising a pointer to the object and a user-defined name. The run-time garbage collector is then responsible for ensuring that the object and everything reachable from it are written to disk. Later, a program can open the database and request the object by name. A pointer is returned, which may then be dereferenced normally. A run-time address translation mechanism moves objects into memory as they are referenced.[11]

All objects in PS-Algol carry self-identifying type information. While this ensures that objects created by one program are properly type checked in another, there is a price: the added type information requires space, and type checking must be dynamic. Another cost factor concerns buffer space. Since the run-time system reads entire objects, applications requiring very large or very many objects may suffer poor performance. Thus, PS-Algol may be suitable only for small systems. However, the language is clearly distinguished for placing persistence in a general and uniform framework.

---

[10] This statistic includes I/O calls and code to pack and unpack data structures.

[11] We will describe this mechanism in more detail later in this chapter.

### 2.3.2. Napier88

The designers of PS-Algol have recently introduced a new language, Napier88 [AtkM85, Dear89], which builds upon their experience with the earlier language. Unlike its predecessor, Napier88 was designed from scratch and makes several significant advances in the concepts of persistence. In PS-Algol, the mechanism providing persistence is the "persistent name space," which is not strictly part of the language, but rather, is a user-defined data structure. In Napier88, the idea of a name space is central to the structure of the language itself.

A name space in Napier88 is like an environment in Galileo; it defines the names, types, and objects available during the compilation of a statement or declaration. While lacking the extensive set of operators that Galileo provides for environments, Napier88 does allow for dynamically adding and deleting names in an name space. There is one important difference from environments, however. Name spaces are first class objects in Napier88; the actual referencing environment for a particular section of code may be the result of a function call. Since the name space for a piece of code can also be statically bound, Napier88 has the very nice quality that static type checking and binding occurs when possible, while dynamic type checking and binding occurs when necessary.

The design of name spaces provides great flexibility in designing software systems. Essentially, the language allows a system designer to decide the tradeoff between flexibility and performance. A system primarily using dynamic binding is easy to evolve, while paying a higher overhead at run-time, than one in which static binding is predominant. Clearly, the design of extensible database systems could benefit from such a mechanism, although there are several problems to be addressed. One problem is that even a statically bound name space can itself change dynamically, perhaps losing names and objects. While such errors are detected, it is not clear if they can be handled in any structured manner. (This is an example of the schema evolution problem, later to be discussed at greater length.) Another issue is the implementation of persistence. Since Napier88 sits atop the same abstract machine layer as does PS-Algol, its performance may be insufficient for the needs of a high-performance database system. Static type checking, however, should give Napier88 a significant performance boost over PS-Algol.

### 2.3.3. Avalon/C++

Like E, the next two languages are both extensions to C++ that include long-lived data. Avalon/C++[12] [Herl87, Detl88], a language designed to support reliable distributed computing. This language utilizes the inheritance mechanism of C++ to allow programmers to design data types having customized synchronization and recovery properties. Persistence is then modeled as a set of objects encapsulated by a server; a server may recover the state of its objects after a crash. E differs from this approach in that persistent objects exist independently of any

---

[12] Avalon/C++ is not really a persistent language, but it does not belong in any of the other sections of this survey, either.

active process. E's goal is to provide transparent persistence for structuring databases and transparent I/O for manipulating them.

### 2.3.4. O++

In an interesting recent development, researchers at Bell Labs have proposed the language O++ [Agra89a, Agra89b] that seeks to blend both high-level and systems-level programming features. Like E, O++ is also an extension of C++ including persistence. However, O++ maintains type extents (one for each class), and it provides support for integrity constraints and triggers. Like most DBPLs, O++ also provides a form of iterator for expressing calculus-like queries over type extents; two variations of this looping construct allow for querying either the extent of a single type or the extents of a type and all of its subtypes. Finally, O++ provides a fix-point operator for expressing recursive queries. As of this writing, O++ is still a paper design, although work on a compiler is under way [Geha89].

## 2.4. OBJECT-ORIENTED DATABASE SYSTEMS

Object-oriented database systems (OODBS) are closely related to persistent languages in that they provide rich type systems, typed persistent objects, and general computation. The difference seems to be mostly in name and, perhaps, orientation of the inventors.[13] OODBSs have appeared recently, both in the literature and in the marketplace.

### 2.4.1. GemStone

GemStone [Cope84, Bretl89] is an OODBS based on the language Smalltalk [Gold83]. Like PS-Algol, GemStone bases its persistence on reachability, but in GemStone, there may be multiple roots of persistence. A persistent name space consists of a dictionary of <name,value> associations, where the name is a string, and the value is either an atomic value, e.g. the integer "10", or a reference to another object. Among its contributions, GemStone was the first OODBS to be implemented and the first to tackle the problem of indexing in the context of objects.

### 2.4.2. Vbase

Vbase [Andr87, Vbas87] is[14] a commercial product calling itself an "integrated object system." It seeks to

[13] There have been lively discussions at workshops in recent years over the question of what, if any, is the difference between a persistent programming language and an OODBS.

[14] The original Vbase is no longer being distributed. Ontologic recently reorganized and is now developing VBase+; it is to be based on C++ and reportedly will be very different from Vbase.

blend an OODBS with the C programming language. The system presents to the programmer two languages and their respective compilers: the type definition language, TDL, in which the one specifies classes and operations, and the C superset, COP, in which one writes methods to implement the operations. Application programs are also written in COP. In order to bind persistent names within a program, both the TDL and COP compilers require a database file name as a command line argument. A Vbase database implements a global persistent name space in which type names and instance names are resolved. It also supports a module construct, however, so that names within a module do not conflict with names at the global level. In the beta release (version 0.8), databases are self-contained and disjoint; a given database contains all of the types, methods, and instances needed for its applications, and there is no sharing between databases.

### 2.4.3. Orion

The Orion system [Bane87, Kim89] is another OODBS developed in recent years. Its basic data model provides classes with multiple inheritance, object identity, and message passing. Orion maintains implicit ***extents*** for each class; unlike GemStone and PS-Algol, only class (extent) names are persistent handles into the database. Orion provides two alternatives that name either the extent associated with a single class or the extents associated with a class and all its subclasses. Orion has been a rather broad-based research effort. One of its original contributions was addressing the problem of schema evolution.[15] Other contributions have included research into transaction management, locking protocols, and composite object support all in the context of an OODBS.

### 2.4.4. $O_2$

The $O_2$ system [Banc88, Lecl89] is another recently-developed OODB. It is similar to Vbase in that it attempts to integrate an object-oriented database system, $O_2$, with a superset of the C language, $CO_2$. (Actually, both Vbase and $O_2$ intend to support a set of languages. While only the C extension was ever implemented in Vbase, $O_2$ has so far been integrated with C ($CO_2$) and with Basic ($BasicO_2$).) Like Vbase, type definitions are written in one language, while methods are written in the C extension. In $O_2$, persistence is based on reachability, as it is in PS-Algol. However, like GemStone, every named object is a root of persistence; in addition, $O_2$ also provides class extents, but unlike Orion, an extent for a class exists only if the class's definition explicitly specifies one.

---

[15] We note that GemStone also supports schema evolution [Penn87].

## 2.5. RELATIONSHIP TO E

The common denominator between E and the languages surveyed in this chapter is the provision for typed, persistent objects. Such a language concept benefits the programmer by making applications involving persistent objects easier to think about and easier to implement. The differences of these languages from each other and from E derive from their intended use. The DBPLs and conceptual languages, targeted at developing end-user applications, integrate such database concepts as entity sets, views, and transactions with such programming concepts as strong typing and procedural abstraction. The conceptual languages provide even higher-level data models than the DBPLs and emphasize declarative specification of integrity constraints over procedural implementations. The persistent languages extend the type system of a base programming language to the realm of long-lived objects. The OODBSs, closely related to persistent languages, provide database systems having object-oriented data models and general computational power.

One feature distinguishing E from these other languages is that E is intended as a systems implementation language. While one can certainly write end-user applications in E, the choice of language constructs makes it more suited for ***implementing*** higher-level data models. For example, the efficient implementations of different data models are likely to require different storage structures, and the operators in the various models are likely to require quite general processing. E thus gives the programmer explicit control in defining the layout of persistent objects and in defining the procedures to manipulate them. E belongs firmly in the category of persistent languages, but differs from other members of that family in its model of persistence and in its implementation thereof. E's persistence, in combination with its other features, provides a blend of language constructs well suited to the task of building persistent systems.

# CHAPTER 3

# THE DESIGN OF E

This chapter, describes the E language design in detail. Through a series of refinements on a binary tree index example, we present each of the major language features. We use a binary tree, instead of a "real" database structure such as a B+tree, since it still illustrates the essential features of E while keeping the examples short enough for presentation; at the end of the chapter, we will outlines how a B+tree can be implemented. Since E is an extension of C++ [Stro86], we begin the chapter with a brief C++ review. The initial example is presented as a complete program. Next we describe iterators and discuss their use as a query structuring mechanism. We then show how iterators may be used to add scanning capabilities to the index example. The following section describes generic types in E, extending the index example into one that abstracts the type of keys stored in the index and the type of entities referenced. Then we discuss the features that make E a persistent language: database types, persistent variables, and collections. One last refinement of the index example shows how a binary tree can be made a persistent object in a database. The chapter closes with a discussion of several important issues germane to programming in E.

## 3.1. C++ REVIEW

### 3.1.1. Classes

E is an extension of C++, which is itself an extension of C [Kern78]. The essential concept in C++ is the *class*. A class defines a type, and its definition includes both the physical representation of any *instance* of the class as well as the operations that may be performed on an instance. Unlike the abstraction mechanisms provided in CLU [Lisk77] or Smalltalk [Gold83], a C++ class does not necessarily hide the physical representation of instances. It is up to the designer of a class to declare explicitly which members (data and function) are private and which are public.

In C++ parlance, objects comprising the representation of a class are called *data members*, and class operations are called *member functions* (a.k.a. methods). Member functions are always applied to a specific instance; within the function, any unqualified reference to a data member of the class is bound to that instance. The binding is realized through an implicit parameter, `this`, which is a pointer to the object on which the method was invoked. An unqualified reference to a member `x` of the class is equivalent to `this->x`.

### 3.1.2. An Example

The example in Figures 3.1 and 3.2 is a complete C++ definition for a very simple binary tree index. The basic operation of the tree is to map a key value to the address of an entity having that key. In this simple example, each tree node stores a floating point key and a pointer to the indexed entity along with pointers to its left and right subtrees. The implementation uses a pair of classes: one which defines the nodes in the tree and one which defines the tree itself. The node class is recursive, both in its representation (i.e., nodes point to nodes) and in its operations (i.e., search and insert are recursive methods). The tree class is a simple "wrapper" class encapsulating the nodes. Further refinements to this example will concentrate largely on the node class. In order to keep the example simple while still showing the major features, the tree is unbalanced, and we limit the operations on the tree to inserting and searching.

Figure 3.1 gives the definition of the class `binaryTreeNode`. The physical representation of each node in the tree follows the class heading. As noted, each node contains a floating point key value (`nodeKey`), a pointer[16] to the indexed entity (`entPtr`), and pointers to the left and right subtrees (`leftChild` and `rightChild`). By default, the members of a class (both data and function) are private, i.e. they are not visible to users of the class.

The keyword **public** introduces a set of member declarations that form the public interface to the class. The interface to `binaryTreeNode` comprises the methods `search` and `insert`, as well as one named `binaryTreeNode`. These member functions are elaborated following the class declaration. Let us first consider the function `binaryTreeNode`. In general, a member function whose name is the same as its class is called a *constructor* for that class.[17] Constructors initialize class instances; the `binaryTreeNode` constructor initializes all the fields of a newly created node. C++ guarantees that if a class has a constructor, then that constructor will be invoked automatically whenever an instance of the class is created (e.g. by coming into scope). If the constructor takes arguments, they must be supplied with the object's declaration as in the example

```
binaryTreeNode    aNode( 0.0, NULL );
```

which declares `aNode` as a `binaryTreeNode` instance with a key of zero and a null pointer.

Now let us consider `search`. This function is always invoked on a particular `binaryTreeNode` instance, e.g. `myNode.search( 1.414 )` or `myNodePtr->search( 3.14 )`, and `this`, a pointer to the instance, is always passed to the function implicitly. References within `search` to `binaryTreeNode` data

---

[16] In C++ (and in newer versions of C), a `void*` may legally point to any type of object.

[17] A class may have many constructors, and in general, C++ supports operator and function overloading. Although it is also supported in E, we not will discuss overloading here.

```
class binaryTreeNode
{
      float               nodeKey;
      void                *entPtr;
      binaryTreeNode      *leftChild;
      binaryTreeNode      *rightChild;
public:

      binaryTreeNode( float, void * );
      void * search( float );
      void insert( binaryTreeNode* );
};

binaryTreeNode::binaryTreeNode
( float insertKey, void * insertPtr )
{
      nodeKey = insertKey;
      entPtr = insertPtr;
      leftChild = rightChild = NULL;
}

void * binaryTreeNode::search( float searchKey )
{
      if( this == NULL )
            return NULL;
      else if( searchKey == nodeKey )
            return entPtr;
      else if( searchKey < nodeKey )
            return leftChild->search( searchKey );
      else
            return rightChild->search( searchKey );
}

void binaryTreeNode::insert( binaryTreeNode* newNode )
{
      if( newNode->nodeKey == this->nodeKey )
            return;  /* no duplicates allowed */
      else if( newNode->nodeKey < this->nodeKey )
            if( leftChild == NULL )
                  leftChild = newNode;
            else
                  leftChild->insert( newNode );
      else
            if( rightChild == NULL )
                  rightChild = newNode;
            else
                  rightChild->insert( newNode );
}
```

Class Definition for Binary Tree Nodes

Figure 3.1

members are implicitly bound to that node. For example, in the third line of the function body, the reference to `nodeKey` is equivalent to `this->nodeKey`. The search function, then, compares the node's key with the argument, `searchKey`, and either returns the node's entity pointer or recursively searches the appropriate subtree. Note that is it possible for `this` to be null inside a member function, and in fact, the search routine checks for this condition in order to terminate the recursion. In the sixth line, for example, if the node has no left child, then the recursive call will pass a null `this` pointer. Thus, if the search routine receives a null pointer, it immediately returns a null pointer, meaning the key was not found. a null pointer.

The `insert` member function takes a pointer to a new node which is to be inserted into the tree. It is assumed that this node has been initialized with its key and pointer values. The routine compares the key in the new node with the one in the current (`this`) node. If the new entry has a key value less then the current key, then the new node either becomes the left child (if there is none), or it is inserted recursively into the left child. If the new key is greater than the current key, then processing proceeds to the right. For this implementation, attempts to insert duplicate keys are simply ignored; the next section will remedy this shortcoming.

Figure 3.2 gives the definition of the `binaryTree` class. As we said above, this class is really a thin wrapper around the node class, and it is mainly used to start the recursion, e.g. in a search. The physical representation of a `binaryTree` is a pointer to the root node. Initially, this pointer is NULL (see the `binaryTree` constructor). To search the tree, we simply search the root node recursively. The `insert` member function contains an example of creating a node dynamically. The **new** operator returns a pointer to a node which has been allocated on the heap; since we are creating an instance of a class having a constructor, we have provided arguments. If the tree is empty, the new node immediately becomes the root. Otherwise, we pass the new node to the root, and the insert proceeds recursively.

Finally, Figure 3.3 shows a main program that uses a `binaryTree`. The program allows the user to keep a student database in which students are indexed by gpa. As mentioned earlier, our binary tree does not allow duplicates. Such a limitation is clearly unacceptable in a secondary index such as this, but we will soon address this problem. In any case, new students may be added, and existing students may be processed (in some unspecified way). To add a new student, we call the function `getNewStudent` which presumably interacts with the user in order to create a new student instance. The function returns a pointer to the instance along with the student's gpa. We then add the student to the binary tree index, `gpaIndex`, by invoking the insert method and supplying it with the gpa and the pointer to the student. Similarly, to "process" a student, we obtain a selected gpa from the user and then search for that gpa in the tree. If an entry is found with a matching key value, the search routine returns the corresponding student pointer; we then pass the pointer to the student processing routine.

```
class binaryTree
{
      binaryTreeNode    *root;
public:
      binaryTree();
      void * search( float );
      void insert( float, void * );
};

binaryTree::binaryTree()
{
      root = NULL;
}

void * binaryTree::search( float searchKey )
{
      return root->search( searchKey );
}

void binaryTree::insert
( float insertKey, void * insertPtr )
{
      binaryTreeNode * newNode;

      newNode = new binaryTreeNode( insertKey, insertPtr );

      if( root == NULL )
            root = newNode;
      else
            root->insert( newNode );
}
```

Class Definition for Binary Trees

Figure 3.2

```
class student
{
      /* ... */
};

binaryTree gpaIndex;     // declare an instance

main()
{
      student *    s;
      float        gpa;
      int          cmd;

      while( (cmd = getCommand()) != QUIT )
            switch( cmd )
            {
            case NEWSTUDENT:
                  getNewStudent( &gpa, &s );
                  gpaIndex.insert( gpa, s );
                  break;

            case PROCESS:
                  getGpa( &gpa );
                  s = (student*) gpaIndex.search( gpa );
                  if( s == NULL )
                        printf("No students with this gpa.");
                  else
                        processStudent( s );
                  break;
            }
}
```

Using a Binary Tree

### 3.1.3. Inheritance

Another reason that we chose C++ as a starting point for E is that it supports subtyping, or, in C++ terminology, class derivation. Given a class `A`, we may define a class `B` that is a subtype of `A` as follows:

```
class A { ... };
class B : public A { ... };
```

`A` is called the base class, and `B`, the derived class. `B` inherits both the representation of `A` as well as `A`'s member functions. The **public** keyword in this context specifies that public members of `A` are also public members of `B`; without this keyword, public members of `A` would become private members of `B`. `B` may declare additional data members and member functions, and it may override the member functions inherited from `A`.

Normally, the invocation of a class method on an object is statically bound. That is, if a member function `f` is invoked on a variable of class `A`, then the call is statically bound to `A`'s version of `f`. Function invocation can also be dynamically bound, however. If a member function `g` of a class `A` is declared ***virtual***, then invocation of

g on an object depends on the actual (run-time) type of that object, which may be a subtype of A. If that subtype has redefined g, then the subtype's version of g will be called; thus, each object responds to the invocation according to its type. Virtual functions provide the C++ programmer with late-binding of method calls, a central concept in object-oriented programming.

Although there are a great many details that we will not discuss here (see [Strou86]), we note that E supports all of the derivation constructs of C++ (including virtual functions), and it extends those constructs to the realm of **dbclasses** (to be discussed in Section 3.4). We further note that the E compiler is based on a version of the AT&T C++ compiler supporting only single inheritance. Version 2 of C++ now supports multiple inheritance [Stro87], and we plan to adopt that version soon.

### 3.2. ITERATORS

Iterators were inspired by CLU [Lisk77]. An *iterator* is a control abstraction comprising two cooperating agents, an iterator function (i-function) and an iterate loop (i-loop), that work together to process a sequence of values. The i-loop is a client of the i-function. It requests a value from the i-function, processes the value, and then requests the next value. From the client's point of view, the i-function is simply a "stream" from which to receive a sequence of values. The i-function produces the values in the stream one at a time by *yielding* a value to the client loop. Unlike a return from a normal function, when an i-function yields a value, it saves its local state so that it may resume execution when the next value is needed. Thus, an i-function can be viewed as a limited form of coroutine, one which may be invoked only within the context of an i-loop.

We chose to include iterators in E for several reasons. First, the ability to separate the production of a sequence of values from the processing of those values is a convenience generally, since a very common programming task involves processing such sequences. We have found iterators to be extremely useful in many diverse programming contexts. Second, E was originally conceived as a DBMS implementation language. Database systems are dataflow-intensive systems in which a large portion of the processing involves scanning and filtering streams of data; an iterator can implement such processing in a direct, natural way. E is not unique in recognizing the utility of iterators for database query processing [Schm77, Rowe79, OBri86], although E iterators are somewhat more general. Finally, iterators are easy to implement and are relatively inexpensive to use [AtkR78]; each invocation costs little more than a normal procedure call.

#### 3.2.1. Iterators in E

Let us now consider the details of iterators in E. Syntactically, an iterator function looks like a normal function, except that the keyword **iterator** precedes the return type, and the function body may contain **yield** statements. An i-function may take parameters of any type and may yield values of any type, that is, they may take

or yield any type that would be legal for a normal function. The code comprising the i-function body is arbitrary; an i-function may invoke other iterators, including itself (i.e. iterators may be recursive).

Consider the example in Figure 3.4. The purpose of the i-function `bigElements` is to yield the elements of an (unsorted) integer array that are greater than the average of all the elements. When `bigElements` is invoked, it first makes one pass through the array in order to compute the average. Then it makes a second pass, yielding each element that is larger than the average. At each yield point, `bigElements` suspends its execution while the client processes the element; when the client requests the next element, the i-function will resume after the yield point, i.e. it will continue onto the next iteration of the **for** loop. When the **for** loop terminates, and control "falls out" the bottom, the i-function also terminates. (An iterator may also terminate by executing a normal **return.**) Although this example shows only one **yield** statement, in general, an i-function may have many.

```
iterator  int bigElements( int * array,  int size )
{
        float sum = 0.0;
        float ave = 0.0;

        /* first compute the average */
        for( int i = 0; i < size; i++ )
                sum += array[ i ];
        ave = sum / size;

        /* now yield the big elements */
        for( i = 0; i < size; i++ )
                if( array[ i ] > ave )
                        yield  array[ i ];
}

main()
{
        int   A[ 10 ];

        /* Initialize A */
        ...

        /* Now find big elements. */
        iterate( int nextEl = bigElements( A, 10 ) )
                printf("%d ", nextEl);
}
```

A Simple Iterator Example

Figure 3.4

An iterate loop comprises the keyword **iterate**, followed by one or more i-function invocations in parentheses, followed by a statement which forms the loop body. Each invocation supplies actual arguments to an i-function, and it declares a variable to receive the yielded values. For example, the following i-loop activates the i-functions `f` and `g`, where the yielded types are `int` and `char`, respectively.

```
iterate( int x = f();  char y = g(); int z = f() )
{
        ...
}
```

Note that there are two simultaneous activations of `f`, one associated with `x` and one with `z`.

An i-function may be invoked *only* within the context of an i-loop. Figure 3.4 also shows a main program containing an i-loop that uses the `bigElements` iterator. After initializing the array `A`, control enters the loop, and the i-function is activated. When control returns to the loop, `nextEl` holds the first value of the sequence. After the loop body prints that value, control returns to `bigElements` if it is still active; if `bigElements` has terminated, then the loop also terminates, and control flows to the next statement in the program.

### 3.2.2. Flow of Control

In the example in Figure 3.4, the flow of control through an iterate loop is implicitly defined, and it follows the rules introduced in CLU [Lisk77]. That is, at loop entry, and at the top of the loop in each iteration, the i-function is resumed in order to obtain the next value. The number of iterations is determined by the i-function, i.e. the loop iterates until the i-function decides to terminate. In addition, a single i-function controls the loop. E provides for several variations on this theme, providing the programmer with more general control flow capabilities.

First, E allows multiple i-functions to be activated in parallel.[18] In this case, the default flow of control resumes *all* i-functions at the top of the loop; the order of resumption is undefined. The loop terminates when all i-functions have terminated. If some i-functions have terminated while others are still active, then the loop variable associated with the terminated i-function continues to retain its last yielded value. In order to allow the program to determine which i-functions have terminated, E provides a built-in function, `empty`, which may be applied to any i-loop variable; `empty( v )` returns 1 if the i-function activation associated with variable `v` has terminated, and 0 otherwise. We will see an example shortly.

#### 3.2.2.1. Advance

The default flow of control described above is too restrictive in certain cases. Consider an iterator that is supposed to merge two sorted streams of values, yielding a single sorted stream. Naturally, we wish to produce the

[18] This capability is distinct from having several active i-functions due to nesting of i-loops, which is also allowed.

sorted streams using iterators, and so the merge i-function is also a client of other i-functions. The default flow of control is inappropriate for this task. If we simply try

```
iterate( int val1 = stream1();   int val2 = stream2() )
{ ... }
```

then we will march down the streams in lock-step, and the loop body will have to buffer an arbitrary number of values (up to the entire sequence produced by one of the streams). If we try nesting, then we will repeat the entire inner i-loop for each element considered by the outer:

```
iterate( int val1 = stream1() )
    iterate( int val2 = stream2() )
    { ... }
```

Clearly, more flexible control is needed.

The **advance** statement was introduced in part to meet this need. As an example, consider

```
advance  val2;
```

where this statement appears within the context of either of the two **iterate** loops above. The effect of the statement is to resume the i-function activation associated with `val2`, in this case, `stream2()`. After the **advance** statement, `val2` has its new value. In its general form, **advance** may have a comma-separated list of variables; the i-function activation associated with each of the variables is resumed (in an unspecified order). If an **advance** statement is executed on any given pass through the body of an i-loop, then *no* default resumptions are carried out, i.e. if any i-functions are advanced, then those are the *only* i-functions advanced for that iteration.

Figure 3.5 shows how the **advance** statement and the **empty** function may be used to implement the merge example. When control enters the i-loop, two i-functions, `stream1` and `stream2`, are activated, and the loop variables, `val1` and `val2`, receive their initial values. We first test to see if either i-function has terminated, and if so, we simply yield the element from the other stream. The default flow of control will then advance the one active i-function until it is exhausted. If both i-functions are active, then we yield the smaller value and explicitly advance the i-function from which it came; the other i-function will not advance on that iteration. The loop terminates when both i-functions have terminated.

#### 3.2.2.2. Break

The merge example shows how the client loop can decide which i-function activation to resume on any given iteration. So far, though, loop termination has still been determined by the i-functions, i.e. the client iterates until all i-functions have terminated. Alternatively, a client may decide to **break** out of an i-loop; normally, this causes immediate termination of all active i-functions associated with that loop.

A given i-function may sometimes require explicit control over the termination sequence, however. It may, for example, need to release heap space or to perform other bookkeeping tasks. To handle such cases, we have

```
iterator int merge()
{
        iterate( int val1 = stream1();
                 int val2 = stream2() )
        {
                if( empty(val1) )
                        yield val2;
                else if( empty(val2) )
                        yield val1;
                else if( val1 < val2 )
                {
                        yield val1;
                        advance val1;
                }
                else
                {
                        yield val2;
                        advance val2;
                }
        }
}
```

Using the **advance** Statement

Figure 3.5

extended the yield statement syntax with an optional termination clause. This clause is a statement which is executed if, and only if, the client terminates the i-loop while the i-function is suspended at that yield point.[19] For example, suppose that an i-function has built some structure which it must deallocate before terminating, and suppose that the variable `p` points to the root of the structure. Then in the following example, if the client breaks after the i-function has yielded `x`, the (user-defined) `cleanUp` routine will be called before the i-function terminates:

```
yield x : cleanUp(p);
```

In the absence of this clause, the i-function is terminated automatically.

### 3.2.3. A Recursive Iterator Example

As a final example, Figure 3.6 modifies the binary tree implementation from the previous section so that it handles duplicate keys. We have amended the insert routine so that it no longer ignores a duplicate entry; instead, if it finds a match, it recursively inserts the new entry into the left subtree. Now, since we must be prepared to find many entries with the same key value, we have rewritten the tree search in Figure 3.6 as an iterator which yields a

[19] A more general exception handling facility would have been useful here.

```
iterator void * binaryTreeNode::search( float searchKey )
{
	if( this == NULL )
		return;

	if( searchKey <= nodeKey )
	{
		if( searchKey == nodeKey )
			yield entPtr;

		iterate( void * p = leftChild->search( searchKey ))
			yield p;
	}
	else
		iterate( void * p = rightChild->search( searchKey ))
			yield p;

}

void binaryTreeNode::insert( binaryTreeNode* newNode )
{
	if( newNode->nodeKey <= this->nodeKey )
		if( leftChild == NULL )
			leftChild = newNode;
		else
			leftChild->insert( newNode );
	else
		if( rightChild == NULL )
			rightChild = newNode;
		else
			rightChild->insert( newNode );
}
```

Allowing Duplicate Keys

Figure 3.6

sequence of pointers to each of the entities with matching keys. Furthermore, just as the original search routine was a recursive function, the new search routine is a recursive iterator.[20] We again ground the recursion by first checking if `this` is a null pointer. If so, then the iterator returns without having yielded any values; from the client's perspective, the iterate loop terminates immediately, and the loop body is never entered. Assuming that `this` is not null, we compare key values. If the search key is less than or equal to the current node's key, then we first check to see if the keys are, in fact, equal. If so, then the entry in the current node is yield to the level above. We then recursively search the left subtree; each entry so obtained is also yielded. If the search key is greater than

[20]The search routine is, of course, more efficient if coded without recursion. We show it this way to illustrate how recursive iterators may be used. It also makes for a shorter and more elegant solution.

the key in the current node, we simply search the right subtree.

At the top level, the client picks up the return values one-by-one. At any given point in the client loop, there is a chain of active i-functions corresponding to levels of the tree. We note that the client may break out of the loop before all duplicate entries have been yielded; this event triggers a cascading termination of all active i-functions. Although the yield statements in the search iterator do not contain termination clauses (since none are needed), any such clauses would be executed as described above, beginning with the deepest activation.

## 3.3. GENERATOR CLASSES

As mentioned in the introduction, one of the problems facing the DBI is that much of the system code for a DBMS must be written without knowledge of the types of objects that the code will manipulate. Traditionally, a DBMS had knowledge of a few basic attribute types "wired in." The basic operators and access methods could operate on any of these types, essentially by switching on the type of the attribute at hand. One obvious problem with this approach is that the set of basic types is fixed, and therefore the system is difficult to extend. Another problem is that in order to handle different record types, offset and length information must be passed explicitly to each routine. In addition, the programmer is responsible for interpreting untyped buffer pages. One of the original goals of E was to make such mechanical tasks implicit. We were inspired by generators [Lisk77] as providing an elegant solution to the problem.

### 3.3.1. Parameters to a Generator Class

#### 3.3.1.1. Class Parameters

A generator is a parameterized type, i.e. one that is defined in terms of one or more unknown (formal) types. A generator defines an infinite family of related types and provides a natural way of defining container classes. The classic example is the generic type `stack[ T ]`, which, given any element type `T`, defines the type of a stack of `T` elements. In the case of our binary tree, we can (and will) make it a generic class by introducing two type parameters: the type of the key and the type of the entity being indexed.

E introduces generators in the form of *generator classes*.[21] A generator class may have any number of class parameters; the formal class names may be used freely within the generator as data member types and as argument or return types for member functions and iterators. Figure 3.7, for example, shows the E definition of a (bounded)

[21] CLU also has generator procedures and iterators. Although E does not provide these, we note that the same effect can be achieved, albeit indirectly. Consider a generator class having no data members and having a public member function, `f`, such that the return and/or argument types of `f` involve class parameters. Then `f` is effectively a generic function. The reason this approach is indirect is that we must declare "dummy" instances of the instantiated class in order to invoke `f`.

generic stack class. Syntactically, a generator class has the form of a regular class, except that the formal parameters are specified in square brackets following the class name. The parameters themselves have the form of empty class declarations.[22] We shall omit showing the `stack` member functions, since the only notable feature is that `T` is used wherever the name of the element type is needed.

In order to use a generic class, we must first *instantiate* a specific class by supplying actual arguments to the generator. For example, assuming we have a (nongeneric) class `frame`, we can then define a type describing stacks of frames by:

```
class frameStack : stack[ frame ];
```

Given this definition, we can now declare and use `frameStack` instances. For example, the declarations

```
frameStack  S1;
frame       f;
```

specify that `S1` is an instance of `frameStack` and `f` is an instance of `frame`. We can then push `f` onto `S1`:

```
S1.push( f );
```

Attempting to push anything but a `frame` onto `S1` will be flagged as a type error at compile time.

#### 3.3.1.2. Constraints on Class Parameters

If a class parameter is specified with an empty body, as in the `stack` example, then there are no constraints on the actual type that may be used in an instantiation. We could, for example, define `intStack` to be `stack[ int ]`, even though `int` is not really a class. As in CLU, E allows the specification of constraints on instantiating types; constraints are specified by "fleshing out" the parameter class body with member function declarations. Only classes having member functions with the same names and type signatures can be used to instantiate the generic

```
class stack [ class T { } ]
{
        int     top;
        T       stk[ 100 ];
public:
                stack();
        T       pop();
        void    push( T );
};
```

A Generic Stack Class Definition

Figure 3.7

[22]We will shortly fill out these declarations in order to specify certain constraints on the formal parameters.

class.[23] Furthermore, within the generator, these member functions may be invoked on objects of the parameter type. For example, we mentioned earlier that the binary tree class can be made generic by introducing two type parameters, one of which is the key type. In order for a key type to be useful, however, we must be able to compare two key values to determine their ordering. One means of accomplishing this is to constrain the key type:

```
class binaryTreeNode
[
        class keyType
              { public: int compare( keyType* ); },

        class entityType { }
]
{ ... };
```

With this declaration, an actual class may be bound to `keyType` only if it has a public member function named `compare` that takes a `keyType` pointer (as well as the implicit `keyType*` parameter `this`) and returns an integer. Within the search routine, we can now compare keys as follows:

```
int cmpVal = searchKey.compare( &nodeKey );
if( cmpVal < 0 )
       { ... }
else if( cmpVal == 0 )
       { ... }
else
       { ... }
```

Of course, there is an implicit additional requirement that the integer returned by the compare function be less than, equal to, or greater than zero corresponding to the ordering of the two keys. Such semantic constraints cannot be expressed within the E type system, however.

### 3.3.1.3. Function Parameters

One shortcoming of the approach taken in the above example is that it is no longer possible to instantiate (directly) a tree with floating point keys since `float` is not a class with a compare routine. While it is a relatively simple matter to define wrapper classes around the fundamental types, there is also a potentially more serious disadvantage here. In the definition of a generator class, the names of class parameters are formal names. However, if a class is constrained to have a certain member function, the function's name is actual. In the example above, any class may instantiate `keyType` provided that it has a member function whose name is literally "compare" and that is has the appropriate type signature. While this may be useful in some contexts, in others it may be too restrictive. For example, we may have a preexisting class that does have a comparison routine, but the routine's name may not

[23]For completeness, E also supports constraints specifying data members, although it is not clear how useful such constraints will be.

be "compare." Or, we may have a class that defines several different comparison routines corresponding to different criteria for ordering instances. Although the name "compare" may be overloaded within the class, the various overloaded routines must have different type signatures, so only one routine could be used in the instantiation of `keyType` in our example. An alternative, more flexible approach is to make the key comparison routine a *function parameter* to our `binaryTreeNode` class.

A function parameter specifies the argument and return types required of any actual function parameter; these types may be other formal class parameters. Function parameters are especially useful when we would otherwise be required to pass a function argument with each method invocation. Using a function parameter, we can specify the comparison routine as a separate parameter to the class:

```
class binaryTreeNode
[
        class   keyType( ),
        class   entityType( ),
        int     compare( keyType*, keyType* )
]
{ ... };
```

With this class definition, we may now use any type at all to instantiate `keyType`. If the instantiating type is a fundamental type, e.g. `float`, we must still write a comparison routine, of course.

If the instantiating type is a class having its own comparison routine(s), the desired routine may be supplied as the function parameter. In order to match the type signature, note that the implicit first argument to any method of a class `C` is the pointer `this` whose type is `C*`. Assume that we have a class `dataPoint` for recording data associated with some experiment and that we wish to build an index over such points. The key is to be a complex number taken from the experimental data, where `complex` is defined as follows:

```
class complex
{
        /* representation... */
public:
        int cmpImag( complex* ); // compare imaginary parts
        int cmpReal( complex* ); // compare real parts
};
```

We may then instantiate a node type in which the keys are ordered by their imaginary parts as follows:

```
class complexNode
        : binaryTreeNode[ complex, dataPoint, complex::cmpImag ];
```

#### 3.3.1.4. Constant Parameters

The last kind of class parameter that E supports is a constant. A constant parameter may be of any fundamental type, and within the generator class, it may be used freely as a **const**. This kind of parameter is particularly useful in defining array data members whose size depends on the particular instantiation. For example,

we may define a generic stack class where the maximum number of elements is a class parameter:

```
class stack [ class T { }, int STKMAX ]
{
      int   top;
      T     stk[ STKMAX ];
      ...
};
```

We may then define a stack of one hundred integers as follows:

```
class intStack : stack[ int, 100 ];
```

### 3.3.2. Class Name Scoping

In most respects, E is upward compatible with C++. The one exception is in class name scoping. In C++, it is legal to define nested classes, but this is "at most a notational convenience since a nested class is *not* hidden in the scope of its lexically enclosing class" [Stro86, p.152]. In order to support class parameters, which have no meaning outside the scope of a generator, we have changed this rule in general. In E, any nested class *is* hidden within the scope of its lexically enclosing class. Specifically, if class `B` is nested within class `A`, then `B` is visible only to `A` and `A`'s member functions, i.e. `A` may declare private members of type `B`, and `A`'s member functions may declare local variables of type `B`. `A` may not declare public data members of type `B`, nor may `A`'s public member functions have argument or return types involving `B`. We felt justified in making this exception to upward compatibility since generators *require* different scoping rules and since class nesting in C++ has no apparent benefit.

There is a subtle point involving scoping and generator classes. Consider the following definition:

```
class gen[ class T{ } ]
{
public:
      T     genFunc();
};
```

While the formal class `T` is essentially nested within `gen` (i.e. `T` is hidden within the scope of `gen`), `T` is also the return type of the public member function `genFunc`. This definition is perfectly valid, however, because `gen` must be instantiated before it may be used. The *instantiated* class has a public member function whose return type is well defined. For example, in

```
class intGen : gen[ int ];
```

the new class `intGen` has a member function, `genFunc`, whose return type is `int`.

### 3.3.3. Nested Instantiations

In defining a class `C`, it is normal to use other classes as part of `C`'s representation. Similarly, in defining a generator class `GC`, we often would like to make use of previously defined generators in `GC`'s representation. We

have said that a generator must first be instantiated before it may be used. In the case of generators using other generators, however, those specific types are rarely known. Our binary tree example is a case in point. We have shown how to define `binaryTreeNode` as a generator class, but what about the wrapper class, `binaryTree`? Certainly we would like to define a generic binary tree type, but how do we define the type of tree node that it encapsulates?

E's modified scoping rule for nested classes allows the definition of new types within the context of a class including definition through instantiation. Furthermore, within the context of a generator class `GA`, we may instantiate another generator `GB` by supplying any or all of `GA`'s parameters to `GB`. Then any instantiation of `GA` with actual parameters causes a nested instantiation of `GB`. We can make the `binaryTree` class a generator as shown in Figure 3.8. Within the context of `binaryTree`, a new class `btn` is instantiated from `binaryTreeNode` by passing along the parameters supplied to `binaryTree`. We will complete this class definition in the next section.

### 3.4. DB TYPES AND PERSISTENCE

In the discussion so far, we have described language extensions in E that allow the programmer to process sequences of values and to define parameterized types. Both features are important for database programming. However, the data objects available to the program thus far are still volatile objects whose lifetimes are bounded by a program run. We now introduce the features of E that allow a program to create and use persistent objects and thus to describe a database and its operations strictly within the language.

```
class binaryTree
[
        class   keyType{ },
        class   entityType{ },
        int     compare( keyType*, keyType* )
] {
        class btn : binaryTreeNode[ keyType, entityType, compare ];
        btn  *root;
public:
        binaryTree();
        entityType * search( keyType );
        void insert( keyType, entityType* );
};
```

A Generic Binary Tree Class

Figure 3.8

### 3.4.1. Database Types

E mirrors the existing C++ types and type constructors with corresponding database types (db types) and type constructors. Any type definable in C++ can be analogously defined as a db type. Db types are used to describe the types of objects in a database, i.e. the database schema. However, not every db type object is necessarily part of a database; db type objects may also be allocated on the stack or in the heap. (Another way of saying this is that persistence is orthogonal to db types.) We will shortly convert the binary tree class into a db type.

Let us informally define a db type to be any of the following:

(1) One of the fundamental db types: dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, or dbvoid. Fundamental db types are fully interchangeable with their non-db counterparts For example, it is legal to multiply an int and a dbshort or to assign a dbint to a float.

(2) A dbclass (or dbstruct, or dbunion). Every data member of a dbclass must be of a db type. The argument and return types of member functions may be either db or nondb types.

(3) A pointer to a db type object. The usual kinds of pointer arithmetic are legal on db pointers, and casting is allowed between one db pointer type and another. It is *not* possible to convert a db pointer into a normal (non-db) pointer, nor into any non-pointer type (e.g. `int`). It is legal to convert normal pointers into db pointers, however.

(4) An array of db type objects. As in C or C++, an array name is equivalent to a pointer to its first element.

### 3.4.2. Persistent Objects

An fundamental property of a language with persistence is that objects in the database may be manipulated using the same expression syntax as for volatile objects. In order to evaluate such an expression, however, there must first exist a binding between symbols in the program and objects in the persistent store; such a binding is informally called a "handle" on the database. Part of what distinguishes one persistent language from another is the nature of these handles: When are they established, and to what can they attach?

In E, if the declaration of a db type variable specifies that its storage class is **persistent**, then that variable survives across all runs of the program (and across crashes). A simple example is a program that counts the number of times that it has been run:

```
persistent dbint count = 0;
main() { printf("This program has been run %d times.", count++ ); }
```

Here, the integer `count` is a persistent variable whose initial value is set to 0. Each time the program runs, it

prints the current value of `count` and then increments it.[24] Note that there are no explicit calls to read or write `count`, and there are no references to any external files; I/O is implicit in the program. The great convenience of language support for persistence is that it allows the programmer to concentrate on the algorithm at hand rather than on the details of moving data between disk and main memory [AtkM83].

### 3.4.3. Collections

While the above example illustrates the essential concepts of persistence, it is hardly convincing; a single integer does not a database make! In fact, while the **persistent** storage class is the root of all persistence in E, by itself it is insufficient for the needs of database programming. First, it implies that every object in the database must be named, and second, it implies that creating a new object requires calling the compiler. What is needed in addition is a facility for managing unbounded collections of dynamically allocated, persistent objects.

Different researchers have taken different approaches to this problem. As we saw in Chapter 2, Pascal/R introduced **relation** as a type constructor; tuples could be added or deleted under program control, although the relations themselves could only be named variables. One implication of this restriction is that nested relations were not allowed. The other DBPLs, e.g. Rigel and Plain, took a similar approach with similar restrictions. PS-Algol made the run-time heap the basis for persistence. Any type of object could be made persistent by simply making it reachable from the database root pointer. However, the persistent heap has no notion of a collection of objects; such a collection would have to be coded explicitly as a persistent data structure. E takes an intermediate approach. Like the DBPLs, a given collection stores a specific type of object, and there are facilities for processing all of the objects in a collection. Like PS-Algol, there are no restrictions on the type of object that may be persistent (except that it must be a db type); for example, one may define collections of collections. E does *not* provide an implicit persistent heap; the dynamic creation of a persistent object requires the specification of a collection in which to create the new object.

E introduces collections via the built-in generator class `collection[T]` where `T` may be any db type. A `collection[T]` is an *unordered* collection of objects of type `S`, where `S` is `T` or any public subtype of `T`.[25] Like any generic class, the programmer must first instantiate a specific type of collection before declaring a collection object. As with any db type, a given collection object may be volatile or persistent, depending on the declaration, and it may be declared as a data member of another class. The lifetime of an object within a collection is bounded by the lifetime of the collection; in particular, if a program creates an object in a persistent collection,

---

[24]The initialization to zero occurs only once, i.e. when the object is created.

[25]That is, any type `S` for which `T` is a public base type, directly or indirectly. The reason for this restriction is that during a scan over a collection of `T`, the client obtains a pointer of type `T*` to each object in the collection. If an object in the collection were of type `S` where `S` inherited `T` privately, then giving the client a `T*` to that object would violate C++'s type rules. For an explanation of public versus private base classes, see [Stro86].

then that object will also be persistent.

#### 3.4.3.1. Creating Objects in a Collection

An extension to the syntax of the **new** operator accommodates the creation of objects in a collection. As an example, suppose that `person` is defined as a **dbclass** and that it has a constructor taking a character string, e.g. the person's name. Then the following E code defines a type describing collections of persons, declares an instance of that type, and creates two people within the collection:

```
dbclass person { ... };
dbclass City : collection[ person ];
persistent City Madison;
person * p1;
person * p2;
p1 = in( Madison ) new person("Jane Doe");
p2 = in( Madison ) near( p1 ) new person("John Doe");
```

Here, the syntactic extensions are the **in** and **near** clauses. In general, the **in** clause may be followed by any expression which evaluates to a collection as long as the type following **new** is the same as or a subtype of the type of entity in the collection. In this example, we are creating instances of `person` in a collection of persons, but if, for example, `student` were a subtype of `person`, we could also create student instances within this collection.

The **near** clause on the last line specifies a clustering hint to the underlying storage layer; in this case, the hint requests that the new `person` object be created physically near the object referenced by `p1`. In general, **near** may be followed by any pointer-valued expression, and the referenced object need not be of the same type nor in the same collection as the newly created object. It is up to the implementation of the underlying storage layer to determine what "near" means, and at worst, the hint will be ignored. In the implementation of the EXODUS Storage Manager, the search for a nearby location begins on the same disk page if the objects are part of the same collection, and on the same disk cylinder otherwise.

#### 3.4.3.2. Scanning Collections

The collection generator class has an iterator member function for scanning all of the elements in a collection. This iterator, `scan()`, returns a sequence of pointers to the objects in the collection. The following example processes all of the people in Madison:

```
iterate( person * p = Madison.scan() ){ ... }
```

Note that even though a collection of `T` may contain objects of a subtype of `T`, scans over such collections always return pointers to type `T*`. For example, the preceding scan always yields a `person*`, although the referenced object might be of a subtype of `person`. However, if a `T` member function, `f`, is invoked through the returned pointer, the binding of the call will be late or early as `f` is virtual or not, respectively. If `f` is virtual, then the version of `f` associated with the actual type of the object will be called; if `f` is not virtual, then `T`'s version of

`f` will always be called.[26] Although early binding may be acceptable in some cases, in general, if a `collection[T]` is to contain instances of a subtype of `T`, then `T`'s member functions (and those of `T`'s subtypes) should be declared virtual. Of course, if the collection is to contain only `T` objects, then this discussion does not apply.

#### 3.4.3.3. Destroying Objects and Collections

The usual **delete** operator may be used to remove an object from a collection. For example, we can delete "John Doe" (from our earlier example) with:

```
delete p2;.
```

If the object's type has a destructor, then the destructor will be called first, and then the object will be destroyed.

If a collection is destroyed, the objects that it contains are destroyed also. If the collection contains objects of type `T` where `T` has a destructor, then the destructor will be invoked on each object before the collection is destroyed. Assume we wish to delete `Madison`, which is a `collection[person]`. Conceptually, this process involves the following steps:

```
iterate( person * p = Madison.scan() ) { delete p; }
/* now destroy the empty collection...*/
```

For performance reasons, however, our implementation does not actually destroy the objects individually. Rather, the entire collection is then destroyed ***en masse***.

The previous remarks concerning collections and virtual functions also apply to destructors. If a `collection[T]` is destroyed, then the actual destructor function invoked on each object depends on whether `T` has declared its destructor to be virtual or not.

### 3.4.4. The Binary Tree Example Revisited

Let us now (finally) reimplement our binary tree example as a db type. Figures 3.9, 3.10, and 3.11 replace Figures 3.1, 3.2, and 3.3, respectively.

The node class shown in Figure 3.9 has changed from the C++ version of Section 3.1 in the following ways: The insert routine accepts duplicates, and the search routine is an iterator (as developed in Section 3.2); the key and entity types are type parameters, and the key comparison routine is a function parameter (as developed in Section 3.3); and the class itself is a **dbclass** (as developed in Section 3.4).

---

[26]This description simply restates the semantics of C++ virtual functions, as described in Section 3.1.3.

```
dbclass binaryTreeNode
[
        dbclass keyType{ },
        dbclass entityType{ },
        int     compare( keyType*, keyType* )
] {
        keyType                 nodeKey;
        entityType              *entPtr;
        binaryTreeNode          *leftChild;
        binaryTreeNode          *rightChild;
public:
        binaryTreeNode( keyType, entityType * );
        iterator entityType * search( keyType );
        void insert( binaryTreeNode * );
};

binaryTreeNode::binaryTreeNode
( keyType insertKey, entityType * insertPtr )
{
        nodeKey = insertKey;
        entPtr = insertPtr;
        leftChild = rightChild = NULL;
}

iterator entityType * binaryTreeNode::search( keyType searchKey ) {
        int cmp = compare( &searchKey, &nodeKey );
        if( cmp <= 0 ) {
                if( leftChild != NULL )
                        iterate( entityType * p = leftChild->search( searchKey ))
                                yield p;
                if( cmp == 0 )
                        yield entPtr;
        }
        else if( rightChild != NULL )
                iterate( entityType * p = rightChild->search( searchKey ))
                        yield p;
}

void binaryTreeNode::insert( binaryTreeNode * newNode ) {
        int cmp = compare( &(newNode->nodeKey), &nodeKey );
        if( cmp <= 0 )
                if( leftChild == NULL )
                        leftChild = newNode;
                else
                        leftChild->insert( newNode );
        else if( rightChild == NULL )
                rightChild = newNode;
        else
                rightChild->insert( newNode );
}
```

The Binary Tree Node Class

Figure 3.9

```
dbclass binaryTree
[
       dbclass keyType{ },
       dbclass entityType{ },
       int    compare( keyType*, keyType* )
] {
       dbclass btn : binaryTreeNode[ keyType, entityType, compare ];
       dbclass btnSet : collection[ btn ];

       btnSet      allNodes;
       btn         *root;

public:

       binaryTree();
       iterator entityType * search( keyType );
       void insert( keyType, entityType * );
};

binaryTree::binaryTree()
{
       root = NULL;
}

iterator entityType * binaryTree::search( keyType searchKey )
{
       if( root == NULL )
             return;
       else
             iterate( entityType * p =  root->search( searchKey ))
                   yield p;
}

void binaryTree::insert( keyType insertKey, entityType * insertPtr )
{
       btn * newNode;
       newNode = in( allNodes ) new btn( insertKey, insertPtr );

       if( root == NULL )
             root = newNode;
       else
             root->insert( newNode );
}
```

The Binary Tree Class

Figure 3.10

Figure 3.10 shows the binary tree class. In order to define this class, we must first instantiate two new classes which we then use. The class `btn` is `binaryTreeNode` instantiated with the same parameters as `binaryTree`, i.e. this is a nested instantiation as described in Section 3.3.3. Next, `btnSet` is instantiated as a type of `collection` containing `btn` nodes. The binary tree itself is now represented as `allNodes`, a collection containing the nodes, and `root`, a pointer to the root node. On an insert, the new node is allocated in

the tree's collection. Other changes to the binary tree class parallel those made for the node class: the use of type parameters and the definition of `search` as an iterator.

Figure 3.11 shows an example using a persistent binary tree index. Like the original main program in Figure 3.3, this one builds an index over students keyed on gpa. Since the students must persist, we first define `school` as a collection of students, and we declare a persistent instance, `UWmadison`, of this type. We then define a comparison routine for floating point numbers, and we use this routine, along with the types student and dbfloat, to

```
dbclass student{ ... };
dbclass school : collection[ student ];
persistent school    UWmadison;

int compare( dbfloat * x, dbfloat * y ) {
      float cmp = (*x - *y);
      if( cmp < 0 )
            return -1;
      else if( cmp == 0 )
            return 0;
      else
            return 1;
}

dbclass gpaIndexType : binaryTree[ dbfloat, student, compare ];
persistent gpaIndexType gpaIndex;

main() {
      student *    s;
      float        gpa;
      int          cmd;

      while( (cmd = getCommand()) != QUIT )
            switch( cmd )
            {
            case NEWSTUDENT:
                  getNewStudent( &gpa, &s );
                  gpaIndex.insert( gpa, s );
                  break;

            case PROCESS:
                  getGpa( &gpa );
                  iterate( student * s = gpaIndex.search( gpa ))
                        processStudent( s );
                  break;
            }
}
```

Example Using a Persistent Binary Tree

Figure 3.11

instantiate a specific index type. Next we declare a persistent index, `gpaIndex`. Finally, the main program parallels the operation of that in Figure 3.3, except that the `getNewStudent` function is assumed to create the student in the `UWmadison` collection, rather than on the heap. Any students entered during one run of the program will therefore still be present in later runs.

### 3.4.5. Implementing a Database Index

The binary tree example that we have developed in this chapter is obviously hinting at the implementation of "real" database index structures, e.g. B+trees, in which each node contains many keys. In defining such structures, an essential constraint is that each index node must fit on one disk page and must make maximal use of the space on that page. If we define the node type as a generic class, then clearly, the number of keys that will fit on a page varies with the specific key type. One approach is to define the generator with a constant parameter, as we did in the stack example of Section 3.3.1.4. However, this approach forces the user of the class to compute the maximal number of keys for each instantiation.

An easier approach is to make use of the fact that within a generator, the expression `sizeof(T)`, where `T` is a type parameter, is treated as a constant and may be used in calculating array bounds. For example, assume that `PAGESIZE` is a constant giving the size of a disk page in bytes. (This could be a class parameter, but it is more

```
dbclass BTreeLeaf
[
      dbclass keyType{ },
      dbclass entityType{ },
      int   compare( keyType*, keyType* )
] {
      /* auxiliary definitions */
      dbstruct      kpp {
            keyType             keyVal;
            entityType *        entPtr;
      };

#define      MAXSPACE     (PAGESIZE - sizeof(dbint))
#define      MAXENTRIES   (MAXSPACE / sizeof(kpp))

      /* data members */
      dbint nKeys;
      kpp   kpPairs[ MAXENTRIES ];

public:
      ...
};
```

A Generic DbClass for B+Tree Leaf Nodes

Figure 3.12

likely to be a system-wide constant.) In Figure 3.12, we have outlined the definition of a (simplified) generic class describing leaf nodes in a B+tree; each node is to contain an array of key-pointer pairs where the number of array elements is the maximum that will fit on one page. Like the binary tree example, this class is parameterized by the key and entity types and by the key comparison routine. For convenience, we have defined an auxiliary type, `kpp`, for key-pointer pairs; the tree node is an array of these structures. Note that since `kpp` is defined in terms of class parameters, it is also a generic type, and it is implicitly instantiated with each instantiation of `BTreeLeaf`. We then define two macros for convenience. The amount of usable space on a page is the size of the page minus any overhead for control information; in this simple example, the only control data is an integer giving the current number of entries in the array. Finally, the maximum number of array entries is the amount of available space divided by the size of an entry, i.e. by `sizeof( kpp )`. The data member, `kpPairs`, is then defined to be an array whose dimension is this maximum.

## 3.5. TWO LANGUAGE DESIGN ISSUES

This chapter has presented the major features of E within the framework of developing a specific example. Let us now explain the rationale behind two major language design issues. In Chapter 8, we shall consider these and other design decisions in retrospect and give some suggestions for possible improvements.

### 3.5.1. Orthogonality

One legitimate question is why we chose to give E a "two-headed" type system, rather than simply to introduce persistence as an orthogonal property of all types. Orthogonality is, after all, often cited as a desirable feature of a persistent language [AtkM83, AtkM87]. The reasons for E's use of db types stem both from philosophy as well as from implementation concerns. First, E was originally conceived as a language in which to write database management systems. In such systems, there is a clear distinction between those objects that persist and those that are volatile. For example, lock tables and transaction descriptors are definitely not persistent[27], while objects in the database definitely are. The "db" attribute of a type distinguishes between objects that *may be* persistent and those that are definitely volatile.

The importance of separating normal types from db types also has a strong grounding in performance considerations. Those same system resources that are known to be volatile are often the ones that are accessed with the highest frequency. If every object that is referenced can potentially be a persistent object, then before every access the system must check that the needed object is in memory. Even if the check costs only one boolean test,

[27] We ignore design transactions which need to hold long term locks. Again, however, the system designer would be aware of the distinction between long term (persistent) locks and short term locks.

the cost of accessing these critical system resources might be significantly increased. In E, accesses to non-db type objects suffer *no* loss of performance over the same accesses in C++.

In addition to the cost of a pointer dereference, another factor in our decision to introduce db types is the representation of pointers. On a VAX, a pointer is 32 bits, giving an address space of approximately 4 GB. However, databases are already exceeding this limit and, in fact, are moving into the terabyte range. If persistence were orthogonal to all types, then we would be faced with several not very appealing alternatives. We could make all pointers 32 bits, and thus guarantee that E would already be obsolete for real world applications. Alternatively, we could make all pointers be "large enough" for projected needs. E programs use the EXODUS Storage Manager [Care86a] as its persistent store, and every object id is 12 bytes; since a pointer to a persistent object also contains an offset, the total size is 16 bytes. Thus, this approach would quadruple not only the space needed but also the copying cost for every pointer.

As a final comment on the question of orthogonality, note that (1) any nondb type can be (easily) defined as a db type, and (2) persistence *is* orthogonal over all db types. Thus, it is easy to give E the appearance of a language having completely orthogonal persistence if desired. By including the file shown in Figure 3.13, the programmer may omit all mention of "db", and may declare any variable persistent. Referring to the above discussions, the resulting "language" is then one which takes the approach of making all pointers large and all pointer dereferences a bit more costly.

### 3.5.2. Persistent Handles

In a language without persistence, long term store has traditionally been implemented by files. The persistent name space is simply the space of file names maintained by the operating system. In this case, the program's handle on the database is a character string representing a file name. A run-time call to a system routine (e.g. open)

```
#define    class     dbclass
#define    struct    dbstruct
#define    union     dbunion
#define    int       dbint
#define    short     dbshort
#define    long      dblong
#define    float     dbfloat
#define    double    dbdouble
#define    char      dbchar
#define    void      dbvoid
```

A Useful Include File

Figure 3.13

establishes the binding (e.g. a file descriptor) between the program and an actual file. The program can then access the persistent data via read and write calls. One obvious drawback of this approach is type safety; since a file may be accessed independently, there is no guarantee that the file bound to one run of the program has any relationship to the one bound in another, except that they have the same name.

In PS-Algol, a program's persistent handle is a character string naming an operating system file, and the actual binding is established with a run-time call. This time, however, the file contains a persistent PS-Algol heap, and the open call binds the heap to a pointer in the program called a database *root*. The program is then free to dereference through the root pointer to access the rest of the database. The top level object in every database (i.e. what the root points to) is an associative index of pointers keyed on string names. By providing a character string argument to a lookup routine, the user gets back a pointer to the object associated with the string. The persistent name space thus comprises whatever strings the user has stored in the index. It should be noted that these strings are not recognized as variable names by the compiler, so there is no persistent binding between program symbols and database objects. A program can cause an object to persist at run-time by establishing an access path to the object from the root of an open database. When the program closes the database, all objects reachable from the root are written out to disk.[28]

In E, if a variable is declared to have the `persistent` storage class, that variable's name is a persistent handle. Having db types allows the E programmer to define the types of objects in a database; the **persistent** storage class provides the basis for populating the database. The existing scoping rules of C++ determine the organization of the name space. That is, a name *n* in one source module does not conflict with *n* declared in another module unless one attempts to link the two into a single program. The binding between the name and the persistent object is established at compile time[29], and it remains valid until the module is explicitly destroyed. There is no explicit run time open call, and there are no references to external file names. Bindings between the source code and persistent objects are maintained implicitly by the programming environment. In order to access the database, one compiles and runs an E program which manipulates persistent objects. These objects may be declared in the same module with the program, or they may be declared as `extern` variables. In the latter case, one must link the program together with other object modules (.o files) that contain the desired persistent variables. The language does not define how such modules are named or accessed, however.

---

[28]This is an oversimplification. An object is written back only if was created or changed during the program run.

[29]This describes the implementation in versions 1.0 and 2.2 of the compiler. Version 2.1 defers the actual creation of the object to the first run of a program that uses the persistent handle.

# CHAPTER 4

# COMPILER ORGANIZATION

In the next three chapters, we describe the implementation of db types and persistence in version 2.2 of the E compiler. This chapter presents an overview of the entire compilation process, describing the pieces that relate to persistence and showing how these pieces fit together. We begin with a description of the compiler's internal structure and show how new functionality has been integrated with the phases of an existing C++ compiler. Next we discuss the processing of declarations of persistent objects and their types. We then present an overview of code generation, previewing briefly the material that Chapters 5 and 6 will cover in detail. This chapter concludes with discussions of two implementation problems that arise due to the combination of E's model of persistence with C++ semantics; we describe both the problems and the solutions adopted in version 2.2 of the compiler.

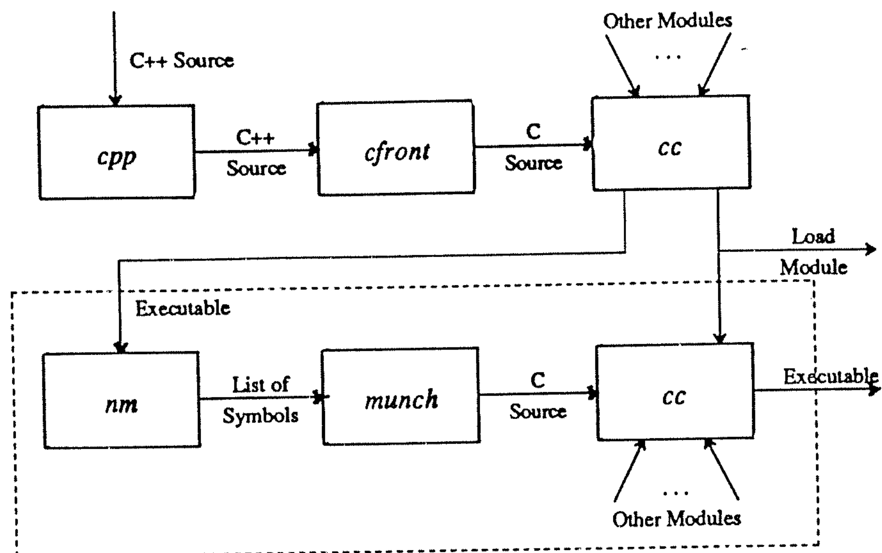
## 4.1. ARCHITECTURE OF THE COMPILER

The E compiler is an extension of version 1.2.1 of the AT&T C++ compiler. We chose this compiler as our starting point for several reasons, the main one being that we had access to the source code. We did not want to start from scratch because reimplementing the C++ subset of E was not our interest and would require a significant amount of time and effort. At the time we began work on E, there were only a few C++ compilers available; the AT&T compiler, in addition to being available, also had the distinct advantage of being quite stable.

The C++ compiler consists of a large shell script, *CC*, which spawns three processes, as illustrated in Figure 4.1. First, the standard C preprocessor, *cpp*, performs macro expansion and file inclusion. The C++ front end, *cfront*, then translates the result into C source code. The C compiler, *cc*, compiles the output of cfront into binary form. Although not shown in the figure, cc itself comprises a series of processes: *cpp* (again!); *ccom*, which translates C into assembler code; *as*, the assembler; and *ld*, the link editor.

If the result of the last step is an executable program, the compiler then performs an additional series of steps (enclosed in the dashed box in Figure 4.1). A C++ program may declare an object of a class with a constructor. If such an object is declared in the global scope, then the constructor for that object must execute before the main program runs. The extra steps in the dashed box are part of the mechanism that implements this feature.

The majority of the work in implementing E has involved extending the source code for cfront into an E-to-C translator called *efront*. The internal organization of efront is shown in Figure 4.2. Of the five phases shown, all but *db simplify* were part of the original C++ compiler. A yacc-produced parser [John75] builds an abstract syntax tree of the source text. The parser consumes one external declaration at a time, e.g. one function definition or one global



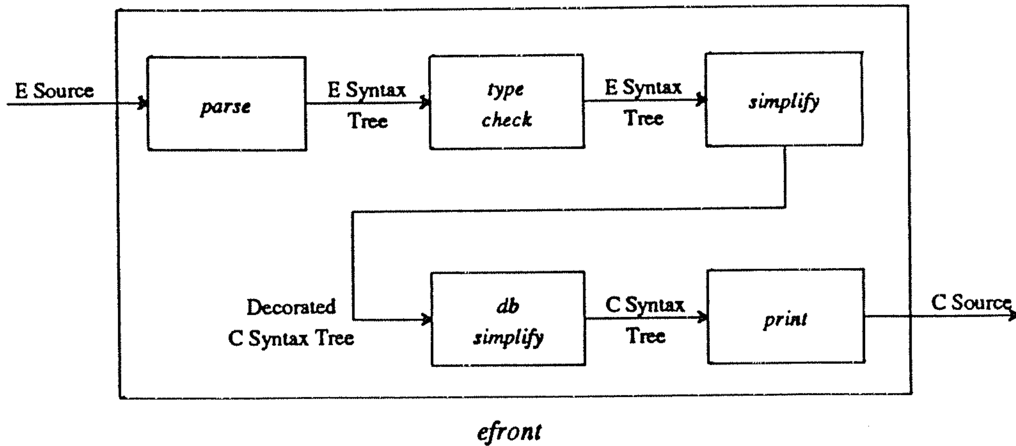

The Process Structure of CC

Figure 4.1

variable declaration. A few new keywords and productions were added to the grammar in order to handle constructs related to persistence. In addition, when the parser adds a type node to the syntax tree, a special flag is set in that node if it represents a fundamental db type (e.g. dbint) or a dbclass. In all other respects, it is like any other type node. This fact is important for the second phase, which handles type checking. Slight changes were needed here to prevent, for example, assigning a db address to a non-db type pointer. Most other type checking is handled normally, to allow such common activities as assigning a dbint to an int, or adding an int and a dbfloat. The simplification phase transforms C++ constructs into equivalent C constructs. This phase has been extended also to handle E generators and iterators.[30] The new fourth phase, and the one which is the concern of these three chapters, transforms constructs related to db types and persistence. Finally, the print phase walks the resulting C syntax tree, producing C source code.

Given the architecture of efront, the input to the db-simplification phase is a C syntax tree in which certain nodes are decorated. Any expression or object of a db type will point to a type node marked "db." Any object declared persistent will have the **persistent** storage class recorded as part of its symbol table entry. It is the responsibility of the db-simplification phase, therefore, to look for such decorated nodes and to apply appropriate

[30]This work was done by another member of the EXODUS project, Dan Schuh. See acknowledgements.




Compilation Phases in Efront

Figure 4.2

transformations, producing as output another C syntax tree.

### 4.2. PROCESSING DECLARATIONS

This section details the transformations that are applied to the declarations in a program, e.g. to type or data declarations. The interesting cases include types that define pointers to db objects (db type pointers) and declarations of persistent data.

#### 4.2.1. Representation of Objects and Pointers

One important consideration in the implementation of a persistent language concerns the implementation of objects on disk. How are objects organized internally? Is there a format-conversion as objects are brought into memory? How are their addresses represented? In order to place E's approach to these issues in some perspective, we compare it to the implementation of PS-Algol [AtkM83, AtkM84], a particularly well known predecessor.

The representation of persistent objects in a persistent language is dictated, in part, by base language from which it is derived. S-Algol [Morr82], the starting point for PS-Algol, is a heap-based language in which "the data type pointer comprises a structure with any number of fields, and any data type in each field" [AtkM83]. Any pointer may point to any structure, that is, pointers are inherently untyped. This design has two important implications for PS-Algol. One is that performance is affected since all pointer dereferencing is subject to run-time

type checking. For example, in the expression[31] `p(x)`, the compiler cannot know in general that at run-time `p` will point to a structure with a field named `x` whose type matches that required in the current context. Therefore, a run-time check is included to validate the expression. The other important implication is that run-time type descriptors must be available in order to carry out this check. In a persistent extension, of course, these descriptors must also be persistent.

Another reason that PS-Algol needs type descriptors at run-time is that the basis for deciding on the persistence of an object is reachability from a persistent root. In fact, garbage collection is an integral part of any PS-Algol implementation. Since an object may be an arbitrary composition of structures, arrays, and pointers, and since pointers in PS-Algol are inherently untyped, there must be type descriptors available in order to do the reachability traversal. Again, since the objects being traversed are persistent, so must be the descriptors. The efficient representation and processing of type descriptor information thus formed a significant part of the implementation effort of PS-Algol [Cock84, Brow85].

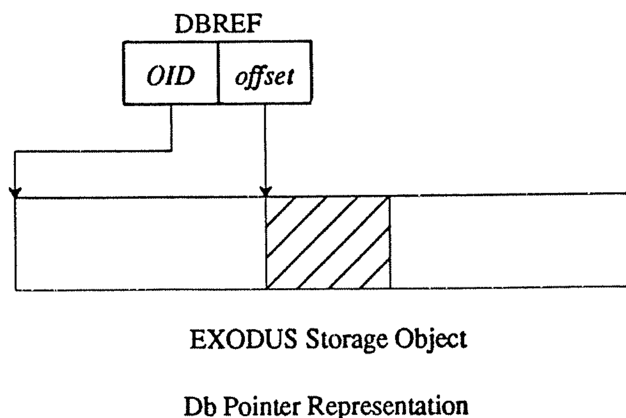
In contrast, E, being derived from C++, is a language in which the physical structure of objects is known (and specified) by the programmer. Dynamic storage allocation is under explicit programmer control, i.e. there is no garbage collection. Furthermore, pointer is a type *constructor*, rather than a fundamental type, so all pointer dereferences are type checked at compile time. Because there is no garbage collection, and because type checking is static, there is no general need to maintain persistent type descriptors.

Another way in which E differs from PS-Algol is in the implementation of pointers. The PS-Algol run-time system recognizes two kinds of pointers, both one word in length, that are distinguished by their most significant bit (msb). A pointer with an msb of zero is called a Local Object Number (LON) and it contains the virtual memory address of the object it references. If the msb is one, the pointer is a Persistent IDentifier (PID) and contains the database address of the object. At run-time, PIDs are converted to LONs and back, as objects are moved in and out of memory.

Like PS-Algol, E recognizes two kinds of pointers. Unlike PS-Algol, they are distinguishable by type at compile time. Any pointer whose type is defined in the C++ subset of E is a normal C++ pointer, i.e. it is one word in length and contains a virtual memory address. Any pointer whose type is a db type has a different format. The Lvalue of a db type object comprises an EXODUS storage object id and an offset into the object, as shown in Figure 4.3. This [OID,offset] pair is called a DBREF. The offset is necessary because it is possible (and quite common) for a program to produce an address which lies in the middle of a storage object. For example, one often processes an array by incrementing a pointer to each element in turn. Also, member functions of a dbclass, like their non-

---

[31] PS-Algol uses parentheses to express structure access.




EXODUS Storage Object

Db Pointer Representation

Figure 4.3

dbclass counterparts, are passed a pointer, `this`, to the beginning of the class instance; this instance, however, may be embedded as a data member in some containing class object. This aspect of E pointers — really, a straightforward extension of current C++ semantics — is in contrast to PS-Algol, in which pointers are constrained to reference only "top-level" objects, i.e. to the start of an independently allocated unit of storage. Finally, a pointer to a nonpersistent db type object comprises an OID indicating that the object resides in virtual memory and an offset field that contains the object's virtual address.

#### 4.2.2. Type Declarations

E provides a full complement of fundamental types having the "db" attribute: dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, and dbvoid. These types are duals of the fundamental C++ types, and they have the same representations.[32] For the purposes of assignment, expression evaluation, and parameter passing, these db types are equivalent to their non-db counterparts. A node in the abstract syntax tree that represents a fundamental db type is not changed by db-simplification; it is simply printed as its non-db dual. For example, if the type of an argument to an E function is `dbint`, then in the C translation, the type of that argument is `int`.

A node representing a function type is processed by recursively transforming the function's return and argument types. If the function also has a body, i.e. if it is a definition, then we also generate code, as described later.

[32] These representations are machine dependent, of course.

For now, we may simply observe that the following (pointless) function

```
dbvoid  fcn( dbfloat x,  dbint y )
{
     x = y;
}
```

translates to:

```
char  fcn( x, y )
float x;
int   y;
{
     x = y;
}
```

A node representing a dbclass (or dbstruct or dbunion) is processed by recursively transforming the dbclass's data and function members. Although we do not show an example, it should be noted that by the time control enters this phase of the compiler, classes related to generators (i.e. generators and classes instantiated from them) have already been transformed into an equivalent set of non-generator classes.

So far, the transformation of types is not very interesting. The one important translation occurs when a type node represents a pointer to an object of some db type. Recall from the discussion in the last section that the address of a db type object is represented by an [OID,offset] pair. In the C translation of every E program is a series of typedefs culminating in the following declaration:

```
struct DBREF {
     OID          oid;
     int          offset;
};
```

Any type node representing a db pointer is printed as a **struct** DBREF. Consider, for example, the declaration of a dbclass implementing a binary tree node (based on the examples in Chapter 3):

```
dbclass binaryTreeNode {
     dbfloat           nodeKey;
     dbvoid            *entPtr;
     binaryTreeNode    *leftChild;
     binaryTreeNode    *rightChild;
};
```

The C translation is printed as:

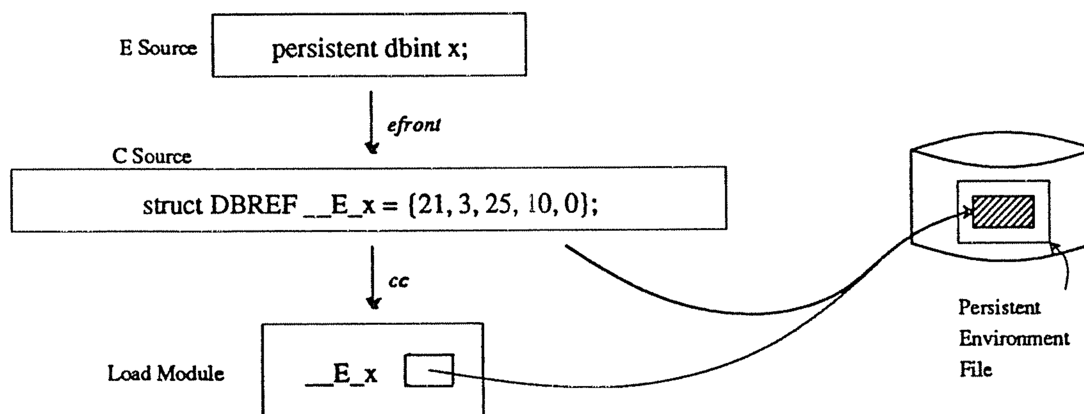
```
struct binaryTreeNode {
     float         nodeKey;
     struct DBREF  entPtr;
     struct DBREF  leftChild;
     struct DBREF  rightChild;
};
```

#### 4.2.3. Data Declarations

When the source code contains the declaration of a db type object, the correct transformation depends on the scope of the object and its storage class. Obviously, the most interesting case is the declaration of a persistent object. In this version of the E compiler (version 2.2), an object which is declared persistent establishes a binding with a physical object at compile time.[33] The name of the persistent variable is then the programmer's handle on the object; any program in which the variable's name is visible may then access the object.

Before creating any persistent objects, the compiler first asks the storage manager to create a file; all persistent objects declared in the source module are then created within this file. Thus, compiling an E source module that contains the declaration of one or more persistent objects yields both its C translation and a file in the storage manager containing the objects.

The general approach is illustrated in Figure 4.4. When the db-simplification phase sees the declaration of the persistent db integer, `x`, it asks the EXODUS storage manager to create a 4-byte object. The OID of the new object is then introduced into the output in the form of a DBREF structure with an initializing expression.[34] This DBREF variable is called the ***companion*** of x, and the initializer assigns it the OID returned by the storage manager together with an offset of 0. Note that only the companion's declaration appears in the C output.




Compiling A Persistent Object Declaration

Figure 4.4

[33]The implementation in version 2.1 of the E compiler allows for more flexible bindings, e.g. it allows recompilation of a source module to retain previous bindings.

[34]The interpretation of the numbers composing the OID is not important for this discussion.

If a db type object is declared to be external, db-simplification transforms this declaration into an external reference to the object's companion. For example,

```
extern dbint x;
```

becomes

```
extern struct DBREF __E_x;
```

A function in one module may thus access persistent objects declared in another via the usual C++ external reference mechanism. Conversely, it implies that if a module declares a *non*persistent db type variable in the global scope, then a companion must be generated for that object as well, since another module may include the variable's name in an **extern** declaration. The companion must be initialized with the address of this object, which in this case, is in main memory. Such addresses use a null OID indicating "in memory", and the actual address of the object is embedded in the offset. Thus, if the declaration

```
dbint x;
```

appears in the global scope, then the translation is:

```
int x;
struct DBREF __E_x = { 0, 0, 0, 0, (int) &x };
```

The example of Figure 4.4 showed a persistent object declared in the global scope. Persistent objects may also be declared locally in a block. For example,

```
int counter()
{
        persistent dbint x;
        return x++;
}
```

In this case, although the object is persistent, its name is visible only within the block. Again, the object is created at compile time, and a companion is introduced into the local scope. Here, the companion is given the storage class `static` so that it need not be reinitialized every time the block is entered. The declaration in the above function thus becomes:

```
int counter()
{
        static struct DBREF __E_x = { ... };
        /* return ... */
}
```

For nonpersistent db type objects declared in a local scope, there is no special processing. A local dbint `x` in the E source simply becomes a local int `x` in the C translation because expressions which use `x`, e.g. addition, *know* that `x` is not persistent. In the case where an expression takes the address of `x`, e.g. in passing `x` by reference, a temporary companion is constructed.

To be consistent with C++ semantics, a persistent object declared without an initializer will be initialized to all zero bytes. Thus, the above counter function will return 0 the first time it is called. A persistent object may also be declared with an initializer, as in Figure 4.5. In this example, the user has declared a persistent array of dbfloats, specifying the first 3 elements. In such cases, the compiler itself interprets the expressions (which must evaluate to constants) and sends the binary image of the object to the storage manager.

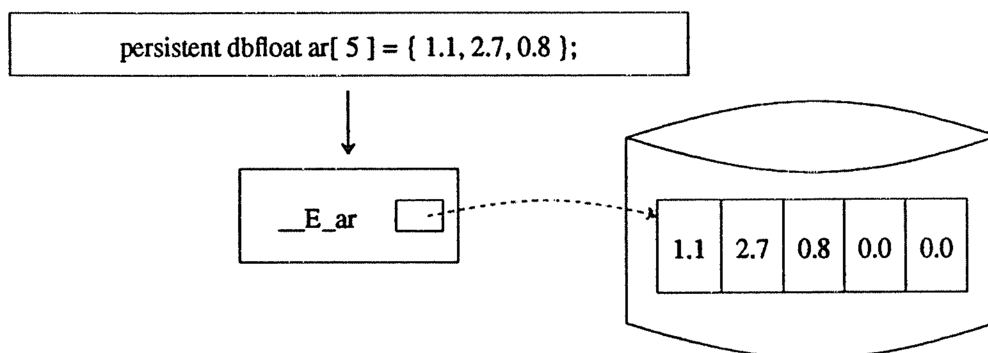
## 4.3. GENERATING CODE

Code generation presents one of the major challenges in the implementation of persistence for E. The code generator's job is to take as input an abstract syntax tree for an E procedure and to introduce code that manipulates persistent objects; since such manipulations involve I/O, it is critically important that the compiler generate efficient code. While Chapters 5 and 6 describe code generation in detail, here we discuss the rationale behind our approach to the problem, show how code generation ties in with the rest of E's implementation, and give a flavor of how the code generator operates.

### 4.3.1. Two Machine Models

#### 4.3.1.1. A Persistent Virtual Memory

Given that the persistent objects of a program reside on secondary storage, we must decide the mechanism by which these objects migrate in and out of main memory during a program run. We must also decide how external addresses are mapped to internal addresses that are usable by the program. The approach pioneered by PS-Algol is




An Initialized Persistent Object

Figure 4.5

to view the storage layer as a persistent virtual memory. A software[35] check of the address format precedes each pointer dereference. If the pointer is in persistent identifier (PID) format, an "object fault" is signaled. The object is read into memory, and the pointer that caused the fault is overwritten with the object's virtual address, i.e. with its LON. Once a pointer has been converted into LON format, it will no longer cause an object fault, although every reference is still subject to the run-time check. Maier refers to such pointer translations as "pointer swizzling" [Maie87b].

When an object X is written back to disk, any pointers in X that were converted to LONs must first be restored to PID format. Furthermore, any pointers (i.e. in other objects) which contain X's LON must now have X's PID restored. This description is obviously simplified, but the basic idea was used in several different implementations [AtkM83b, Cock84, Brow85]. It should be noted that this approach has much in common with the mechanisms employed in LOOM [Kaeh83], an implementation of object virtual memory that supports Smalltalk [Gold83]. The major difference is that LOOM does not attempt to support persistence or transactions [Kaeh86].

#### 4.3.1.2. A Load/Store Machine

E views the problem of code generation as being roughly analogous to register allocation in a load/store machine. In such a machine, a program must first move a data word from main memory into a register before it can manipulate that data. The number of registers is assumed to be smaller than the amount of data used by the program, and the transfer rate of data into the registers is limited by the speed of the slower main memory. The greater the difference in memory speeds, the more important it becomes to reduce the number of loads and stores performed by the program. In the case of persistent objects, the slow memory (disk) is four to five orders of magnitude slower than fast memory (main memory).

This analogy with load/store architectures may be extended further. In addition to fast and slow memory, there is often a memory of intermediate speed, a cache, between the two. The cache is, in most cases, invisible to the compiler. By exploiting the locality of references in time and space, the cache is able to intercept most memory references, thus giving the appearance that slow memory is responding at the speed of the cache. For persistent E objects, the cache is the buffer pool provided by the EXODUS Storage Manager. When the program issues a request for persistent data, the Storage Manager accesses the disk only if the data is not already resident in the buffer pool. The speed of this cache is essentially the path length through the Storage Manager on a buffer pool hit.[36]

---

[35]Hardware implementations are also possible and, in fact, some have been designed. See, for example, [Cock87].

[36]This figure is somewhere in the range of 100-200 instructions. Thus, our cache is about 2 orders of magnitude slower than our fast memory, and between 2 and 3 orders of magnitude faster than our slow memory.

Of course, there are significant differences between generating code against a load/store machine and generating code against the EXODUS Storage Manager. The greater difference in memory speeds makes certain dynamic techniques economical. Spending two instructions to save one makes no sense, but spending two to save several hundred is certainly a win. We will develop this idea in Chapter 6. Another difference from register allocation is that registers are usually fixed in number and in size, and each load operation usually moves only one word into a register.[37] The Storage Manager, by contrast, supports requests for arbitrary-length strings of bytes. This capability allows the compiler to "coalesce" separate requests for neighboring byte ranges into a single request for the combined range. Finally, because of the possibility of aliasing, register allocation must be careful not to load the same data word into two different registers; otherwise an update to one would not propagate to the other, possibly leading to inconsistent results. On the other hand, the Storage Manager is a non-copy-based system (unlike, for example, WiSS [Chou85b]). The result of a data request is a pointer to the data in the buffer pool, and two different requests for the same data item receive pointers to the same location. Thus, the impact of aliasing is less severe (although still present) in the E compiler than it is in compilers for typical hardware architectures.

### 4.3.2. The Storage Manager Interface

Because the EXODUS Storage Manager figures so heavily in the design of E's code generator, it is worthwhile to digress briefly here and review its interface. The basic abstraction provided by the Storage Manager is the *storage object*, which is an uninterpreted byte sequence of virtually any size. Whether the size of an object is several bytes or several gigabytes, clients of the Storage Manager see one uniform interface. Each object is named by its object ID (OID), which is its physical disk address. Two operations, `sm_ReadObject` and `sm_ReleaseObject`, are the basic means of moving data in and out of memory, i.e. they are the analogues of load and store, respectively. (Again, however, the Storage Manager "machine" has a cache, so a release operation does not actually return the data to disk.) The read call specifies an object ID (OID), an offset, and a length. The Storage Manager reads the specified byte sequence into the buffer pool and then returns the address of a *user descriptor* to the client. This descriptor contains a pointer to the data in the buffer pool, and the client may read the data by dereferencing through this pointer. Because the Storage Manager design supports atomic, recoverable transactions, the client updates data with a procedure call, `sm_WriteObject`, which updates the target object and performs the appropriate logging. When it is finished with the data, the client program returns the user descriptor to the Storage Manager in a release call, after which the descriptor is no longer valid.

---

[37]Statements about hardware architectures must always be qualified with terms such as "usually" because there are always exceptions. The NYU Ultracomputer [Edle85], for example, exposes the cache to software control, and the Berkeley RISC machine [Patt82] provides a variable number of registers in each stack frame.

It is important to understand that the data requested in a read call is *pinned* in the buffer pool until the client releases it. Pinning is a two-way contract: the Storage Manager guarantees that it will not move the data (e.g. page it out) while it is pinned, and the client promises not to access anything outside the pinned byte range. In addition, the client promises to release (unpin) the data in a "timely" fashion, because data that remains pinned unnecessarily can degrade performance by effectively reducing the size of the buffer pool.[38] In subsequent discussions, the terms pin and unpin are used interchangeably with read and release.

An example of a client's interaction with the Storage Manager is illustrated in Figure 4.6.[39] Assume that a range of bytes containing a `struct S` is embedded at location `offset` within the Storage Manager object having the given `oid`. The code segment shown multiplies the structure's `x` field by 10. The read call first pins the range of bytes containing the structure. On return, `ud` points to a user descriptor whose first word contains a pointer to the pinned data. The statement after the read then multiplies the x field by 10 and assigns the result to `temp`. The write call specifies that four bytes are to be written starting at offset 0 within the pinned byte range described by `ud`; the last argument to this call gives the address of the data to be copied into the object. Finally, the pinned data is released. While the EXODUS Storage Manager is a powerful utility, it is necessary for the programmer to learn numerous procedures and to execute many steps in order to perform even simple tasks. There are 25 interface routines to the Storage Manager, although only a few may be needed for a given application. Not shown in this example are the (necessary) steps of initializing the Storage Manager, mounting a volume, and allocating buffer space via the buffer group mechanism. The resulting complexity was one of the original motivations for the E language.

```
struct S      { int x; float y; };
USERDESC *    ud;
int           temp;

sm_ReadObject( oid, offset, sizeof(struct S), &ud );
temp = ((struct S *) *ud)->x * 10;
sm_WriteObject( ud, 0, 4, &temp );
sm_ReleaseObject( ud );
```

Interacting with the EXODUS Storage Manager

Figure 4.6

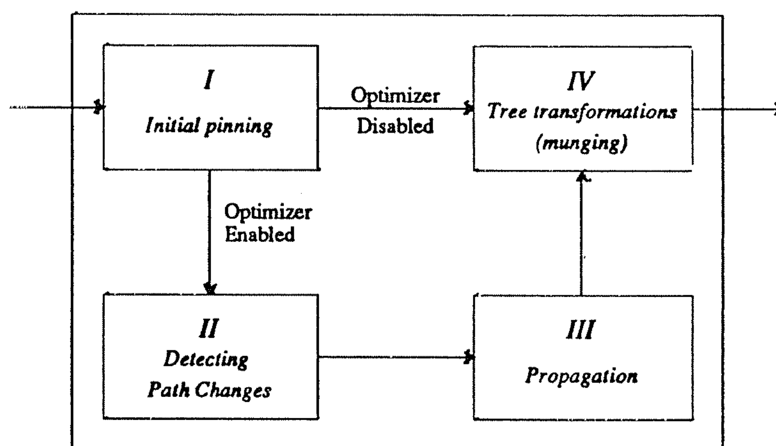
[38] Actually, the EXODUS Storage Manager provides *buffer groups*. A buffer group is a set of pages requested by a transaction and managed with a specified page replacement policy. The idea is to avoid interference in the paging characteristics of different transactions. Thus, if a transaction leaves data pinned, the performance degradation will tend to affect only the pinning transaction itself.

[39] The actual calls to the Storage Manager are slightly different than those shown here. Additional parameters include, for example, a transaction id and a buffer group specifier. Such parameters are not essential for this discussion and have been omitted for simplicity.

### 4.3.3. Overview of Code Generation

The input to the code generator is essentially a C syntax tree in which certain type nodes are decorated with the "db" attribute and certain variables are marked as having the persistent storage class. The output of the code generator is a C syntax tree in which the db-related constructs have been replaced with equivalent C constructs. In particular, the code generator must graft Storage Manager calls onto the tree at appropriate points, replace each use of a persistent variable with an expression to reference the object in the buffer pool, and transform arithmetic on db pointers into expressions involving OIDs and offsets. This section provides an overview of the code generator; Chapters 5 and 6 will present the details.

Basic code generation is split into two phases; if enabled, optimization introduces two additional phases. Figure 4.7 illustrates this organization. Phase I accepts a decorated C syntax tree and identifies the objects in the program that need to be pinned and unpinned. It also decides the points in the program where those pinning operations should occur. The output of this phase is the same tree that it received, except that sets of objects to be pinned and unpinned have been attached to the appropriate program nodes. These sets are called ***pin sets***, or simply, ***p-sets***. If optimization is not enabled, phase IV traverses the tree produced by the first phase. If a node ***n*** has a p-set of objects to be pinned, then for each object in the set, phase IV grafts a call to `sm_ReadObject` before ***n*** and a corresponding call to `sm_ReleaseObject` after ***n***. In addition, within the region of the program defined by ***n***, phase IV replaces each use of the object's value with an expression that dereferences through its user descriptor into the buffer pool to produce the value. Assignments to the object are replaced with calls to




Phases of Code Generation

Figure 4.7

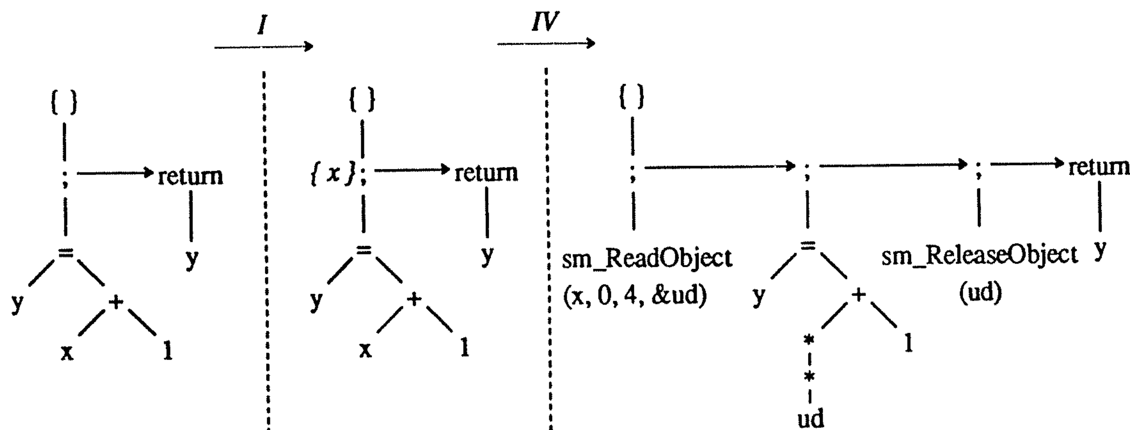
sm_WriteObject. Finally, phase IV is also responsible for transforming arithmetic on db pointers into expressions involving OIDs and offsets.

A very simple example will help to clarify this discussion. Consider the following function:

```
persistent dbint x;
int foo( ) {
        int y = (x + 1);
        return y;
}
```

The variable x is a persistent integer, and the function foo simply assigns the value of (x + 1) to a local variable y and then returns y's value. Figure 4.8 summarizes the processing performed by phases I and IV on the body of the function. The tree representation of the function body enters on the left. Phase I walks the tree and determines that the variable x should be pinned for the addition. Accordingly, it attaches a p-set containing x to the statement in which x is used. The transformation phase then traverses the tree again; on discovering the p-set associated with the first statement, this phase grafts pin and unpin operations for x around that statement. Finally, phase IV also replaces the use of x on the left-hand side of the addition with a dereference through x's user descriptor.

If optimization is enabled, then the output of phase I feeds into two intermediate phases. The aim of optimization is to reduce the number of Storage Manager calls executed by the compiled program. The basic strategy is to alter the plan created by phase I such that a given object remains pinned over many uses, rather than being pinned for each use and then immediately unpinned. We will accomplish this by moving the unpin operations




A Simple Example

Figure 4.8

from their originally scheduled locations to points in the program determined by the optimization phases. Phase II detects points at which objects must be unpinned due to the side effects of an assignment or procedure call. (Chapter 6 will explain why this is necessary.) Phase III detects points at which an object should be unpinned because there are no more uses of the object following that point. This phase changes the p-sets that were produced by phase I to reflect the new pinning plan. The syntax tree output by phase III, having the same form as the output of phase I, finally passes to phase IV, where the improved plan is translated into C code.

## 4.4. OTHER IMPLEMENTATION ISSUES

So far, this chapter has described the E compiler's internal organization and has outlined its general approach to implementing persistence. In this section, we consider two specific implementation issues that arise from the interaction of the semantics of C++ with the semantics of persistence.

### 4.4.1. Constructors and Destructors

An interesting problem arises when a persistent object is declared to be an instance of a dbclass that has a constructor. Consider again the binary tree class definition in Figure 3.10. The class's constructor initializes the tree's root node to NULL, indicating that the tree is empty. By definition, a constructor is called when the object is created. Suppose, as in Figure 3.11, a program declares a persistent tree instance. Since the compiler creates the persistent object, it would appear that the compiler must also invoke the constructor. But how is this to be accomplished? The compiler, that is, efront, has only the abstract syntax tree for the program. Should we write an interpreter? In general, a constructor may call other functions arbitrarily; what if those functions are externally defined? In general, unlike this example, it may be that the constructor itself has been declared, but not yet defined.

In the implementation of version 2.2 of the E compiler, the problem is handled as follows. Strictly speaking, it is not necessary to call the constructor at compile time. Rather, it is sufficient to ensure that the object is initialized before any program actually uses it, and that it is initialized only once. A very slight extension to an existing cfront mechanism provides a simple implementation satisfying these conditions. Before describing how constructors are called on persistent objects, then, we first review the mechanism used in cfront for constructing static objects.

Consider for a moment the C++ binary tree implementation in Figure 3.2, and suppose that a module[40] *m* (not necessarily the one containing `main()`) defines a tree instance `T` in the global scope. Any program which includes *m* as one of its components must be sure to initialize `T` before the main program begins. In the general case, a given program comprises a set of modules $M$ each of which contains a set $S_m$ of statically allocated objects

[40]The term "module" is used to emphasize that C++ programs are usually composed of separately compiled units.

that need initialization. The approach adopted in the AT&T C++ compiler involves first generating, as part of the C translation, an initializer function $f_m()$ for each module in $M$. The initializer function simply calls the appropriate constructor for each object in $S_m$:

```
fm()
{
        constructor( s1, args1 );
        constructor( s2, args2 );
        ...
}
```

In the case of the nonpersistent tree example, the initializer for module $m$ would look something like[41]:

```
void _STI_m( )
{
        _binaryTree_ctor( &T );
}
```

The first action of every C++ main program is to call the initializer function for every module in the set $M$. The steps enclosed in the dashed box in Figure 4.1 bind the calls made by the main program to the initializer functions.

E extends this mechanism as follows: In addition to the set of static objects $S_m$ an E source module contains a (possibly empty) set $P_m$ of persistent objects, each of which is of a dbclass with a constructor. To implement the desired semantics, efront amends the initialization function as shown in Figure 4.9. When $f_m()$ is called for the first time, the persistent flag has the value TRUE. The flag is then cleared and constructors are invoked on the persistent objects in $m$. If the same program is run again, or even if another program containing $m$ is run, these constructors

```
fm()
{
        persistent BOOL init = TRUE;
        if ( init ) {
                init = FALSE;
                constructor( p1, args1 );
                constructor( p2, args2 );
                ...
        }
        constructor( s1, args1 );
        constructor( s2, args2 );
        ...
}
```

Initializer for a Module with Persistent Objects

Figure 4.9

[41] The actual name of the function is the name of the source file prefixed by "_STI", for STatic Initializer.

will not be called again because the flag is itself persistent and shared by all programs that include *m*. Once the basic E persistence mechanism began working, this solution was trivial to implement; the compiler simply builds the function internally as a normal E function and then passes it through the usual compilation phases.

One shortcoming of this solution is that it does not extend to destructors. A destructor is the inverse of a constructor. Whenever an object of a class is destroyed, e.g. by going out of scope, the class's destructor (if it exists) is called first. A named persistent object is destroyed by deleting the module containing its persistent handle. If that object is of a dbclass with a destructor, then we should call that destructor. While persistent objects are initialized in the context of a running program (i.e. when all necessary code is available), the problem here is that they are essentially destroyed "out of context," (i.e. when all we have is a module containing persistent handles). We currently provide a utility program, *erm*, for destroying modules. *Erm* ensures that destructors are called properly, but it requires the user to specify explicitly any other modules that contain the code necessary for such calls.

#### 4.4.2. Virtual Functions

The C++ mechanism that supports "true" object-oriented behavior — the late binding of code to a method invocation — is the *virtual* function. If a member function of a class is declared to be virtual, then the run-time calling sequence involves an indirection through a dispatch table. In the AT&T C++ compiler, there is one such table for each class having virtual functions, and every object of the class contains a pointer to that table. Thus, the dispatch table is a kind of run-time type descriptor, and the table pointer embedded in each object of the class plays the role of a type tag. The amount of type information known to the compiler thus allows for a very fast implementation: a virtual function call adds one pointer dereference and one indexing operation to the cost of a normal procedure call.

Unfortunately, when virtual functions are combined with persistence, this implementation no longer suffices. We cannot store the virtual memory address of the dispatch table in a persistent object, as that address will be different for different programs. One approach is to make the dispatch table itself a persistent object, thus making the addresses embedded in the objects valid persistent addresses. This is the solution adopted in Vbase, for example [Andr88]. However, this approach implies a dynamic loading implementation, as the dispatch table is filled with the persistent addresses of the method code fragments; like the dispatch table and the objects themselves, methods are "faulted in" as the program runs.

For E, we have implemented a different solution. For every dbclass `C` having virtual functions, the compiler generates a unique integer type tag, and every instance of `C` contains this tag. The dispatch tables are still main memory objects, but in addition, we introduce a global hash table (also a main memory object) for mapping type tags to dispatch table addresses. Like the dispatch tables, the hash table is initialized at program startup; for each

dbclass in the program having virtual functions, we enter its type tag and dispatch table address into the global hash table. Then, to call a virtual function at run-time, we hash on the type tag in the object. Once we obtain the dispatch table address via hashing, the calling sequence then proceeds as before.

The current implementation computes the type tag as a 32-bit hash value based on various attributes of the class definition. The possibility of a collision between types within a program does exist, although remotely. For now, such collisions are simply detected at program startup and reported as an error. A complete solution, i.e. one *guaranteeing* uniqueness, requires a substantial programming environment design and implementation. Such an environment would have to distinguish between the first use of a given type and subsequent uses; the former case requires that a new tag be generated, while the latter would reuse the existing tag. Note that classes themselves would become objects in such an environment. We view such an environment as ultimately necessary, and we discuss it further in Chapter 8.

# CHAPTER 5

# CODE GENERATION

In Chapter 4, we outlined the basic approach that the E compiler takes in generating code to manipulate persistent objects. In this chapter and the next, we will describe this framework in detail. This chapter describes phases I and IV, while Chapter 6 details the optimization phases, II and III. For the sake of clarity, this chapter omits a few details from the description of phases I and IV; these details relate to optimization and will be provided in Chapter 6.

## 5.1. PHASE I: INITIAL PIN SCHEDULING

Phase I traverses the syntax tree of a function's body and collects information that is needed by later phases (II and III, as well as IV). As mentioned in Chapter 4, this includes identifying objects in the program that must be pinned as well as deciding initially where in the program those operations should be inserted.

In this section, we will describe the operation of phase I in detail. In order to make the discussion of code generation more concrete, however, we first introduce an example that will be used in this chapter and the next. Figure 5.1 shows the class definition and one class method for a doubly linked list implementation. Each linked list node comprises an integer data field, as well as pointers to the next and previous nodes in the list. The `insert` method performs a series of four pointer assignments to insert the node referenced by `that` between the node referenced by `this` and its following neighbor.

### 5.1.1. Identifying Common Subexpressions

The first task performed by phase I is the identification of common subexpressions (CSEs). In one pass over the syntax tree, phase I builds a dag representation of the expressions in the procedure. The difference between the dag and the syntax tree is that a given expression in the dag is shared by all contexts in which it appears; in the syntax tree, each occurrence is a separate copy of the expression. During phase I, each expression node in the tree is made to point to its common representative in the dag. As a result, later processing can easily check that two expressions are "the same" by seeing if they point to the same representative. Although techniques for building such dag representations are well known [ASU86], there are a few small complications in this environment that should be noted. First, in building a dag from intermediate code, one is usually limited to expressions that are either named variables or arithmetic operations. The expressions in a C syntax tree also include operators that dereference through a pointer (`*`), index into an array (`[]`), and select a field from a record (`.`). Furthermore, C allows several different syntactic forms to denote the same object. For example, since arrays and pointers are equivalent, `a[n]`

```
dbclass dll {
      dbint data; // data field
      dll * prev; // pointer to previous node
      dll * next; // pointer to next node
public:
      ...
      void  insert( dll * ); // insert new node
};

void  dll::insert( dll * that ) {
      that->next        = this->next;
      that->prev        = this;
      this->next->prev  = that;
      this->next        = that;
}
```

Doubly Linked List Example

Figure 5.1

and `*(a+n)` are equivalent expressions.

Because CSE identification is based on matching syntactic forms, all such constructs are first normalized:

(1) The expression $e_1[e_2]$ becomes $*(e_1+e_2)$ if $e_1$ is actually of a pointer type.

(2) The expression $*(e_1+e_2)$ becomes $e_1[e_2]$ if $e_1$ is actually of an array type.

(3) The expression $e_1 \rightarrow mem$ becomes $(*e_1).mem$ in all cases.

Fortunately, these additional operators present no real difficulty; the dag-building algorithm simply considers (`[]`) and (`.`) to be additional binary operators, and (`*`) to be another unary operator.

Another problem is that, unlike languages such as Pascal, C considers assignment to be an expression and not a statement. This feature leads to the possibility of such constructs as `(p = f())->x`, meaning, call `f`, assign the result to `p`, and dereference through that pointer to obtain the field `x`. The problem here is that CSE information is used to eliminate redundant pinning by detecting when the same object is being referenced in two different places. If an expression has a side effect, then eliminating the evaluation of one instance of the expression could result in an incorrect program. For example, suppose a program contains the expression:

```
( (p = f())->x  +  (p = f())->x )
```

It would be incorrect to produce code that pins `x` once and uses the pinned value for both sides of the addition. We must assume that `f` returns a different value for each call[42], and hence, the two operands of the addition denote *different* objects. For these reasons, the CSE identification algorithm considers every expression having side effects

[42] Interprocedural dataflow analysis could perhaps improve this conservative estimate.

to be a unique expression, even if the operands are the same. Such expressions include assignment in all its various forms (e.g. =, +=, ++, etc.) as well as all function calls.

The last issue complicating CSE identification involves scoping. Since every name in the leaves of an expression tree is defined within a particular scope, the expression itself is defined only within a particular scope. Specifically, a CSE is associated with the most deeply nested scope of any of its operands. For example, consider the following program fragment.

```
void f( dbint * p )
{
    {
      int   n;
      ... *(p + n) ...
    }
}
```

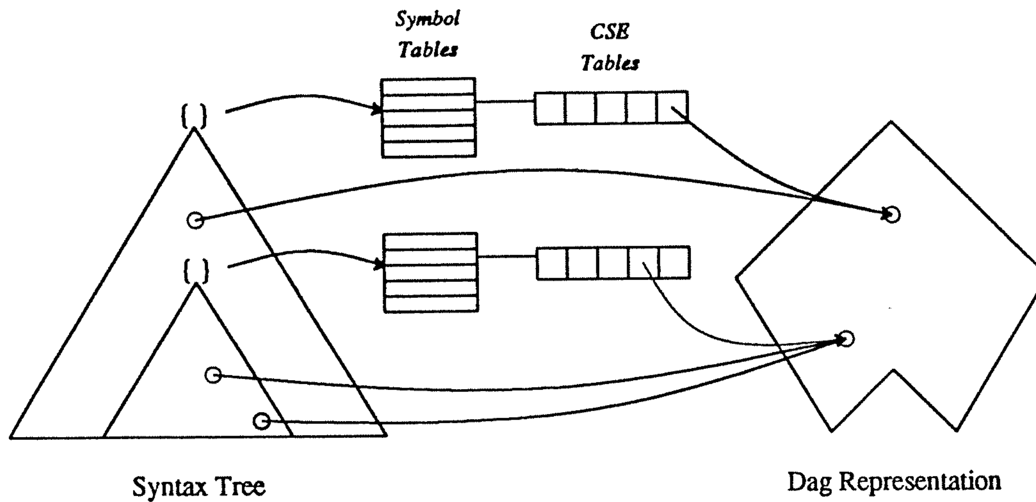
The name expression `p` is associated with the scope of the function's body, while the expressions `n`, `p+n`, and `*(p+n)` are all associated with the scope of the inner block. If a function references a global variable by name, that variable is temporarily "imported" and associated with the scope of the function body.[43] Constants (literal values) are also associated with the scope of the function body.

Every symbol table owns a table of the CSEs that are associated with its scope. Figure 5.2 shows this relationship. When a new CSE is detected, it is given an identification number and added to the appropriate table. Id numbers start at zero for each scope and are assigned in order of CSE identification. The *global* id for a given CSE is determined by adding the number of CSE's in all enclosing scopes to the CSE's local id. CSEs in mutually exclusive scopes may have the same global id.

Since algorithms for constructing dag representations are well known, and since the complications we described have fairly straightforward solutions, we will omit giving the full algorithm here. Instead, Figure 5.3 shows only the two most important procedures: `enterScope` and `matchCse`. `EnterScope` builds a new CSE table when we enter a new lexical scope and immediately adds all of the locally declared names to that table. `MatchCse` accepts an operator (token) and its associated CSE operands. If a CSE having the given operator and operands already exists, it is returned. Otherwise, `matchCse` builds a new CSE, enters it into the appropriate table (based on the scopes of the operands), and then returns it. Obviously, we are only sketching the algorithms here; there are many details in the actual implementation that are not of interest for this discussion. For example, the `matchCse` procedure shown in Figure 5.3 only handles binary operators, when in fact, we must handle unary and ternary operators as well.

---

[43] Thus, the "most global" scope for any item is a function body. If we were to introduce interprocedural analysis into the compiler, then this aspect of the implementation would have to change.




Syntax Tree and Dag Representations

Figure 5.2

Let us examine the process of collecting CSEs in the context of the linked list insertion example given earlier. Figure 5.4 illustrates the resulting data structures after the first two statements have been processed. When we enter the scope of the procedure, `enterScope` adds the locally declared names. In this example, the implicit parameter `this` receives the id 0, and the parameter `that` receives id 1. Now, when we descend the expression tree for the first assignment, we first discover the leaf node, `that`, and find that it is already a CSE whose id is 1. Returning to next higher level, we look for a CSE whose operator is `*` and whose operand is CSE 1. Finding none, we identify `*that` as a new CSE, and assign it the next available id, 2. In addition, the instance of `*that` in the syntax tree is made to point to the newly created CSE. Continuing in this fashion through the first two assignments in the procedure, we obtain the labelings as shown in the figure. For clarity, we do not show all of the arcs actually present. Instead, we only show the two occurrences of `*that` pointing to their common representative CSE.

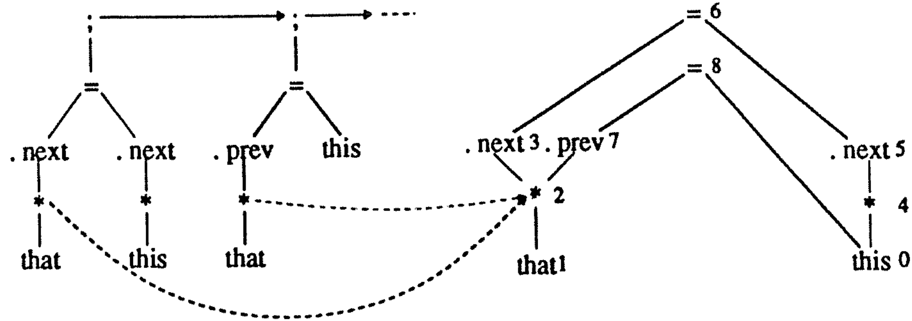
### 5.1.2. Items

For the purposes of identifying CSEs, all expressions in the program are of interest. However, for the task of pinning and unpinning objects, we will only be interested in a subset of the CSEs called the *items*. An item is an Lvalue, i.e. an expression denoting an addressible object. Informally, an item is any expression that may legally appear on the left hand side of an assignment. An item is easily identified by its syntactic form. First, any named variable is an item. Next, if $e_1$ is a pointer valued expression, then $*e_1$ is an item. If $e_2$ is a structure valued expression, then $e_2.mem$ is an item, where *mem* is the name of a field in the structure. Finally, if $e_3$ is an array valued expression and $e_4$ is an integer valued expression, then $e_3[e_4]$ is an item. Thus, given an expression tree,

```
void  enterScope( symbolTable stbl )
{
      create a new cse table, cseTbl, and attach it to stbl;
      initialize number of entries, nEntries, to 0;
      foreach( name in stbl )
      {
            enter new cse corresponding to name into cseTbl;
            initialize id# of cse to nEntries;
            make name point to cse;
            increment nEntries;
      }
}


cse  matchCse( token op, cse cse1, cse cse2 )
{
      cseTbl = of cse1's table and cse2's table, the one
              of deeper lexical level;

      if( op is an assignment or function call )
      {
            create new cse = <op, cse1, cse2> in cseTbl;
            return cse;
      }
      else
      {
            look in cseTbl for cse == <op, cse1, cse2>

            if( found )
                  return cse;
            else
            {
                  create a new cse = <op, cse1, cse2> in cseTbl;
                  return cse;
            }
      }
}
```

Procedures for Collecting CSEs

Figure 5.3




CSE's from the Insert Method

Figure 5.4

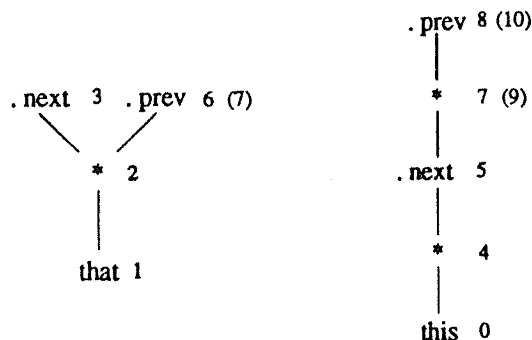
one need only look at the operator in the root node to determine if the expression is an item. Note that an item may of course have subexpressions that are not items. For example, if `p` is a pointer, then `*(p + 1)` is an item, but `(p + 1)` is not.

Because the scheduling of pin and unpin operations is concerned exclusively with items (as opposed to general CSEs), the E compiler's implementation is optimized for manipulating sets of items. Within a given scope, the items are distilled out of the general pool of CSEs and given *local item ids*, again starting at 0 for each scope. Thus, if there are n items in a given scope, they will have local item ids ranging from 0 to n-1. Figure 5.5 shows the items from the linked list example, each labeled with its local item id. For those items whose item id differs from its CSE id, the latter value is also given in parentheses. As with CSEs, the global id for a given item is its local id plus the number of items in all enclosing scopes; items in mutually exclusive scopes may again have the same global item id. Because the items in Figure 5.5 all belong to the scope of the function body, their local ids are also their global ids.

By renaming items as described above, it is then possible to represent sets of items compactly with bit vectors. Like CSEs and items, item sets are associated with scopes. An item set may contain items from its scope or from any enclosing scope, and the numbering of the bits in the set's representation corresponds to *global* item ids. For example, the item set associated with the scope of the insert routine requires nine bits, as there are nine items in that scope. The set { 100 011 000 } represents the items `this`, `*this`, and `(*this).next`.

### 5.1.3. Initial Pin Scheduling

In addition to identifying common subexpressions and items, phase I also decides on an initial plan for pinning and unpinning items. As we saw in the overview in Section 4.3.3, this plan is represented by sets of items, called the p-sets, attached to certain nodes in the syntax tree.




Items from the Insert Method

Figure 5.5

#### 5.1.3.1. Detecting Items to Pin

The algorithm for detecting which items to pin is essentially the same as that used in the code generator for the first E compiler [Rich89]. (We will review this algorithm shortly.) The major difference lies in the action taken when such an item is discovered. Formerly, the syntax tree was immediately transformed; a Storage Manager call to read the object along with a dereferencing expression to produce the object's value combined to replace the item's occurrence in the tree. In phase I of the current code generator, however, there are *no tree transformations*. Instead, items that are to be pinned are simply added to the p-sets of certain nodes in the tree. Deciding where those sets should be located will be discussed in the next section. For now, let us concentrate on finding the items.

Assume for the moment that `x` is a persistent integer. The mere occurrence of `x` in the syntax tree does not necessarily imply that `x` should be pinned. For example, if `x` occurs in the expression `(&x)`, then clearly, all that is desired is the location of `x`, which is available in `x`'s companion. In general, then, whether an item should be pinned or not depends on the context in which it is found. Accordingly, as the expression tree is traversed during phase I, each call level passes a parameter to the next call level indicating whether the subexpression is intended to produce an Lvalue (an address) or an Rvalue (a data value).

Figure 5.6 shows the algorithm for detecting which items to pin. (The cases shown are representative rather than exhaustive.) Although the algorithm is written as if it were an independent pass over the tree, we note again that the tasks of identifying CSEs and items, of deciding which items to pin, and of deciding where initially to schedule the pins are all performed in one pass. The first parameter to `detectPins` is `ex`, a pointer to an expression node in the syntax tree. If the expression is a name, then the `base` field contains the token `NAME`; otherwise, `base` contains the operator token for the expression. In the latter case, the node also contains fields `e1` and `e2`, which are pointers to the operands of the expression. (For unary operators, `e2` is unused.) The second

```
void detectPins( ex, context )
{
        switch( ex->base )
        {
        case NAME:
                if( NAME's type is not db )
                        break;
                if( NAME is persistent or extern
                &&  context == Rvalue )
                        schedulePin( ex );
                break;

        case DOT:       /* (e1.mem) */
                detectPins( ex->e1, Lvalue );
                if( mem's type is db
                &&  context == Rvalue )
                        schedulePin( ex );
                break;

        case DEREF:     /* (*e1)     */
                detectPins( ex->e1, Rvalue );
                if( ex->e1 is pointer to db type
                &&  context == Rvalue )
                        schedulePin( ex );
                break;

        case ADDROF: /* (&e1)     */
                detectPins( ex->e1, Lvalue );
                break;

        case ASSIGN: /* (e1 = e2)       */
                detectPins( ex->e1, Rvalue );
                detectPins( ex->e2, Rvalue );
                break;

        case PLUS:    /* (e1 + e2)       */
                detectPins( ex->e1, Rvalue );
                detectPins( ex->e2, Rvalue );
                break;

        /* etc... */
        }

        return;
}
```

Detecting Items to Pin

Figure 5.6

parameter to the routine is `context`, an enumerated type having one of the values Lvalue or Rvalue. When the algorithm detects an item that should be pinned, it calls the routine `schedulePin`, which adds the item to the

appropriate p-set.

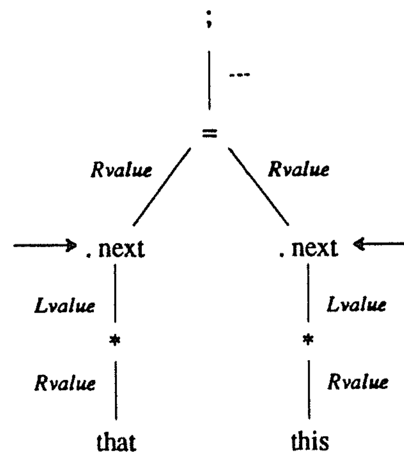
As an example of the operation of `detectPins`, consider the first assignment statement in the linked list insertion routine. Figure 5.7 shows the syntax tree for this assignment. Each arc in the tree is labeled with the value of the `context` parameter for the associated invocation of `detectPins`. Arrows point to the items that the algorithm decides should be pinned.

### 5.1.3.2. Deciding Where to Pin Items

Given that we are able to identify *what* should be pinned, we must now decide *where* those pin operations should be inserted into the program. Although it may appear that we can simply pin the items needed by a statement right before entering that statement, we must beware of certain subtleties involving side effects and flow of control through C expressions. According to Kernighan and Ritchie,

> ...the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified. [K&R78, p. 185]

These comments apply to expressions involving such operators as `+`, `*`, and `=`. There are a few expressions, however, in which the evaluation order *is* defined. In particular, the comma operator explicitly specifies sequential




Example of Detecting Items to Pin

Figure 5.7

left-to-right evaluation; in $(e_1,e_2)$, for example, $e_1$ is evaluated before $e_2$.[44] The conditional operator is another example; in $(e_1?e_2:e_3)$, $e_1$ is evaluated first, followed by the evaluation of either $e_2$ or $e_3$, depending on $e_1$'s value. Finally, the boolean connectives *&&* (AND) and || (OR) are defined to have short-circuited evaluation; in $(e_1\&\&e_2)$, $e_1$ is evaluated first, and only if the result is nonzero is $e_2$ then evaluated.

These control flow considerations are important for deciding where to pin items in the tree, as the process of pinning an item involves evaluating the expression that denotes the item's location. Consider for example, the item `p->x`. If this were to appear in

```
( p = f(), p->x = 1 )
```

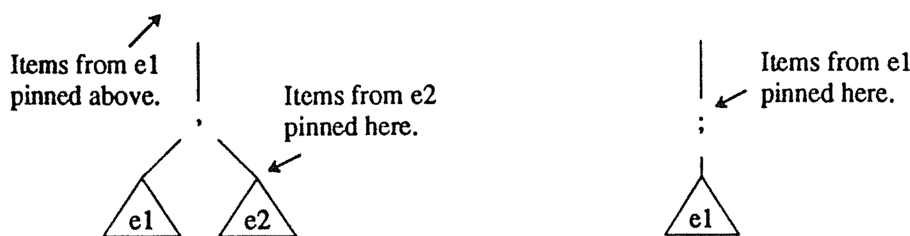
it would be incorrect to pin the item before the comma expression; the programmer has been careful here to define the path to `p->x` before traversing it, i.e. by assigning a value to `p`. If, however, the item instead appears in

```
(( p = f(), y ) + p->x )
```

then the compiler is free to pin `p->x` before the expression. Here, the programmer has, in all likelihood, assumed incorrectly that the operands to + will be evaluated from left to right.

Based on these issues, phase I of the code generator decides where to schedule pin operations as follows. During its recursive descent of the syntax tree, this pass looks for nodes that specify flow of control; such nodes include all statements as well as the expression types mentioned above (e.g. comma expressions). When such a node is encountered, phase I pushes either that node or the appropriate *child* of that node onto a stack called *flowNodes*. Later, when it discovers an item that must be pinned, phase I adds that item to the p-set associated with the top node on the flowNodes stack. The reason for pushing "either that node or the appropriate child" may be seen through two examples illustrated in Figure 5.8. Assume that phase I encounters a comma expression. The correct sequence of events is to process the left subtree (`e1`) recursively, then to push the root node of the right subtree (`e2`) onto flowNodes, then to process the right subtree, and finally, to pop the stack. We do not push the comma node itself for two reasons. First, for items discovered in `e1`, pinning them only for the duration of the comma expression is more limiting than is necessary; we may legitimately pin those items higher up in the tree. Second, and more importantly, for items discovered in `e2`, pinning them before entering the comma expression is incorrect, as explained. Now assume that phase I encounters an expression statement, i.e. a statement consisting of an expression followed by a semicolon. In this case, we could correctly attach the p-set for `e1` either to the statement node or to the root node of the expression; the order of events during execution would be the same in either case. We chose the former, however, since it ultimately produces better code. Here, "better" means that the output has smaller expressions, since the pin and unpin operations become separate statements, rather than being welded into

[44] Note that the commas separating arguments in a function call are *not* comma operators; the order of evaluation of function arguments is undefined [K&R78, p. 186].




Illustrating Where Items Are Pinned

Figure 5.8

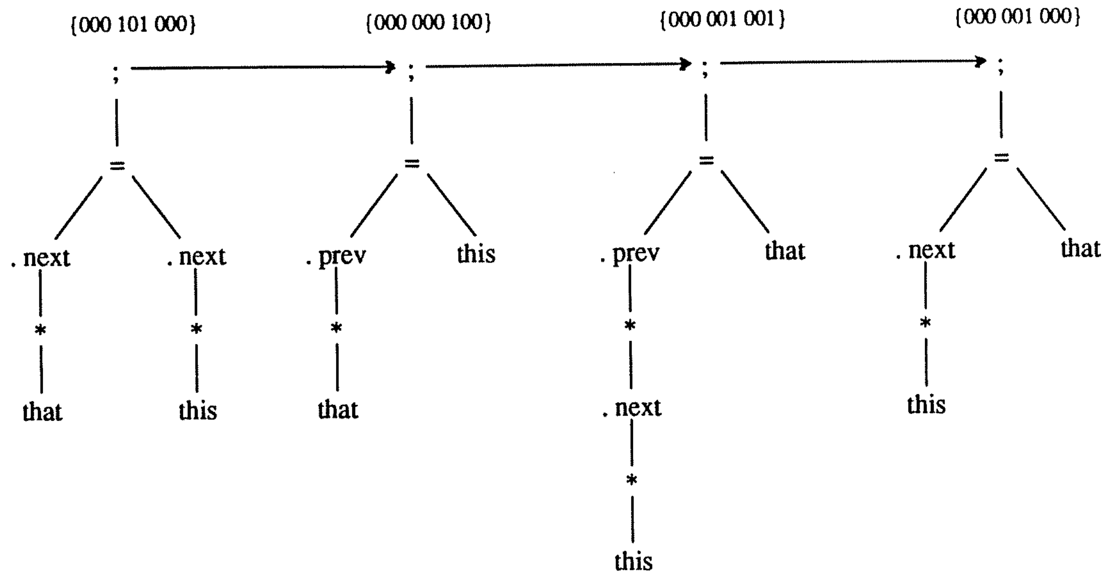
one very large expression. Not only does this make debugging the compiler easier, but it also avoids the problem that some C compilers have (rather arbitrary) limits on the size of the expressions that they can handle.

In light of the preceding discussion, the linked list insertion routine has quite a simple structure. In its series of four assignments, none of the expression operators specifies control flow; therefore, all pinning of items will occur at the statement level. Figure 5.9 shows the output of phase I. Attached to each of the four statement nodes is a p-set for that node. The first statement, for example, has a p-set containing `(*that).next` and `(*this).next`. Note that the p-set for the third statement contains `(*this).next` (item 5) and `*((*this).next).prev` (item 8). That is, before the assignment, we must pin item 8, which is the `prev` field from the object following the one referenced by `this`. In order to do this, however, we must first pin item 5, which is the `next` field in the object referenced by `this`.

## 5.2. PHASE IV: TRANSFORMING THE SYNTAX TREE

If optimization is not enabled, then the output of phase I feeds directly into phase IV of the code generator. In this section, we describe the process by which phase IV transforms an E syntax tree (decorated with p-sets) into a C syntax tree. The major tasks in this process include: generating code fragments (*sprigs*) to pin, unpin, read, and write the items found in the p-sets; grafting the sprigs onto the tree at the appropriate points; and transforming db pointer arithmetic into DBREF manipulations.

There are several issues that affect the design of phase IV. First, the items to be pinned are not necessarily persistent; E allows db type variables to have any of the normal storage classes in addition to **persistent**, and db type objects may also be allocated from the heap (i.e. with the **new** operator). In some cases, the compiler knows for sure whether an item is persistent or not. For example, if the item is a named variable, the compiler can simply look in the symbol table to find its storage class. In many (perhaps most) cases, however, the persistence of an item cannot be determined until run-time. For instance, in our linked list example, the compiler cannot tell if the pointers passed into the insertion procedure refer to persistent or to volatile objects. Therefore, the sprig that implements a




*Legend of Items*

| | | | | | |
|---|---|---|---|---|---|
| 0 | this | 3 | (*that).next | 6 | (*that).prev |
| 1 | that | 4 | *this | 7 | *((*this).next) |
| 2 | *that | 5 | (*this).next | 8 | *((*this).next).prev |

Phase I Pinning Plan for the Insert Method

Figure 5.9

pin operation must handle both possibilities. This same problem also affects the sprig to reference or to write a pinned item.

Another issue that affects the design of phase IV is that items are expressions of rather arbitrary composition. A given item may involve a chain of pointer dereferences, for example; in order to pin such an item, we must first pin each intermediate pointer in the chain. We just saw an example of this in Figure 5.9 with the item `*((*this).next).prev`. Another item may involve pointer arithmetic, e.g. `*(p + n);` in order to pin this item, we must first transform the arithmetic into DBREF manipulations. The point here is that generating the sprig to pin an item in a p-set — one of the tasks of phase IV — may require a recursive call of phase IV on that item in order to obtain its DBREF address.

### 5.2.1. The Functions genSprigs( ) and mungeTree( )

Phase IV is implemented with a pair of mutually recursive procedures, `genSprigs` and `mungeTree`.[45] The routine `genSprigs` takes an item as a parameter and builds a set of code fragments (the sprigs) for that item. These sprigs implement the basic operations needed for manipulating the item at run-time. The ***pinning sprig*** pins the item; it includes a call to the Storage Manager routine `sm_ReadObject`. The ***unpinning sprig*** unpins the item; it contains a call to `sm_ReleaseObject`. The ***reading sprig*** produces the item's value for use in an expression; it involves dereferencing through the item's user descriptor. Finally, the ***writing sprig*** assigns a new value to the item; it contains a call to `sm_WriteObject`.[46]

In building the pinning sprig, `genSprigs` must include arguments with the Storage Manager call that provide the OID and offset of the item. In order to do this, `genSprigs` must have access to an expression giving the DBREF address of the item. Such an expression can be produced by the routine `mungeTree`. Thus, in the course of building the pinning sprig for an item, `genSprigs` calls `mungeTree`, passing it the item and requesting the item's Lvalue.[47]

MungeTree is the "top level" routine of phase IV. That is, phase IV begins when the compiler calls `mungeTree`, passing it the syntax tree produced by phase I. `MungeTree` traverses this structure and performs the following tasks:

(1) If a node in the syntax tree has a p-set, then `genSprigs` is called for each item in the set. `MungeTree` then grafts the pinning and unpinning sprigs before and after the node, respectively.

(2) If an item occurs in the tree, and that item is pinned, then that occurrence is replaced with either a reading or a writing sprig, as appropriate for the context. For example, if the item appears on the left hand side of an assignment, then a writing sprig replaces the occurrence (and actually replaces the assignment itself). However, if the item appears on the right hand side, then it is replaced by a reading sprig.

(3) If an expression performs arithmetic on a db pointer, then it is replaced with an expression that manipulates a DBREF structure.

Together, the routines `genSprigs` and `mungeTree` transform the E syntax tree into a C syntax tree. Now that we have outlined what the two routines do and how they interact, let us describe each one in more detail.

---

[45] **munge** ( munj ) *v.* 1. *slang* to fold, spindle, or otherwise mutilate; in this case, to transform the syntax tree into a form no longer recognizable as the original.

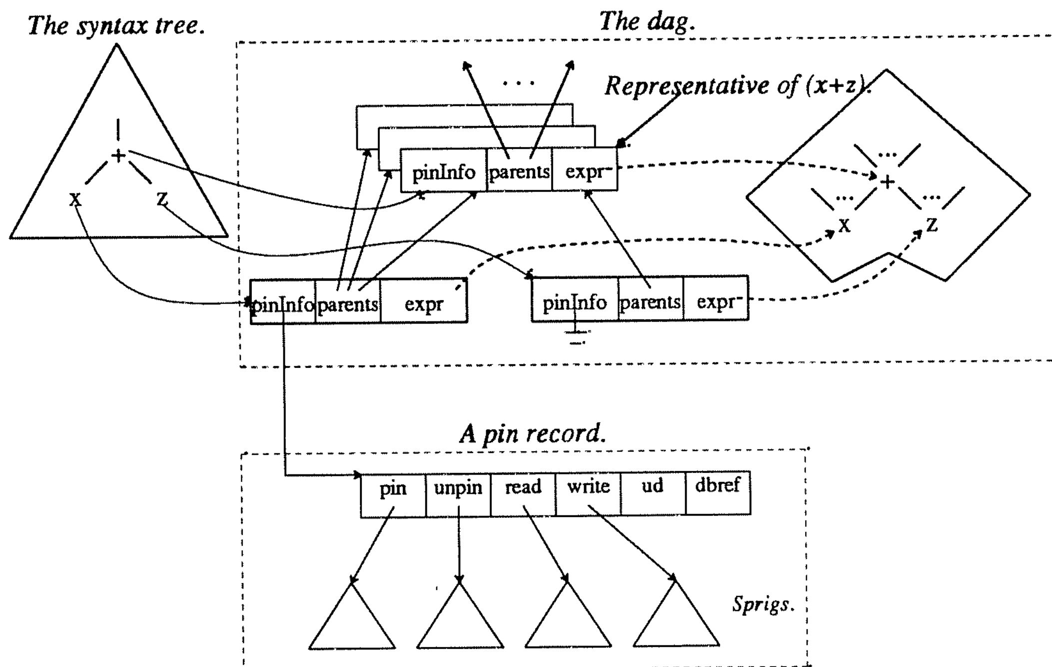
[46] For detailed implementation reasons, writing sprigs are not generated until they are actually needed for grafting onto the tree.

[47] Since `mungeTree` alters the expression tree passed to it, `genSprigs` actually passes a *copy* of the item.

### 5.2.2. Generating Code Sprigs

In this section, we describe the form of the code fragments built by `genSprigs`. Before we can proceed, however, we must first explain in more detail the data structures that connect an expression in the syntax tree with its representative in the dag built by phase I.

As shown in Figure 5.10, an expression's representative is a data structure that maintains several pieces of information, one of which is a pointer to the CSE, i.e. to the expression nodes in the dag. This pointer is represented as a dotted arc in Figure 5.10. Note that the expression nodes in the dag also point (back) to this representative, although for clarity those arcs are not shown in the figure. A representative also maintains a pointer to every CSE that is an immediate parent of this CSE. Given this structure, we may find all subexpressions of a CSE by traversing down through the subexpression arcs, and we may find all supexpressions (i.e. expressions containing this CSE) by traversing up the parent arcs. For example, starting at the `+` node in Figure 5.10, we can find the subexpressions `x` and `z`. Conversely, starting at `x`, we can traverse upward along `x`'s parent arcs to find that `x` is a subexpression of `(x + z)`. Lastly, although not shown in Figure 5.10, we note that a CSE's representative node in the dag also includes the CSE's local id and a pointer to the CSE table to which it belongs.




Detail of Data Structures for CSEs and Items

Figure 5.10

When `genSprigs` is called on an item, it creates a structure called a *pin record* which it then attaches to the item's representative in the dag. This structure serves mainly to store the code fragments built by `genSprigs` for later retrieval. It also holds the names of two variables needed for building the sprigs. The first variable, labeled "ud" in Figure 5.10, is the name of a temporary variable introduced by the compiler to serve as the user descriptor argument to the various Storage Manager calls that are generated. This variable is declared to be of type `int*`.[48] The second variable recorded in the pin record is the DBREF variable that contains the item's address. As stated earlier, `genSprigs` invokes `mungeTree` in order to obtain the address of the item. `MungeTree` returns an expression that computes this DBREF along with the name of the variable that will contain this value at run-time. `GenSprigs` then records the variable's name in the pin record. For example, if the item being pinned is `*p`, then the expression to "compute" the address of the item is simply the variable `p`. If the item is instead `*(p+1)`, then the expression assigns `p` to a temporary and increments the temporary's offset.

#### 5.2.2.1. The Pinning Sprig

After `genSprigs` has obtained the name of a DBREF that will hold the item's address at run-time, it builds a Storage Manager call to pin the item. The OID and offset arguments are supplied by the DBREF. The length parameter is a constant equal to the size of the type of the item being pinned. As we stated above, the user descriptor argument is a pointer (temporary) declared by the compiler. The declaration has the following form:

```
int * _tmpUd001    = NULL;
```

The initialization of this pointer to NULL is important for reasons that will be explained in Chapter 6. The first approximation of a pin operation, then, is an expression the form:

$pin_0$ `::= sm_ReadObject( dbref.oid, dbref.offset, length, & _tmpUd001 )`

Every Storage Manager call returns a status value; zero indicates success, and any other value indicates an error of some kind. E programs check every call to `sm_ReadObject`, and if there is an error, the program is aborted. Thus, we modify the pin operation as follows:

$pin_1$ `::= ((__E_error =` $pin_0$`) ? __Eabort( __E_error ) : 0 )`

`__E_error` is a global integer variable in all E programs, and it records the status value returned from Storage Manager calls. `__Eabort` is a library routine that reports the error, aborts the transaction in the Storage Manager, and exits the program.

As we have mentioned previously, the compiler cannot generally tell if the item being pinned is persistent or not. If the item is *not* persistent, then the DBREF contains an OID that indicates virtual memory and an offset that

[48] Recall that a client of the Storage Manager passes the address of a pointer in the `sm_ReadObject` call. On return, the pointer references a user descriptor inside the Storage Manager.

contains the item's virtual address. To handle this case, the compiler allocates a second temporary:

```
int    _tmpInMem001;
```

and the pin operation is modified as follows.

```
pin2 ::=
    ((_tmpInMem001 = INMEM( dbref.oid )) ?
        (_tmpInMem001 = dbref.offset, _tmpUd001 = & _tmpInMem001)
        : pin1);
```

INMEM is a macro that does a simple test (an integer comparison) on a portion of the OID and returns TRUE if the OID references a nonpersistent object. Thus, the pin operation first checks if the OID indicates a virtual address and then assigns the boolean result to `_tmpInMem001`. If the result is TRUE, then `_tmpInMem001` is reassigned with the virtual address of the object, and `_tmpUd001` is made to point to `_tmpInMem001`. (Note that after the reassignment, `_tmpInMem001` is still nonzero, i.e. TRUE.) If the result is FALSE, then the operation calls the Storage Manager.

The reason for designing the pinning sprig in this way is that no matter which branch is taken during the pin, read access to the object is uniformly a double indirection through `_tmpUd001`. The alternative would be to perform another check at the point of access to see which branch was taken in the pin. If the Storage Manager branch was taken, the access would perform the double indirection, and if not, it would perform a single indirection using the virtual address in the DBREF. This alternative has two disadvantages. First, we would essentially be making the same test several times instead of once, i.e. not only at the pin, but also at every access. Second, on at least some architectures, a double indirection can be performed in one machine instruction; the second alternative would require a test, a branch, and *then* either a single or a double indirection.

### 5.2.2.2. The Unpinning Sprig

The unpin operation is a call to the Storage Manager routine `sm_ReleaseObject`. This call is needed only if the pin operation took the Storage Manager branch. Accordingly, an unpin has the form:

```
unpin0 ::= ( _tmpInMem001 ? 0 : sm_ReleaseObject( _tmpUd001 ) )
```

That is, if the object is in virtual memory, do nothing; otherwise call the Storage Manager to release it.

### 5.2.2.3. The Reading Sprig

As explained above, read access to a pinned item always follows a double indirection. Assume the item is of type T. The reading sprig then has the form:

```
read0 ::= *( (T*) *_tmpUd001 )
```

That is, this operation dereferences through `_tmpUd001`, casts the result as a `T*`, and then dereferences one more time.

When we discuss coalescing in the next chapter, we will see that a read operation will sometimes require an additional offset into the pinned range. In such cases, the operation will have the form:

```
read1 ::= *( (T*) (*_tmpUd001 + offset) )
```

Since `_tmpUd001` is declared as an `int*`, the addition is integer addition, and the offset thus specifies a number of bytes.

#### 5.2.2.4. The Writing Sprig

Unfortunately, unlike the read operation, the writing sprig cannot be made uniform with respect to the persistence of the item being written. If the pin operation took the in-memory branch, then we must update the item directly through the pointer. If the pin operation took the Storage Manager branch, however, we must call `sm_WriteObject` to update the object. Parameters to this routine include a pointer to the user descriptor associated with the pinned byte range, an offset into the range, and the number of bytes to be written. Furthermore, this call requires as a parameter the *address* of the new data to be written into the object. As a result, we must often copy the value to be assigned into a temporary location so that we may pass the address of the temporary in the call. For example, if `X` is a persistent integer, then in the assignment `X = 0`, we must first assign 0 to a temporary and then pass the address of the temporary to `sm_WriteObject`.

Assume that the item being updated has type `T`, and that <expr> is the value to be written. The compiler allocates a temporary, `_tmpVal001`, of type T. The write operation then has the form:

```
write0 ::=
        ( _tmpVal001 = <expr>,
          (_tmpInMem001 ?
              ( read1 = _tmpVal001 )
              : sm_WriteObject( _tmpUd001, offset, size, &_tmpVal001 )))
```

If the object is in virtual memory, we assign to it directly, so the left hand side of the assignment has the same form as the read expression for the item. If the item is persistent, we call the Storage Manager instead. The `offset` parameter has the same value as in the reading sprig, and `size` is a constant equal to the size of `T`.

### 5.2.3. Grafting the Sprigs

The code fragments generated by `genSprigs` are grafted onto the syntax tree by the routine `mungeTree`. As we stated earlier, `mungeTree` is actually the top level routine of phase IV, and its job is to traverse the tree, transforming it into a C syntax tree. This job involves calling `genSprigs` for the items found in p-sets, inserting the code produced by `genSprigs` at appropriate points, and translating db pointer expressions into DBREF expressions. Figures 5.11 and 5.12 show the `mungeTree` routine. (The procedure is split between the two figures

for clarity of presentation.) MungeTree takes as its first argument a pointer to an expression[49], and it returns a pointer to the transformed expression. Thus, when mungeTree calls itself recursively (i.e. to descend the expression tree), the form of the call is:

```
ex->e1 = mungeExpr( ex->e1, ... );
```

Here, mungeTree recursively processes the subexpression referenced by e1 and replaces e1 with a pointer to the result.

As was the case with the phase I routine detectPins, each invocation of mungeTree receives parameters from the caller indicating what kind of transformation is required. For example, like detectPins, mungeTree has a context parameter indicating whether the transformation is to produce the Lvalue or the Rvalue of the expression. In addition, mungeTree takes a third argument, RorW. The value of this argument is either READ or WRITE, and it is used when mungeTree replaces a pinned item's occurrence in the syntax tree. If the item's value is being used in an expression, then RorW will have the value READ, and mungeTree will replace the item with its reading sprig. If the item is being assigned, then RorW will have the value WRITE, and mungeTree will replace the item with its writing sprig.

We may now summarize the actions of an invocation of mungeTree. As shown in Figure 5.11, the first act is to check the expression node to see if it has an attached p-set; if so, then mungeTree calls genSprigs on each item in the set. On return, the pin record for each item contains the different code sprigs that will be needed later. In the next step, mungeTree checks to see if the given expression is a pinned item, that is, one for which a pin record currently exists. If so, and if the current context requires the item's Rvalue, then mungeTree immediately returns either the item's reading sprig or its writing sprig, depending on the value of the parameter RorW. If the context requires the item's Lvalue, then processing continues on to the main body of mungeTree, shown in Figure 5.12. For the moment, let us postpone considering that part of the procedure and continue instead with Figure 5.11. Assume that we have processed any subexpressions and that we are now ready to complete the invocation at this level. If genSprigs was called at the beginning of the invocation, then mungeTree grafts the pinning and unpinning sprigs onto the tree. MungeTree grafts each pinning sprig by building a comma expression in which the sprig is the first operand and the expression is the second; unpinning sprigs are grafted similarly, except that the order of the operands is reversed in the comma expression. MungeTree then calls the procedure delSprigs which deallocates certain data structures built by genSprigs. Finally, mungeTree terminates the invocation by returning the transformed expression.

[49] MungeTree is really two routines, mungeStmt and mungeExpr. The actions of mungeStmt (not shown) are similar enough to those of mungeExpr that they do not need an independent explanation.

```
expr * mungeTree( ex, context, RorW )
{
      if( ex has a p-set )
            foreach( item in p-set )
                  genSprigs( item );

      if ex is the root node of a pinned item
      &&  context == Rvalue )
      {
            if( RorW == READ )
                  return the item's reading sprig;
            else
                  return the item's writing sprig;
      }

      /*
      ** Process subexpressions recursively here.
      ** Also transform db pointer arithmetic.
      ** (See Figure 5.12.)
      */

      if( ex has a p-set )
            foreach( item in p-set )
            {
                  let p = the item's pinning sprig;
                  let u = the item's unpinning sprig;
                  ex = ( p, ex );    // prepend the pin
                  ex = ( ex, u ); // append the unpin
                  delSprigs( item );
            }

      return ex;
}
```

The Procedure MungeTree (part 1)

Figure 5.11

Figure 5.12 shows the remainder of the body of the `mungeTree` routine. It is here that the routine processes subexpressions and converts db pointer manipulations into DBREF manipulations. The `context` parameters that are passed in the recursive calls match the values passed in the `detectPins` routine. For example, if the current invocation is to produce the Rvalue of a structure member (i.e. of a DOT expression), then it requests the Lvalue of the containing structure (i.e. of the subexpression `e1`). If the current invocation is to produce the Rvalue of a PLUS expression, then it requests the Rvalue of both operands. In the recursive calls, the `RorW` parameter has the value `READ` in all cases except for processing the left hand side of an assignment; in that case, `RorW` has the value `WRITE`.

Let us consider some specific examples and show how mungeTree and genSprigs work together to transform a syntax tree. Suppose we pass mungeTree the expression (p + n) where p is a nonpersistent dbint* and n is an int. (We must be requesting the Rvalue, since this expression does not denote an addressible object.) When we enter mungeTree (Figure 5.11), we will fall through both of the first two steps; since we are assuming that nothing here is persistent, the expression does not have a p-set, nor is it the root of a pinned item. In the main body of mungeTree (Figure 5.12) we then go to the PLUS branch in the switch statement, and will request the Rvalue of both operands. Consider the left operand, p. In the new invocation, we will fall to the NAME branch of the switch. Since context specifies Rvalue, we break out of the switch, and ultimately return from mungeTree with the same expression (p) that was passed in. The same path is taken in processing the right operand of the addition, n. Thus, after processing both operands, no changes have yet been made. In the next step of the PLUS branch, however, we will discover that the left operand is a DBREF, indicating that the program is performing arithmetic on a db pointer. At this point, the compiler declares a DBREF temporary variable, and builds the following expression:

```
*( _tmpRef001 = p, _tmpRef001.offset += n*4, & _tmpRef001 )
```

This expression assigns p to the temporary, increments the byte offset in the temporary by the n times the number of bytes in an integer, and finally produces the value of the temporary for use in a larger expression.[50]

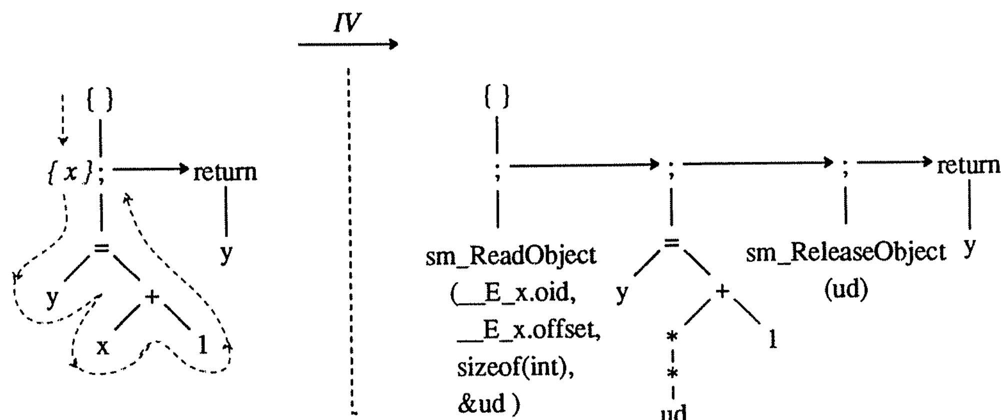
As a more interesting example, consider the first statement of the program in Figure 4.8. This example is reproduced (slightly modified) in Figure 5.13 for convenience. When (the statement version of) mungeTree is invoked on the first statement, it discovers that the node has a p-set containing the persistent dbint x. Just as in Figure 5.11, it then invokes genSprigs on the item x. GenSprigs then builds the code sprigs as described in section 5.2.2. In building the pinning sprig, genSprigs invokes mungeTree on x, requesting its Lvalue. This invocation of mungeTree eventually returns x's companion, __E_x. GenSprigs then uses __E_x to build the sm_ReadObject call, builds the other sprigs, and finally returns to the first invocation of mungeTree. This invocation then processes the assignment expression. We will not bother to trace the entire execution path here; the only interesting case occurs when mungeTree calls itself on the left operand of the addition. In this new invocation, context specifies Rvalue, the item x is currently pinned, and the RorW parameter has the value READ. Thus, as shown in Figure 5.11, mungeTree returns the reading sprig for x, which then replaces the occurrence of x in the tree. Eventually, we return to the invocation that called genSprigs. Its last action is to graft the pinning and unpinning sprigs onto the tree. Figure 5.13 shows the

[50] The last expression in parentheses takes the address of the temporary which is then used by the outer dereferencing operator. This is a detail required by C.

```
switch( ex->base )
{

case NAME:
      if( context == Lvalue && NAME's type is db )
            ex = companion;
      break;

case DOT:            /*  e1.member  */
      ex->e1 = mungeTree( ex->e1, Lvalue, READ );
      if( context == Lvalue && member's type is db )
      {
            /* ex->e1 is now a DBREF */
            using ex->e1 as a base, transform ex into
            a DBREF expr with offset incremented by
            member's offset within record;
      }
      break;

case DEREF:  /*  *e1  */
      ex->e1 = mungeTree( ex->e1, Rvalue, READ );
      if( context == Lvalue && ex->e1 is a DBREF )
            ex =  ex->e1;
      break;

case PLUS:   /*  (e1 + e2)  */
      ex->e1 = mungeTree( ex->e1, Rvalue, READ );
      ex->e2 = mungeTree( ex->e2, Rvalue, READ );
      if( ex->e1 is a DBREF )
      {
            using ex->e1 as a base, transform ex into
            a DBREF expr with offset incremented by
            ex->e2 time sizeof referenced type;
      }
      break;

case ASSIGN: /*  e1 = e2  */
      ex->e1 = mungeTree( ex->e1, Rvalue, WRITE );
      ex->e2 = mungeTree( ex->e2, Rvalue, READ );
      if( ex->e1 is a writing sprig for pinned item )
      {
            declare temporary and assign ex->e2 to it;
            use temporary to fill in data arg to
            sm_WriteObject call;
            replace ex by the write expr;
      }
      break;
}
```

The Procedure MungeTree (part 2)

Figure 5.12




A Simple Example Revisited

Figure 5.13

resulting tree; the tree has been simplified for presentation by omitting all nonessential code (i.e. code that is not essential for this discussion).

As a final example, consider the third statement in the linked list insertion procedure:

```
this->next->prev = that;
```

The p-set attached to this statement contains the items `(*this).next` and `*((*this).next).prev`. (See Figure 5.9.) Therefore, `mungeTree` first calls `genSprigs`, passing it the item `(*this).next`.[51] `GenSprigs` then calls `mungeTree`, asking for the Lvalue of this item. Now, tracing the nested calls to `mungeTree`, we ask for the Lvalue of `*this` and in turn for the Rvalue of `this`. The result of these calls is that the Lvalue of `(*this).next` is the OID from the DBREF `this` with an offset incremented by the offset within a list node of the field `next`. After building the sprigs, `genSprigs` returns to `mungeTree`.

`MungeTree` then invokes `genSprigs` on `*((*this).next).prev`. Essentially, we repeat the above process. `GenSprigs` calls `mungeTree` to obtain the DBREF of the item, causing `mungeTree` to begin a descent of the item's expression tree. This time, however, `mungeTree` will eventually call itself on the item `(*this).next`, asking for the Rvalue. At that point, it will discover that this item is pinned, and the reading sprig will be returned (without descending any further into the tree). This expression holds the value of `this->next`, which is the DBREF needed in order to pin `this->next->prev`. Figure 5.14 shows the (approximate) output of phase IV for the third statement of the linked list insertion procedure; the output for the other statements

[51]The order here is important given that the program is to follow a pointer chain. The fact that items are identified (and numbered) in depth first order ensures that we will pin the items in the chain in the proper order.

```
/*
** PIN( this->next )
** Notes:
**     "next" field has offset of 20
**     length param == sizeof(DBREF) == 16
*/
__tmpRef001 = this;
__tmpRef001.offset += 20;
sm_ReadObject
  (__tmpRef001.oid, __tmpRef001.offset, 16, &__tmpUd001);

/*
** PIN( this->next->prev )
** Reads this->next (a DBREF) out of buffer pool.
** Notes:
**     prev field has offset of 4
**     length param == sizeof(DBREF) == 16
*/
__tmpRef002 = *((DBREF*) *__tmpUd001);
__tmpRef002.offset += 4;
sm_ReadObject
  (__tmpRef002.oid, __tmpRef002.offset, 16, &__tmpUd002);

/*
** ASSIGN( this->next->prev ) = that
** Notes:
**     length param == sizeof(DBREF) == 16
**     new data's address == address of "that," a
**     local variable
*/
sm_WriteObject(__tmpUd002, 0, 16, &that);

/*
** UNPIN( this->next->prev )
** UNPIN( this->next )
*/
sm_ReleaseObject( __tmpUd002 );
sm_ReleaseObject( __tmpUd001 );
```

Translation of the Assignment: `(this->next->prev = that)`

Figure 5.14

in the procedure is similar. We note that the code in the figure has been formatted and annotated for presentation. As in Figure 5.13, we have omitted code that is not essential to the discussion.

# CHAPTER 6

# COMPILED ITEM FAULTING

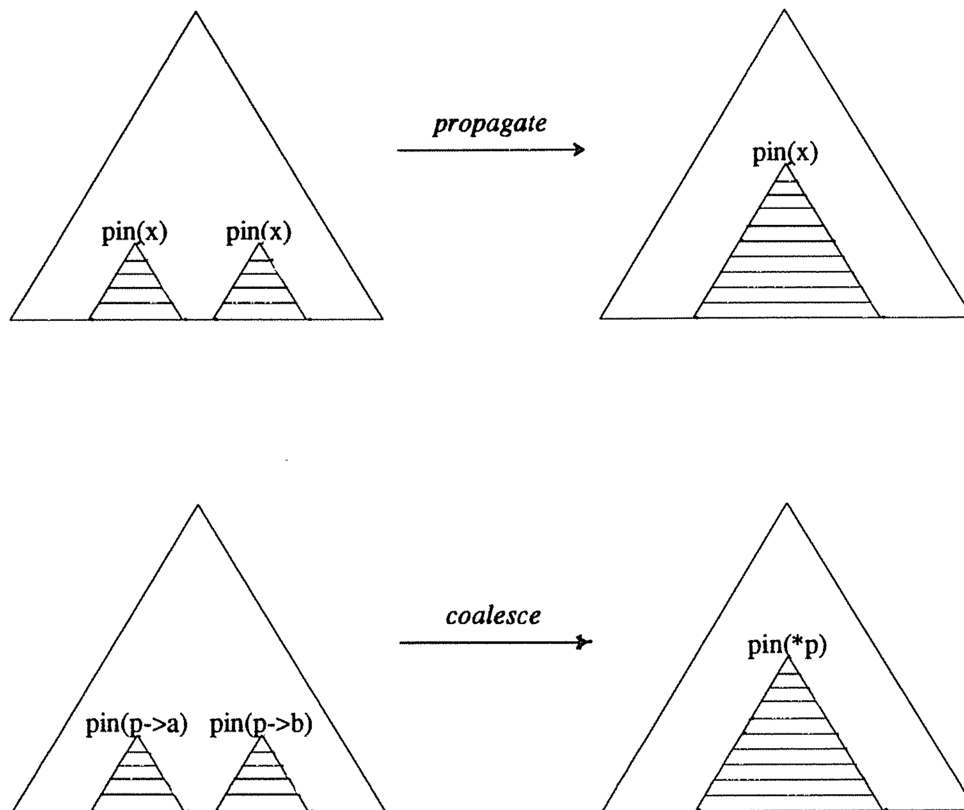
In the preceding chapter, we outlined the basic strategy for generating code to manipulate persistent objects. While the strategy does work, the quality of generated code often leaves much to be desired. Because the code generator uses only local information in deciding where to pin and unpin items, the number of such operations performed during execution can be excessive. For example, while multiple uses of an item within a given expression resolve to a single pin/unpin pair, multiple uses across statements are not recognized. Since pinning is a rather expensive operation (even when the requested data is already in the buffer pool), some kind of global optimization is needed in order for E programs to run with acceptable performance. In this chapter, we describe the approach to optimization taken in the current E compiler. The next section presents an overview of that approach, while following sections detail the implementation.

## 6.1. OVERVIEW

As we have said, pinning is the dominant cost in executing an E program. Therefore, the goal of optimization is to reduce the number of such operations performed at run-time. The E compiler achieves this reduction through two means: *propagation* and *coalescing*. Code generation as described in the previous chapter keeps an item pinned over very localized regions of the program. In addition, each item is treated independently and generates its own pin operations. By propagating information about item usage up the syntax tree, we may be able to keep an item pinned over many uses. Furthermore, when two or more items are part of the same object, we may be able to coalesce the separate pin operations into a single request for the spanning byte range. Figure 6.1 illustrates these ideas. Note that coalescing is also a form of propagation, since we are pinning the spanning byte range over a larger region which includes uses of the individual components.

### 6.1.1. Considerations

In any solution to a program optimization problem, one must consider the aspects of safety and profitability [Much81]. A safe solution is one that does not change the observed behavior (i.e. the output) of the program. In our case, there are two safety requirements. First, at the point in the program where a pin operation actually occurs, we must be sure that the path to the item being pinned is defined. If it is, then we say that the path to the item is *valid* at that point. For example, consider the following program fragment:




Propagating and Coalescing

Figure 6.1

```
if( p != NULL )
{
        p->a = 0;
        p->b = 0;
}
```

While we can legitimately pin `*p` before the first assignment, it would be an error to further propagate the pin out of (i.e. to the point immediately before) the **if** statement, since the path might be invalid at that point, i.e. because `p` is NULL. Note that our definition of "valid" is based on the programmer's intentions as expressed in the code. In the preceding code segment, for example, we consider the path to `*p` to be valid immediately before the first assignment, even though in some executions, `p` might have a (non-NULL) garbage value.

The second safety concern is related to the path validity problem. If an item is pinned over a region of the program, we must detect when the path to the item changes within that region due to an assignment or procedure call. If we did not and control were then to flow from the point of such a change to a use of the item, we would be accessing the wrong data. For example, in the following code segment, if the call to `f()` returns TRUE, then `p` is

reassigned:

```
p->a = 0;
if( f() )
        p = p->next;
p->b = 0;
```

If `*p` is pinned over this whole region, and we do not detect the change to `p`, then the last statement will (probably) assign to the wrong location. Note that, in the absence of information to the contrary, we must assume that such a change *invalidates* that path, since we do not know what value is being assigned.

With respect to profitability, the goal is to reduce the number of pins actually performed at run-time. While propagation seems promising in this regard, we must be careful. If we simply push pin operations up the syntax tree, we might actually *increase* the number of such operations performed by sometimes pinning items unnecessarily. That is, a pin operation might land at a point such that control could flow from that point without actually reaching a use of the item. In such a case, the program will waste time doing unnecessary work.

### 6.1.2. Compiled Item Faulting

These considerations led us to the following design for optimizing E code. The design is based on the observation that phase I already schedules pin operations at safe points in the program; we can avoid the path validity problem if we do not actually move the operations. In addition, by leaving the operations where phase I scheduled them, we also avoid the possibility of pinning an item that the program doesn't actually use. Finally, we can avoid pinning the same item many times if we simply check first to see if the item is already pinned; as we mentioned in Section 4.3.1, the cost of the check is negligible compared to the cost of a redundant pin.

With these observations in mind, let us outline our strategy for reducing the number of pin operations executed by an E program.

(1) A *pinning region for an item i*, $PR_i$, is defined to be a region of the program such that $i$ might possibly be pinned as long as control stays within the region and such that $i$ is definitely *not* pinned when control leaves the region. A pinning region is defined by a sequence, possibly of length 1, of nodes in the syntax tree.

(2) Within $PR_i$, each use of $i$ is preceded by a pin operation, i.e. pin operations are inserted into the code exactly as described in Chapter 5. However, each operation is now preceded by a boolean test; if $i$ is already pinned, then we skip over the (redundant) pin.

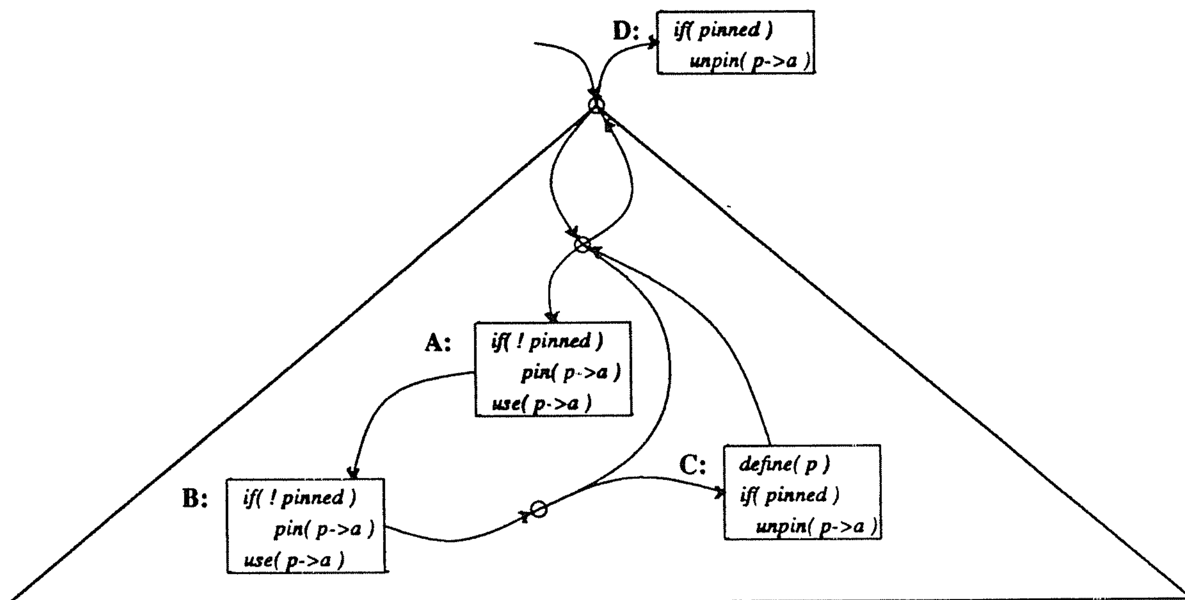
(3) Following each point in the region at which the path to $i$ is invalidated by an assignment or procedure call, we append an unpin. An unpin operation first checks that $i$ is currently pinned and then proceeds.

(4) An unpin operation immediately follows the exit point of the region.

(5) An unpin operation immediately precedes each flow-of-control construct (e.g. **break**) that transfers control to a point outside the pinning region.

Figure 6.2 illustrates the characteristics and control flow of the operations performed for an item within a pinning region. Assume that control enters a pinning region $PR_{p \to a}$ for the item `p->a`. When control first enters the region, `p->a` is not pinned. If control exits the region without reaching a use of the item, then no pinning operation will occur. When (and if) control reaches **A** for the first time, a pin operation occurs. The item then remains pinned as control flows to **B**. As long we remain in a loop between **A** and **B**, no further pinning occurs. If control flows from **B** to **C**, then `p->a` will be unpinned because **C** invalidates the path to the item by redefining `p`; if control flows back into **A**, then `p->a` will be pinned again based on the new value of `p`. If control flows from **B** to **D**, `p->a` will be unpinned because we are leaving the pinning region for the item. Finally, if control flows from **C** to **D**, no unpin is performed at **D** since that was already done by **C**.

We call this framework "compiled item faulting" (CIF). It is a faulting system because, like dynamic object faulting (DOF) (e.g. in PS-Algol), CIF places a run-time check before each use of an object, possibly resulting in a call to the storage layer. There are two important differences, however. First, CIF does not swizzle pointers; therefore, it does not pay the added overhead of unswizzling them when an object is released. Instead, CIF relies on




Control Flow in the Pinning Region $PR_{p \to a}$

Figure 6.2

compile-time analysis to reduce the number of pin operations performed at run-time. While the current implementation performs only simple analysis (as we shall see), the framework may be extended easily. Admittedly, static analysis will always be less accurate than dynamic interpretation, and as a result, there will always be applications in which DOF will execute fewer storage layer calls than CIF.[52] However, the range of applications for which this will be the case is unknown, as is the overall performance impact of pointer unswizzling on DOF.

The second difference between CIF and DOF is that DOF appears to be constrained to read whole objects; if a pointer is swizzled, then the entire object must be memory resident. CIF, by contrast, is able to read only the portion of the object needed in the given context. If an application manipulates small portions of large objects, this may be an important consideration.

## 6.2. IMPLEMENTATION

Having explained the approach to optimization in fairly high level terms, we now describe how it has been implemented in the E compiler. In the course of this description, we will also revisit phases I and IV. Chapter 5 glossed over certain aspects of their implementation in order to convey the essential ideas; this chapter will revise those parts to fit them into the overall optimization strategy.

### 6.2.1. Phase I Revisited

In Chapter 5, we said that phase I identifies the set of items that are to be pinned within a localized region of the program (essentially, within an expression). It then attaches this set, called the p-set, to the node in the syntax tree that defines the region. We then said that phase IV later grafts pin and unpin operations onto the tree based on these p-sets.

When phase I determines that a set of items should be pinned over a region, it actually attaches *three* identical[53] sets to the node defining that region: an *r-set*, an *x-set*, and a *p-set*. (The *i-set* will be explained in the next section.) The r-set (for "region" set) is a set of items such that the node marks the beginning of a pinning region for each item in the set. Similarly, the x-set (for "exit" set) contains items such that the node marks the end of a pinning region for each item in the set. When a single node in the syntax tree (e.g. one statement) defines the entire pinning region for a given item, that item appears in both the r-set and the x-set for that node; when the item's pinning region spans a sequence of nodes, then the item appears in the r-set of the first node of the sequence

---

[52] Note that even DOF systems will fault multiple times on the same object if the references are made through different pointers. DOF wins in this regard only if multiple references are made to the same object through the *same* pointer.

[53] The optimization phases II and III may later alter the r-set and x-sets, however. Since we were not considering optimization in Chapter 5, we did not bother to distinguish between them there.

| Set | Meaning |
|---|---|
| r-set | The node marks the beginning of a pinning region for each item in the set. |
| x-set | The node marks the end of a pinning region for each item in the set. |
| p-set | We must graft a pinning operation before the node for each item in the set. |
| i-set | At the point immediately following the node, the path to each item in the set is invalid. |

Summary of Item Sets at a Program Node

Table 6.1

and in the x-set of the last node of the sequence. Finally, the p-set (as in Chapter 5) contains items for which pin operations (i.e. pinning sprigs) are needed at that point in the program. Table 6.1 summarizes these sets. We do not show the sets produced by phase I for the insertion example, as all three are identical and equal to the p-sets shown in Figure 5.12.

R-sets and x-sets are the subjects of propagation and coalescing; that is, we may enlarge the pinning region for a given item by moving it into the appropriate r- and x-sets further up in the syntax tree. The p-sets, however, do not change after phase I creates them. Phase IV grafts pinning sprigs onto the tree based on the contents of the p-sets, and it grafts unpinning sprigs based (in part) on the contents of the x-sets. That is, phase IV adds a pin operation for each item in a p-set immediately before the node associated with that set, and it adds an unpin operation for each item in an x-set immediately following the node associated with that set.

### 6.2.2. Phase II: Ensuring Path Safety

In the overview of optimization, we said that if the path to an item changes (becomes invalid) within a pinning region $PR_i$, then we must unpin $i$ following the change. It is the responsibility of phase II to compute, for each assignment and function call, the set of items whose *path* could be invalidated. This set, called a i-set (for "invalidates path"), is then attached to the appropriate node in the syntax tree. In addition to its other duties, phase IV will look for i-sets; for a given item in the set, if the invalidation occurs within a pinning range for the item, phase IV will graft an unpin operation immediately following the node associated with the set.

Note that i-sets are related to the def-sets often computed in dataflow analysis [ASU86]. In that setting, we are interested in knowing which items could be written into (defined) by an assignment or a procedure call. Here, we will use that information to derive the set of items whose path is invalidated. The basic idea, as suggested in Figure 6.2, is that changing the value of a pointer invalidates the path to any item that involves a dereference

through that pointer. If the change occurs within a pinning region for such an item, then we must unpin that item following the assignment.

#### 6.2.2.1. The Role of Alias Analysis

In a language with pointers, an assignment may well define more than just what appears syntactically on the left hand side of the assignment. If the left operand of the assignment involves a pointer dereference, then, lacking any other information, we must assume that the assignment defines all items[54] [ASU86]. Even if the language does not have pointers, but does have reference parameters, then we must still be concerned with what might be defined in a procedure call [RicS89]. E has both pointers and reference parameters; furthermore, since E is a derivative of C, pointer usage can be quite unrestricted.

In order to improve our estimate of the side effects of an assignment, we can solve a dataflow problem known as alias analysis. Alias analysis techniques have been under development for quite some time, e.g. [Alle74, Weih80]. Recent work by Larus [Laru88] and Pfeiffer [Pfei89] promises to extend these techniques to languages having structures and heap allocation. An interesting avenue of future research, therefore, would be an investigation of the applicability of these results in the context of E. However, given that this thesis is not about alias analysis, we have taken a very simple approach for now (in the interest of implementing the compiler in a reasonable time frame). Fortunately, this approach proves effective in a number of realistic situations. This phase of the compiler is implemented as a "black box" having a simple, well-defined interface, so it should not prove difficult in the future to integrate more sophisticated alias analysis techniques into the existing framework.

#### 6.2.2.2. The Current Implementation

As we have explained, we are interested in computing the set of all items whose path might become invalid as a result of an assignment or procedure call. To do this, we must first compute the set of items defined at such points. The rules we are about to give form a very simple heuristic for computing this set.

As explained in Chapter 5, every item is associated with a particular scope. We define an item to be ***reachable through a pointer*** (or simply, ***reachable***) everywhere in its scope if any of the following are true:

(1) the item itself involves a pointer dereference. For example, the items `*p`, `p->a`, and `p->next->b` are all reachable through a pointer since they all involve at least one pointer dereference.

(2) the item is a named variable from the global scope.[55] Since we are not performing 'real' alias analysis

[54] That is, only one item is actually defined, but we must assume conservatively that it could be *any* one.

[55] Even though global variables are "imported" to the function body's scope during CSE identification (Section 5.1), they are still known to be global variables.

(either inter- or intra-procedurally), we assume that the address of any global variable could have been assigned to a pointer.

(3) the item is a named variable in the local scope, and its address is taken somewhere in the scope. When we enter the scope in which a name is declared, there are no aliases for that name. An alias can be created only if the address-of operator (`&`) is applied to the name or if the variable is passed by reference. If it is, then we assume, in lieu of dataflow analysis, that the name is reachable through a pointer everywhere in the scope.

Given this definition, we can compute the set of items potentially defined by an assignment or a function call as follows:

(1) An assignment in which the left operand is a named variable defines only that name.

(2) An assignment in which the left operand involves a pointer dereference defines every item that is reachable through a pointer.

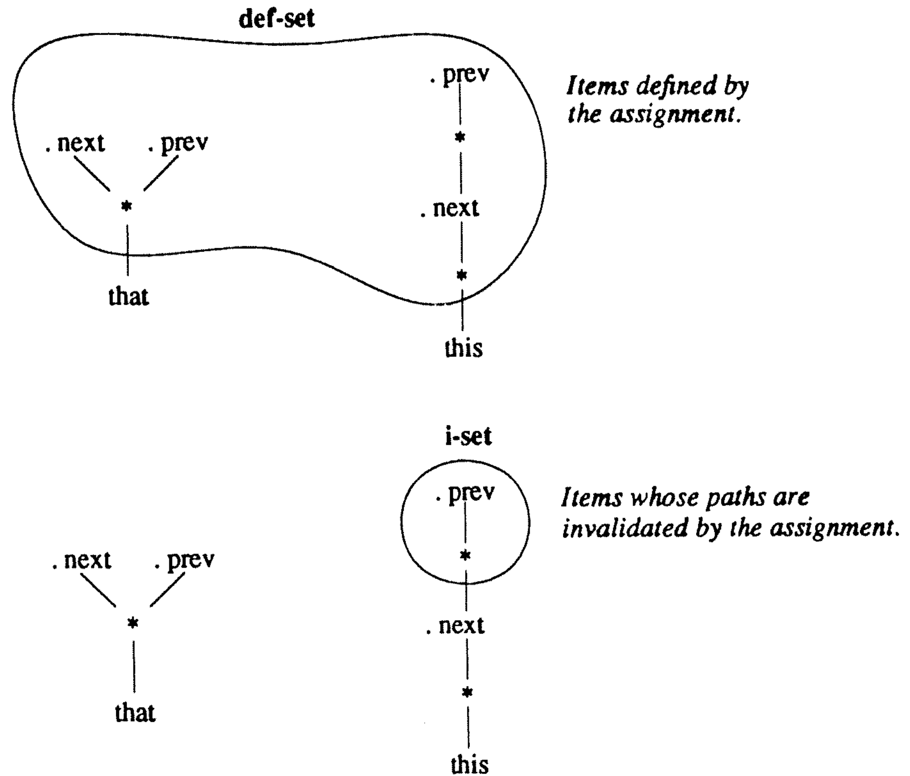
(3) A function call defines every item that is reachable through a pointer.

Based on these (very conservative) rules, computing the set of items defined by an assignment or a function call (the def-set) is extremely simple. From this set, we may then derive the set of items whose path is invalidated (the i-set) essentially by noting the pointers in the def-set; we must now consider the path to any item involving a dereference through such a pointer to be invalid. The algorithm for determining i-sets uses the dag representation of the syntax tree. For each item that is defined by an assignment or function call, we traverse upward through its parent links. In so doing, we locate every expression for which the given item is a subexpression; since the value of the item has changed, so has every expression that includes it. Along *each* path in this upward traversal, we note the first item that is a pointer dereference, i.e. the first item whose expression operator is a `*`. This item, and every item above it on the path, is added to the i-set for the defining assignment or function call.

Let us return now to the linked list insertion example for an illustration of this process. The first statement of the procedure is:

```
that->next = this->next;
```

Figure 6.3 summarizes the computations performed by phase II due this statement. Since the procedure does not take the address of either `this` or `that`, neither variable is considered reachable through a pointer. All other items in the procedure are considered reachable, however, because they all involve pointer dereferences. Since the left hand side of the assignment also involves a pointer dereference, than all reachable items are added to the def-set. From the def-set, we derive the i-set shown at the bottom of the figure. Consider, for example, the item `*this`. Tracing upward, we encounter the items `(*this).next`, `*((*this).next)`, and `(*((*this).next)).prev`. Since the second item on this path is a value produced through a pointer traversal, we add it and the third item to the i-set. Now consider the item `*that`. Tracing upward, we encounter




Computing the i-set for the Assignment: `that->next = this->next;`

Figure 6.3

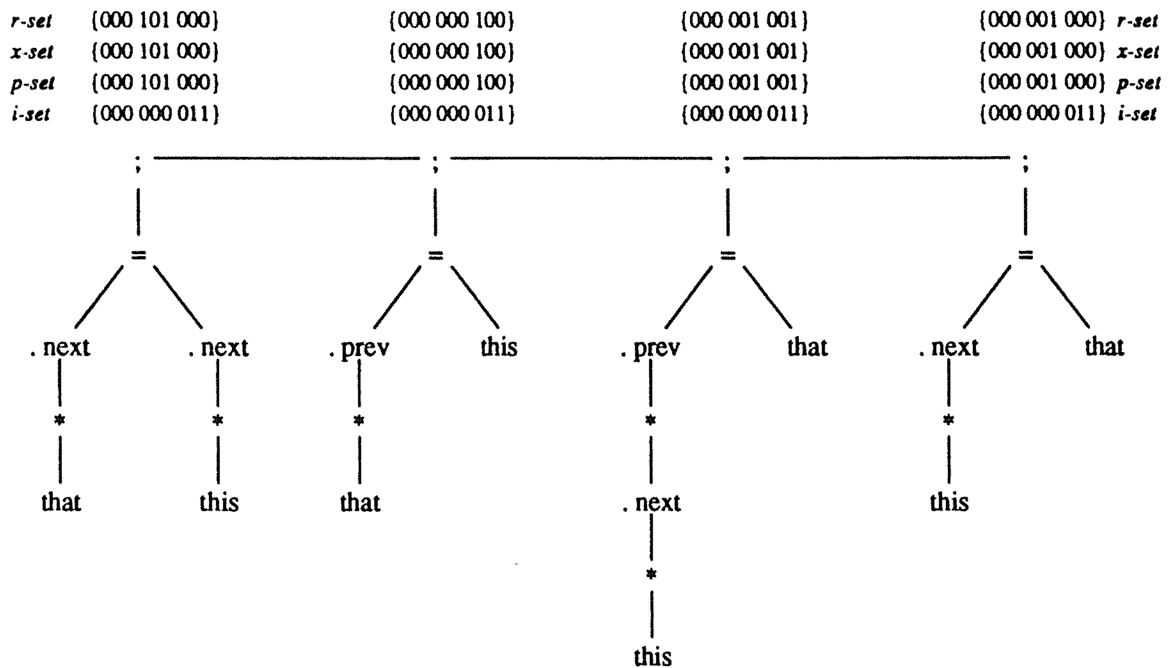
only the items `(*that).next` and `(*that).prev`. Since the definition of `*that` does not imply that the location of either item has changed, neither is added to the i-set.[56]

As this example illustrates, the usual result of our naive heuristic is that items involving a single pointer dereference ("one-hop" items) are not added to the i-set resulting from an assignment through a pointer, while items involving two or more hops will always be added. One-hop items will be added only if the root pointer of the item (a named variable) is considered reachable as defined above. Assuming that the root pointer is not reachable, then the compiler will be quite effective in keeping the one-hop items pinned over large regions of the program. Fortunately, we expect that such cases will be quite common. Consider a class member function, for example. The compiler translates references to class data members into a dereference through the pointer `this`. Thus, we expect items of the form `this->member` to be very common. If the function does not contain the expression `&this`,

[56] That is, assigning to a structure does not imply that the locations of any of its fields have changed.

then the object referenced by `this` can safely remain pinned over large regions of the function. In any case, we do not mean to suggest that true alias analysis is not needed, but rather that our simple heuristic is a reasonable first approximation and that it may be surprisingly successful in a number of common cases.

After phase II computes a given i-set, it attaches the set to the associated node in the syntax tree. The reasoning used in Chapter 5 to determine where phase I attaches p-sets also applies to phase II in determining where to attach i-sets. Since flow of control through most expressions is not defined, E chooses to attach the i-sets for a node as high as possible in the tree, i.e. at the first ancestor node in the tree for which flow of control is defined. The output after phases I and II have processed the list insertion procedure is shown in Figure 6.4.




*Legend of Items*

| | | | | | |
|---|---|---|---|---|---|
| 0 | this | 3 | (*that).next | 6 | (*that).prev |
| 1 | that | 4 | *this | 7 | *((*this).next) |
| 2 | *that | 5 | (*this).next | 8 | *((*this).next).prev |

Result of Phase II for the List Insertion Procedure

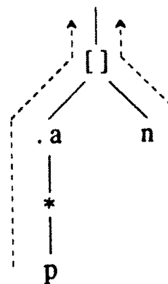
Figure 6.4

#### 6.2.2.3. Handling Array Elements

If an item denotes an element in an array, then the location of the item changes if either the path to the array itself changes or if the value of the array index changes. For example, consider Figure 6.5. The item of interest, `p->a[n]`, is an array element in a structure referenced by `p`. If `p` is defined, then the path to the array has changed, and therefore, so has the path to the array element. If `n` is defined, then the location of the array element denoted by the item also changes. With a very slight extension, the algorithm given above will handle both cases. The extension simply affects the upward traversal through the item's parent links. In particular, if we encounter an item whose expression operator is `[]`, and if we reach this item via its right child (i.e. via the index of the array dereference), then we add this item and any items further up to the i-set being computed.

### 6.2.3. Phase III: Propagation and Coalescing

Phase III reduces the number of pin and unpin operations performed at run-time by propagating the pinning regions of items higher up in the syntax tree and by coalescing the pinning regions of (certain) different items into a single pinning region. Since it is guaranteed that an item will no longer be pinned when control leaves its pinning region, the compiler attempts to make such regions as large as possible. In fact, it is easy to see intuitively that if we simply define the pinning region for every item to be the entire scope in which the item is defined, then the program will perform as few pin operations as static analysis will allow. To see this, imagine that control has reached a use of an item, and therefore, the item is pinned. If control then flows to another use, and in so doing, leaves the current pinning region, then the program will definitely execute another pin operation at the second use. However, if control stays within the same pinning region (i.e. if the region is large enough to contain both uses), then no additional pin is required. If control flows through an intervening point which invalidates the path to the item, then the second pin will be performed in either case.




Detecting Path Changes for Array Elements

Figure 6.5

However, the compiler does not take quite so simplistic an approach. Although the Storage Manager's buffer pool is large, it is still finite, and it is unnecessarily wasteful of space to keep an item pinned when it is no longer needed. (We shall have more to say about the impact of finite buffer space in Section 6.3.) Accordingly, phase III propagates the pinning region of an item to be the smallest region of the program that includes all uses of the item. This approach still results in the "minimum" number of pin operations, but it reduces the program's buffer space requirements.

Of course, approaches other than simply minimizing the number of pin operations are also possible. For example, one could try minimizing the number of pin operations subject to constraints on the total amount of data pinned at any one time. Or, one might try minimizing the estimated space-time product of the program. However, such alternatives appear to be much more complex than the current approach, and it is not at all clear how successful they can be, given the information available at compile time. In order to estimate the amount of buffer space consumed at any point in the program, the compiler would have to know, for example, how objects are distributed on disk pages, and whether two pointers reference the same or different objects. The former is information explicitly hidden by the Storage Manager interface, and the latter is information that the compiler can only approximate, e.g. two pointers might refer to the same object in one execution but to different objects in another.

#### 6.2.3.1. Coalescing

Before proceeding to describe the propagation algorithm, let us first examine in more detail what we mean by coalescing. If two items refer to separate locations within the same Storage Manager object, then we can consider coalescing the pinning regions for the two items into a single pinning region for an item whose byte range spans (at least) the original two. For example, pinning regions for the items `p->a.x` and `p->a.y` can be coalesced into a pinning region for `p->a`; we can further coalesce this region with one for `p->b` into a pinning region for `*p`. However, we cannot coalesce pinning regions for `p->next->a` and `p->b`, nor can we coalesce `p->a` with `q->b`; in both cases, we must assume that the items refer to locations in separate objects.[57]

Coalescing can reduce the number of pin operations at the expense of pinning a larger range of bytes. Given the design of the Storage Manager, however, these larger byte ranges may not increase the actual buffer space requirements of the program at all. Pinning a single byte still requires at least an entire page frame in the buffer pool; pinning two bytes can consume up to four page frames if those bytes happen to be split across two leaf blocks of a large object (see [Care86a]). The compiler therefore assumes that it can coalesce freely unless the resulting

[57]With sophisticated alias analysis, we might be able to improve on this estimate.

item's size exceeds a certain threshold. Currently, this threshold defaults to 8K bytes; the user may reset this limit with the command line option "+Tn" where n is the new limit.[58]

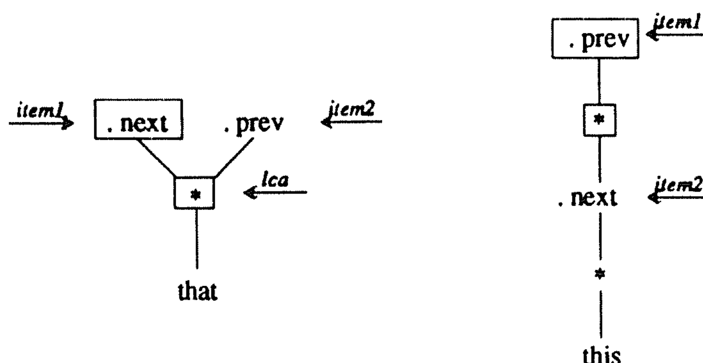
One other limitation on coalescing is that it is based solely on the type composition of the item. For example, in coalescing the pinning regions of two fields of a structure, we only consider pinning the entire structure as the next level of granularity; we do not consider pinning the subrange of bytes that just spans the two fields. The main reason for this limitation is that the alternative would greatly increase the complexity of the implementation, for such subranges do not correspond to any single item (i.e. to any Lvalue). As a result, of course, if the structure contains additional fields such that the total size exceeds the coalescing threshold, then the compiler rejects the coalesce. However, the compiler will still propagate the two pinning regions individually.

The algorithm that the compiler uses for deciding if it can coalesce two pinning regions is quite simple. Suppose that $item_1$ and $item_2$ are the items associated with the regions. We traverse downward through $item_1$, marking each node visited and stopping either at a named variable (since there are no further subexpressions) or at the first node which is a pointer dereference (since the next item down will be located in a different object). In so doing, we mark every item denoting a range of bytes that contains the range denoted by $item_1$. We then traverse downward through $item_2$. If we encounter the root or a pointer dereference without finding a marked node, then $item_1$ and $item_2$ denote locations in different objects, and their pinning regions cannot be coalesced. If we encounter a marked node, then the regions can indeed be coalesced, provided that the size of the item denoted by this node is below the threshold. Given the similarity between this algorithm and finding the least common ancestor in a tree, the item is called the *lca* of $item_1$ and $item_2$. Note that the algorithm works for $item_1 == item_2$, i.e. an item is its own *lca*. Figure 6.6 illustrates the outcome of this algorithm for two pairs of items from the linked list example. Boxes enclose the nodes marked in the first pass (i.e. over $item_1$). In the first example, `(*that)` is the lca of `(*that).next` and `(*that).prev`. In the second example, the items `(*(*this).next).prev` and `(*this).next` have no lca.

We note that the lca of two array elements, e.g. `a[n]` and `a[m]`, is the entire array, `a`. The compiler currently attempts to coalesce array references, generating code that pins the whole array when individual elements are accessed. While this is a perfectly reasonable approach for small arrays, it is not appropriate for very large arrays. Thus, as with all coalescing, the size of the array cannot exceed the threshold. One potential area for future work is in improving the array processing characteristics of E programs.

---

[58] We will probably want to provide a pragma for setting this value so that the threshold can be readjusted within a single compilation unit.



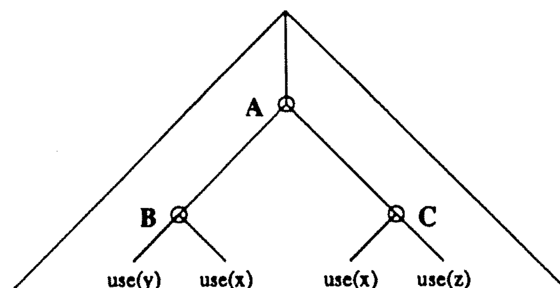

Two Examples of Finding the LCA

Figure 6.6

#### 6.2.3.2. Propagation

The basic operation of the propagation algorithm is to walk the syntax tree, adjusting r-sets and x-sets at each node based on the items used within the program region defined by the node. The goal, as stated earlier, is to propagate the pinning region of each item to be the smallest region of the program that covers all uses of the item. The task is complicated by three issues. First, we cannot determine what to propagate up to a given node simply by looking at what has been propagated to the node's immediate children. This is easily seen in Figure 6.7. The children of node **B** in the figure use the items `x` and `y`; although not shown, phase I has inserted pinning regions around each use. Now, looking only at node **B** and its children, neither `x`'s nor `y`'s pinning region will be propagated to node **B** since doing so would not cover any additional uses. Similar comments apply to node **C**. Thus, when we reach node **A**, the r- and x-sets for nodes **B** and **C** are all empty. However, we intend for the pinning region of `x` to propagate to node **A** in order to cover both uses. Accordingly, the propagation algorithm also builds, for each subregion of the program, sets containing the ***union*** of all items used anywhere within that subregion. These sets are called ***mention sets***, or m-sets, because they keep track of all items mentioned anywhere within the region. At a given node, propagation decisions are then based on these m-sets. A new m-set is formed (by taking the union of the m-sets from the subregions), propagation decisions are made for this node, and the node's m-set is then passed up to the parent. Unlike the sets described in Table 6.1, m-sets are only temporary; phase III discards the m-set for a node once it has been used in making a propagation decision.

The second complication for propagation is that, despite the example in Figure 6.7, we cannot in general determine what to propagate to a given node simply by taking the intersection of the m-sets from the subregions. Rather, the possibility of coalescing must be considered as well. The intersection of an m-set containing `p->a` with one containing `p->b` is the empty set; however, we know from Section 6.2.3.1 that we can coalesce these




Propagation Based on All Uses

Figure 6.7

item into the common subsuming item `*p`. Figure 6.8 shows the algorithm for determining what to propagate, given two m-sets, `S1` and `S2`. For a given pair of items, we check if the two have an lca, as described previously. If `i1` and `i2` do coalesce and the resulting `lca` is not too large, then `i1` and `i2` are removed from the result set (in case either was added in a previous iteration), and their `lca` is added. Note that the order of these two operations is important since the `lca` might actually be equal to `i1` or `i2`. The `lca` of `p->a` and `*p`, for example, is `*p`. After adding `lca` to the result, we remove `i2` from `S2`. The reason is that once the `lca` becomes part of the result set, we do not need to consider `i2` again on any future iteration. Finally, the `lca` replaces `i1` for the remaining iterations of the inner loop.

```
R = { };   /* the result set */

foreach( item i1 in S1 )
    foreach( item i2 in S2 )
      if( lca exists for i1 and i2
      &&  lca's size <= coalesce threshold )
      {
          R = R - { i1, i2 };
          R = R + { lca };

          S2 = S2 - { i2 };
          i1 = lca;
      }
```

Propagation Algorithm for a Node

Figure 6.8

Let us consider an example. Suppose that we start with a node where `S1 = { x, y, p->a }` and `S2 = { x, z, p->b }`. Table 6.2 summarizes the state of the important variables in the algorithm at the start of each iteration of the inner loop and gives the final state when the outer loop exits. (Note that by removing `x` from `S2`, we have saved two iterations.)

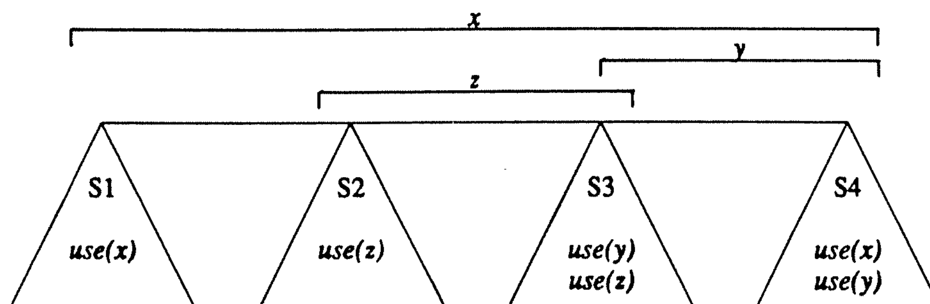
The algorithm in Figure 6.8 can determine which pinning regions should be coalesced and propagated when the destination is a single node in the syntax tree. For example, this algorithm can determine which pinning regions should be propagated to **if** node from the constituent **then** and **else** statements. When we consider a sequence of statements, however, the situation is somewhat more complex for two reasons. First, the pinning range for a given item may begin with one statement in the sequence and end with a later statement; the algorithm in Figure 6.8 implicitly assumes that the pinning region corresponds to a single node. Second, the pinning regions for different items in the sequence may overlap in arbitrary ways; the algorithm assumes that all the propagated pinning regions coincide at the same node. Statement sequences, then, present the third and final complication that we must consider in the propagation phase. Figure 6.9 illustrates this. As in the algorithm in Figure 6.8, we may still be able to coalesce the pinning ranges of distinct items. For example, if statement S1 contains a use of `p->a` and statement S3 contains a use of `p->b`, then we can give `*p` a pinning range from S1 through S3.

Figure 6.10 shows the algorithm for propagating and coalescing pinning ranges in a statement sequence. We assume that an m-set is available for each statement in the sequence. The main data structure for the algorithm is `range`, an array of integer pairs that determines the first and last statements of the pinning region for each item visible in the current scope. This array is first initialized to all 0's, meaning that no items have yet been seen. The nested loop then processes each item in each m-set in the statement sequence. An item, `i1`, is compared against

| iter | i1 | i2 | S2 | R |
|---|---|---|---|---|
| 1 | x | x | { x, z, p->b } | { } |
| 2 | x | z | { z, p->b } | { x } |
| 3 | x | p->b | { z, p->b } | { x } |
| 4 | y | z | { z, p->b } | { x } |
| 5 | y | p->b | { z, p->b } | { x } |
| 6 | p->a | z | { z, p->b } | { x } |
| 7 | p->a | p->b | { z, p->b } | { x } |
| final | - | - | { z } | { x, *p } |

Example of Algorithm in Figure 6.8

Table 6.2




Propagating Over Sequences

Figure 6.9

every item, `i2`, seen so far, i.e. against every item with a nonzero entry in the `range` array. If `i1` and `i2` have an `lca` and the `lca`'s size is not too large, then the `lca` is given a pinning range that extends from the beginning of `i2`'s range through the current statement. Furthermore, if the `lca` is not equal to `i1`, then `i1`'s pinning range is cleared; `i2` is handled similarly. (This step parallels the removal of `i1` and `i2` from the result set in Figure 6.8.) If `i1` had no `lca` with any item already seen, then `i1`'s pinning range is initialized to begin and end with the current statement. When the algorithm terminates, the range array specifies the pinning range for every item in the statement sequence (or else indicates that the item is not used).

The algorithms in Figures 6.8 and 6.10 show the essential processing steps at various levels of the syntax tree. They do not, however, show the details of walking the tree, applying the algorithms, adding and removing items from the r- and x-sets, or generating m-sets. Such details, while important, are not particularly interesting, and we will not show them. We do note, however, that the syntax tree is processed bottom-up and that the levels of the tree essentially alternate between statement hierarchies and statement sequences. For example, an **if** statement has **then** and **else** components, each of which may be a statement sequence. Thus, the algorithms in Figures 6.8 and 6.10 will be applied alternately as we percolate up the tree.

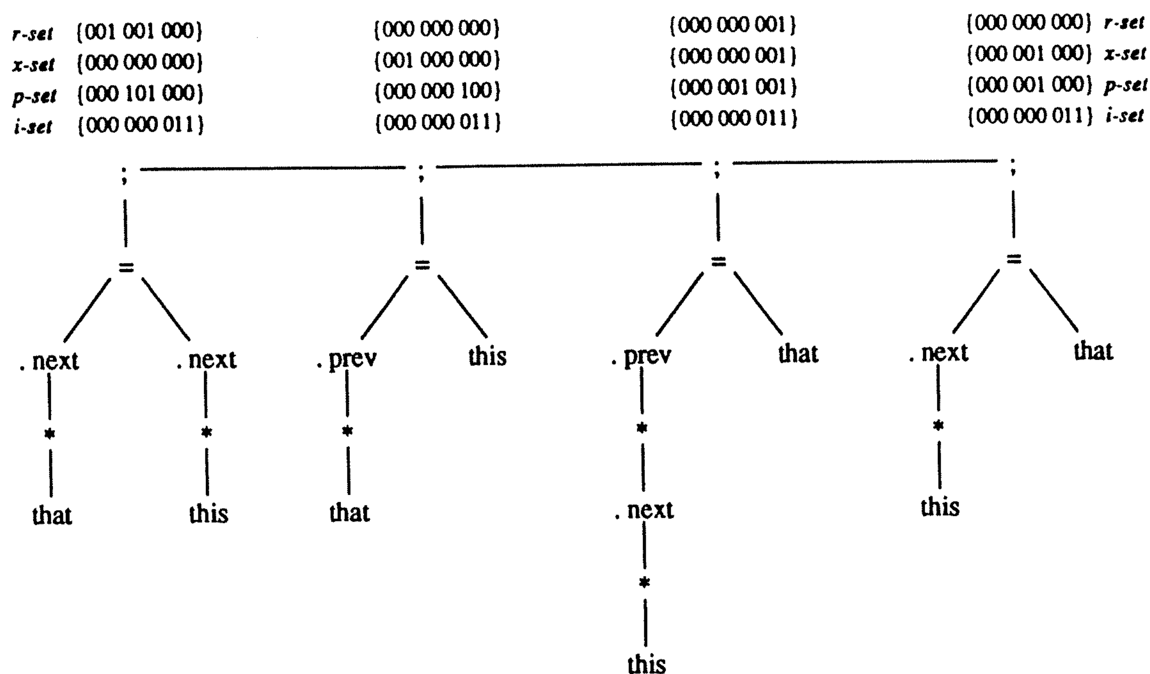
Returning to the linked list insertion procedure, Figure 6.11 shows the result of phase III as applied to this example. Note the changes to the r- and x-sets from Figure 6.4.[59] The interpretation of the sets in Figure 6.11 is now as follows: The pinning region for the item `(*that)` spans the first two statements as it appears in the r-set of the first statement and in the x-set of the second. After the second statement, the object referenced by `that` will be unpinned. Note that the original pinning regions for `(*that).next` and `(*that).prev` have been

[59] The reader may refer back to Table 6.1 for a summary description of these sets.

```
// Let n_items be the number of items defined in the current scope.
// Let range be an array of n_items integer pairs, first and last.
// If i1 is the global item id for a given item, then range[i1] gives
// the algorithm's current idea of the pinning region for the item.

foreach( int i1 = 0; i1 < n_items; i1++ )
{
      range[i1].first = 0;
      range[i1].last  = 0;
}

foreach( statement s in the sequence )
{
      foreach( item i1 in s's m-set )
      {
            foreach( entry in range )
            {
                  if( entry contains 0's )
                        continue;  /* item not seen */

                  let i2 be the item associated with this entry;

                  if( lca exists for i1 and i2
                  &&  lca's size <= coalesce threshold)
                  {
                        range[ lca ].first = range[ i2 ].first;
                        range[ lca ].last  = s;

                        if( lca ≠ i1 )
                        {
                              range[ i1 ].first = 0;
                              range[ i1 ].last  = 0;
                        }

                        if( lca ≠ i2 )
                        {
                              range[ i2 ].first = 0;
                              range[ i2 ].last  = 0;
                        }
                  }
            } /* end of i2 loop */

            if( i1 had no lca for any i2 )
            {
                  range[ i1 ].first = s;
                  range[ i2 ].last  = s;
            }
      } /* end of i1 loop */
} /* end of s loop */
```

Propagation Algorithm for a Statement Sequence

Figure 6.10




*Legend*

| | | | | | |
|---|---|---|---|---|---|
| 0 | this | 3 | (*that).next | 6 | (*that).prev |
| 1 | that | 4 | *this | 7 | *((*this).next) |
| 2 | *that | 5 | (*this).next | 8 | *((*this).next).prev |

Insertion Procedure After Phase III

Figure 6.11

coalesced, since those items no longer appear in any r- or x-set. The pinning region for `(*this).next` spans all four statements. Finally, the item `(*(*this).next)).prev` has a pinning region that spans only the third statement, as only that statement references the item.

### 6.2.4. Phase IV Revisited

Let us now finally return to phase IV and reexamine the process of transforming the syntax tree. The algorithms `mungeTree` and `genSprigs` are still essentially valid as described in Chapter 5. However, the specifics of the actions taken at various points in the traversal need to be modified in order to fit into the framework developed in this chapter.

#### 6.2.4.1. Entering a Pinning Region

During its traversal of the tree, `mungeTree` looks for r- and x-sets attached to the various nodes. When an r-set indicates the start of a pinning region for an item, `mungeTree` calls `genSprigs` to generate code sprigs for the item. As explained in the last chapter, this includes code to pin, unpin, and reference the range of bytes denoted by the item. The reading and writing sprigs are unchanged from the descriptions given in Chapter 5. As discussed at the beginning of this chapter, however, the pinning and unpinning sprigs are now preceded by a boolean check to see if the item is currently pinned. Instead of allocating yet another variable to record this information, we simply utilize the user descriptor pointer. Being sure that this variable is initialized to NULL upon procedure entry, we amend the form of the pinning and unpinning sprigs as follows:

$pin_3$ `::=` `( _tmpUd001 == NULL ?` $pin_2$ `: 0 )`

$unpin_1$ `::=` `( _tmpUd001 != NULL ? (` $unpin_0$`, _tmpUd001 = NULL ) : 0 )`

Here, $pin_2$ and $unpin_0$ denote the pin and unpin expressions, respectively, as given in the last chapter. Thus, if the user descriptor pointer is NULL coming into a pin operation, the pin will be performed. In the process, the pointer will acquire a non-zero value (regardless of whether the object is persistent or volatile), and successive pin operations will then be skipped until after the next unpin. Similarly, an unpin operation first checks the pointer; if it is non-NULL, then we perform the unpin and clear the pointer.

When a pinning region results from coalescing, the byte range of the item actually pinned (i.e. the lca) contains the byte ranges of the items that were coalesced. In such cases, we say that the lca is pinned *explicitly*, and the items that were coalesced are pinned *implicitly*. When phase IV enters a pinning region for an item, the routine `genSprigs` must generate sprigs for all implicitly pinned items as well as for the explicitly pinned item. In particular, it will generate reading sprigs that include appropriate offsets from the start of the pinned (lca) byte range. For example, in the insertion procedure, the pinning regions for `(*that).next` and `(*that).prev` were coalesced. When phase IV enters the pinning region for `(*that)`, it generates code sprigs to pin, unpin, and read the byte range denoted by `(*that)`. Reading sprigs are also generated for the items `(*that).next` and `(*that).prev`; these code fragments include appropriate offsets within the pinned byte range.

#### 6.2.4.2. Grafting Pin and Unpin Operations

When `mungeTree` encounters a node with an associated p-set, it grafts a pinning sprig before the node for each item in the p-set. As explained earlier, phase I determines the location of these nodes and the contents of their p-sets. However, the parameters of the pin operation are dictated by the outcome of phase III. If the item was coalesced, then the operation actually grafted onto the tree is the pinning sprig of the (explicitly pinned) lca.

Phase IV grafts an unpinning sprig onto the syntax tree wherever control leaves a pinning region for an item. For example, if a node has an x-set, then `mungeTree` will insert an unpinning sprig for each item in the x-set. In

addition, **break**, **continue**, and **goto** statements may transfer control out of the middle of one or more pinning regions; for each such region, `mungeTree` prepends an unpinning sprig to the statement. While it is relatively simple to determine exactly which regions will be exited on a **break** or **continue**, **goto**'s are somewhat more difficult. The current implementation simply unpins all items before a **goto**.[60] Finally, as explained earlier in this chapter, if an item is currently pinned and the path to the item changes as a result of an assignment or function call, then the item must again be unpinned. Thus, `mungeTree` also looks for i-sets produced by phase II; for each item in the i-set, if we are currently in a pinning region for the item, then `mungeTree` grafts an unpinning sprig onto the tree following the node.

## 6.3. COPING WITH FINITE BUFFER SPACE

One final but important point needs to be addressed. In the description of Compiled Item Faulting thus far, we have implicitly assumed that the amount of available buffer space is sufficient to handle all pin requests. Of course, the buffer pool is finite, and it is possible for a pin request to fail for lack of space. This section describes a mechanism that has been implemented for handling such exceptional conditions.

The basic idea behind the mechanism is quite simple. First, we expect that the majority of pin requests will succeed; if a request fails, we are willing to run a relatively expensive algorithm in order to continue the program. This continuation algorithm will descend the run-time stack, unpinning items bottom-up, i.e. beginning with the deepest stack frame. After each unpin, the failed pin request is then retried. If it succeeds, then the program continues normally. If, however, the top stack frame is reached without success, the transaction is aborted. In this event, the user can attempt to rerun the program with a larger buffer group.[61]

In order for this mechanism to work, the exception handler must be able to find all of the user descriptors in all existing stack frames. Furthermore, among the pinned items, it must be able to distinguish the persistent ones from the nonpersistent ones; unpinning a nonpersistent item will not help to reclaim buffer space. Thus, we make one last revision to our description of the implementation of phase IV. Instead of allocating a separate user descriptor pointer for each item's pinning region, as implied in Section 5.5.2, phase IV allocates an *array* of pointers for each E procedure. Similarly, it allocates an array of in-memory flags; the i-th user descriptor pointer is associated with the i-th in-memory flag. Each stack frame also includes a descriptor containing four elements:

(1) the address of the array of user descriptor pointers for this frame,

[60] This could certainly be improved in the future.

[61] When an E program starts, it opens a buffer group in the EXODUS Storage Manager. The user can specify the number of pages in this group by setting the environment variable EBGSIZE. Otherwise, the group size defaults to 50 pages.

(2) the address of the array of in-memory flags for this frame,

(3) the size of the two arrays, and

(4) a pointer to the descriptor in the next deeper stack frame.

Finally, a global variable points to the descriptor in the top stack frame. At procedure entry, the descriptor is initialized, and the global pointer is set to this vector. Figure 6.12 illustrates this arrangement for a hypothetical example.

There are two (related) problems that complicate our approach, one of which we have solved in a reasonable manner; although we have a solution to the other problem as well, it is not entirely satisfactory. The two problems are easily seen with an example. Consider the following code fragment:

```
x = y + f() + z;
```

Assume that y and z are both persistent. The compiler will produce (approximately) the following code from this statement:

```
pin(y); pin(z); x = **ud1 + f() + **ud2; unpin(y); unpin(z);
```

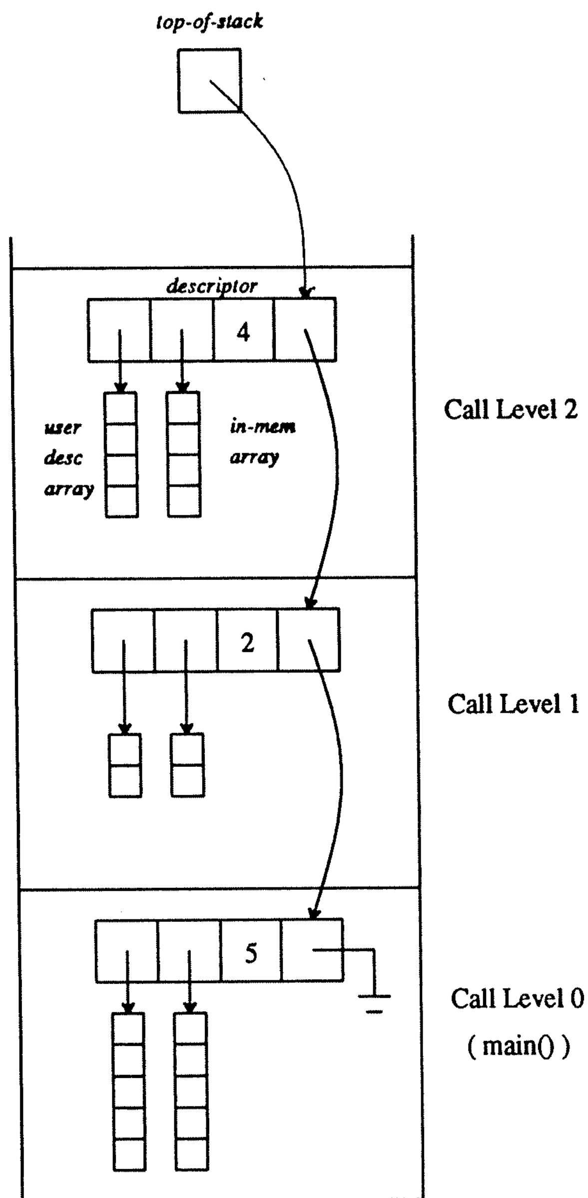
The first problem is that, within the call to f, we may attempt a pin operation that fails for lack of space. If the exception handler unpins either y or z, then depending on how the C compiler chooses to compile this statement, the program will either run normally, crash, or produce undetected errors. We can solve this problem by breaking the call to f out into a separate statement and assigning its result to a temporary variable:

```
_tmp001 = f();
pin(y); pin(z); x = **ud1 + _tmp001 + **ud2; unpin(y); unpin(z);
```

The second problem is not so easily solved. Assume in this example that pinning y succeeds but that pinning z fails. If the exception handler unpins y in order to pin z, we will again have an error. The current (less than ideal) solution is that the exception handler skips the top stack frame in its search for items to unpin. While this approach avoids the error just described, it also implies that the handler will fail to consider items in the top frame that really could be unpinned safely. As a result, there may be situations in which the exception handler will give up in failure prematurely.[62] A better solution would be somehow to recognize that, for a given region of the program, a *group* of items must be pinned simultaneously; the problem just described arises because we are considering items individually.

---

[62]Given that the search starts at the bottom of the stack, this event should be rare.




Descriptors for Use in Handling
"Out-of-Space" Exception

Figure 6.12

# CHAPTER 7

# AN INITIAL PERFORMANCE STUDY

The implementation of the E language raises some interesting performance questions. For example, how effective are the mechanisms described in Chapter 6 in reducing the number times an E program calls the Storage Manager? What causes the scheme to generate poor code, and how bad is the performance loss? What is the actual performance impact of using db types in *non*persistent applications? In this chapter, we present a study designed to provide initial answers to these questions by testing the quality of generated E code on a small set of examples. These experiments are not intended to be exhaustive; rather, they plot a few interesting points in the performance "space."

## 7.1. ORGANIZATION

In this study, we will describe a set of three sample programs. The basic approach was to write the same program in slightly different forms, to compile the different versions, and to compare their performance. For the first two sample programs, we will compare the performance of versions written in C++ and in E. For each program, there will be two E implementations; the first will simply change all types from the C++ implementation into db types, and the second will then make the program's data structure a persistent object. Finally, both E versions will be compiled with and without optimization. For the third of our sample programs, we will compare the performance of a program written in E with that of the same program hand-coded with direct calls to the Storage Manager.

These experiments will allow us to make several useful observations. First, by comparing the optimized and nonoptimized E versions, we can measure the effectiveness of our optimization strategy. Next, we may compare the (nonpersistent) E version of a given program against the C++ implementation to gain an initial estimate of the relative cost of using db types in nonpersistent applications. Finally, by comparing a hand-coded example against a compiled program, we can better understand what factors (besides pinning) are important in determining an E program's performance.

Of course, when measuring the performance of a system that handles persistent objects, we must be concerned with the cost of I/O. However, the I/O behavior of a program running on top of the Storage Manager is highly dependent on the state of the buffer pool and on the program's referencing pattern, neither of which is under

the compiler's control.[63] What the compiler *can* control is the number of pin operations that a program performs, and in this study, that is our main concern. Therefore, all persistent data structures are made small enough to fit into the Storage Manager's buffer pool; the size of the buffer pool for these tests is 150 4KB pages, or 600KB. The programs used in this study ensure that no I/O occurs during a run by traversing the structure once before beginning the time measurements. Two global counters maintained by the Storage Manager, `IO_DiskReads` and `IO_DiskWrites`, record the physical I/O activity during a program run. For all of the experiments reported here, these numbers indeed remained unchanged after the structure was initially faulted-in, confirming that no I/O was taking place.

The two statistics that we gather are the number of pin operations performed and the execution time for each program run. The compiler includes a command-line option that causes pin operations to increment two global counters. One counter tallies the total number of pin operations executed, while the other counts the number of pin operations that actually result in a call to the Storage Manager. The difference between these two gives the number of pin operations that pin in-memory objects. Time measurements are based on the `getrusage` system call. In order to minimize the effects of the clock's low resolution, we measure the execution time for multiple traversals of the given data structure. The times that we report in the following section are computed by dividing this execution time by the total number of nodes visited, i.e. by the number of nodes in the structure times the number of traversals. Finally, all experiments were run on a VAXstation 3200 running the BSD 4.3 UNIX operating system.

## 7.2. THE EXPERIMENTS

The three test programs represent three broad classes of data structures, and they correspond roughly to the kinds of structures commonly found in persistent applications. The three structures are: a directed graph with small, randomly connected nodes, a tree with large nodes and high fanout, and a flat relation containing many small tuples. Essentially, each program builds a data structure according to given parameters and then traverses the structure a specified number of times. Note that both graph traversal and tree traversal are also important tests of the effect of using db types for nonpersistent applications. The penalty for such use is associated with pointers; these structures consist almost solely of pointers, and traversing them is almost entirely a matter of pointer dereferencing.

### 7.2.1. Test I: Graph Traversal

The first program builds a random directed graph of 1000 nodes, each with a maximum out-degree of 5. This structure is characterized by having many small nodes and by being sparsely interconnected (that is, the number of

[63] One of the original goals of E was to provide the programmer with a structured means of declaring buffer group sizes and replacement policies. We have not yet addressed this issue, so every E program uses a single buffer group with LRU replacement.

nodes is much greater than the number of edges per node). This data structure is an important example, as it is likely to be quite common in such applications as CAD systems. Graph traversal is characterized by being highly recursive and by doing relatively little work at each node. This task is an important test case for the compiler since coalescing and propagation, being strictly *intra*procedural techniques, will have only limited opportunities to improve the program's performance.

All nodes in the graph are of the same same size; each comprises an integer id field and an array of five pointers to other nodes. For the C++ version, nodes are 24 bytes long, while for the db type version, nodes are 84 bytes each. The reason for this difference is that the size of a normal pointer is 4 bytes, while the size of an E db pointer is 16 bytes. The graph is built in a breadth first manner, starting at the root. That is, the graph building algorithm creates the root node, adds the root to a queue of nodes to be processed (the work list), and then enters a loop. Each iteration of the loop removes the node at the head of the queue and initializes its out-going arcs, possibly adding new nodes to the work list in the process. The loop terminates when, at the top of the loop, the work list is empty.

In selecting a destination node for a given out-going arc, the algorithm first decides (randomly) between creating a new node and selecting a node at random from those already in the graph. The algorithm allows a given node to point to another given node at most once, and it allows a node to point to itself. If a new node is created, it is added to the work list. For the nonpersistent versions of this program (both C++ and E), new nodes are allocated on the heap, while in the persistent E version, nodes are allocated in a collection. To ensure that the algorithm terminates, the probability of creating a new node drops to zero when the total number of nodes in the graph reaches a threshold; this threshold is supplied as a parameter to the program.

After generating the graph, we measure the cost of traversing it ten times, The traversal is depth-first, and we build a hash table of nodes already visited to avoid visiting a node twice. (Obviously, we clear the table after each of the ten traversals.) Table 7.1 summarizes the results of these traversals.

The first two columns of Table 7.1 show the parameter values that distinguish the different versions of the experiment. The first row shows the results for the C++ version. The remaining rows all refer to versions of the experiment that used db types. In these rows, "pst" indicates that the graph is persistent, and "opt" indicates that the optimization phases were included in compiling the program. The fourth row, for example, refers to the experiment in which the graph was persistent and optimization was turned off. The time per node is shown in column three. This number is the total execution time for the traversal divided by the number of nodes visited. In this case, execution times range from 2.5 to 32 sec, and the traversal (repeated 10 times) visits 10,000 nodes. Finally, columns four and five tally the number of pin operations performed per node during the traversal.

| Graph Traversal<br>No. Nodes = 1000, Max Out Degree = 5 | | | |
|---|---|---|---|
| expts | time per node ($\mu$sec) | pins per node in-mem | sm |
| C++ | 256 | 0 | 0 |
| ¬pst ¬opt | 712 | 8.2 | 0 |
| ¬pst opt | 590 | 1 | 0 |
| pst ¬opt | 3229 | 0 | 8.2 |
| pst opt | 1029 | 0 | 1 |

Results for Graph Traversal

Table 7.1

The first point to note in these results is that the code generator is successful in reducing the number of pinning operations per node (for the db cases). In the unoptimized cases, there are about eight pin operations per node visited. To account for these, we note that the traversal algorithm scans the array of pointers in each node, follows each non-null pointer, and stops when the first null is detected (or when the whole array has been processed). In unoptimized code, the test for a null pointer results in one pin operation, and following that pointer (if it is not null) results in another. Given that the average out-degree is about 3.6, then we should expect about 8.2 pins per node — two pins for each pointer plus one more pin to detect the final null.

In the optimized cases, each node is pinned only once. Given that that Storage Manager interface supports only object-at-a-time access, this figure is a lower bound on the number of pin operations. We note, however, that we achieve this lower bound only because the traversal involves exactly one procedure activation per node. Coalescing and propagation can reduce the number of pin operations to one within a procedure activation (as they do in this case), but they do not eliminate pin operations across procedure calls. Thus, if each node in the traversal were processed by several routines instead of just one, we would see multiple pin operations per node.

It is interesting to compare the benefits of eliminating pin operations for the persistent case relative to the nonpersistent case. While optimization reduces the number of pins by 88% in both cases, the execution time per node drops by only 17% for the nonpersistent case, as compared with 68% for the persistent case. The reason for this difference is simply that the cost of a pin operation is much greater when it involves calling the Storage Manager. In this case, therefore, a reduction in the number of such operations has proportionately greater influence on the program's overall performance. Pinning a nonpersistent object comprises a few boolean tests and a few pointer manipulations. Other costs, such as comparing db pointers or simply moving them around, are more important in the nonpersistent case.

This issue is further demonstrated when we compare the results of the nonpersistent db versions of the graph program with the C++ version. (The C++ version is also compiled by the E compiler.) Comparing the first three

rows of Table 7.1, we see that the execution time for unoptimized E code is greater than the C++ version by a factor of 2.8, while for optimized E code, this factor is reduced to about 2.3. Given that we reduce the number of pins per node to one and still have a program that takes twice as long to execute as the C++ version, it is likely that the remaining cost lies in manipulating db pointers. Besides dereferencing, the important operations on pointers are assignment, arithmetic, and comparison, all of which are more expensive when the pointer is a db pointer (i.e. a DBREF). The reason for this increased expense is partly intrinsic and partly an artifact of the current E implementation. For example, since moving 16 bytes takes longer than moving 4 bytes, db pointer assignment is quite a bit more expensive than normal pointer assignment. And, since a DBREF consists of an offset and an OID (which itself has internal structure), testing two db pointers for equality requires a series of comparisons rather than one. Currently, in fact, a procedure implements db pointer comparisons, thus adding the cost of a procedure call to each comparison.[64] Given that graph traversal consists almost entirely of pointer manipulations, this example represents a kind of worst case test of E. For a less pointer-intensive application, we can expect a less severe performance penalty; further experimentation would be needed, however, to explore this trend in detail.

It is possible to derive a rough estimate of the cost of a pin operation from Table 7.1. Consider rows two and three, which show the results for the experiments using nonpersistent db type objects. The same graph is traversed the same number of times in both cases; the only difference in the amount of work done is in the number of pin operations performed. Thus, the difference in execution time per node should be due solely to the extra pin operations performed in the unoptimized case. Dividing this time difference by the difference in the number of pins per node should thus give us the cost of a pin operation. This computation yields a cost of 16.9 μs per pin when the objects are nonpersistent, and it yields 305.6 μs per pin for persistent objects that are already in the buffer pool.

However, these pin costs should be considered as only approximate. The reason lies in the way that statistics were collected in the experiment. Recall from Chapter 6 that each pin operation is preceded by a boolean check to be sure that the item is not already pinned. The count of pin operations is incremented immediately after this check if the item is not currently pinned:

```
( (! pinned) ? (pin_count++, <pin>) : 0 )
```

In *un*optimized code, the pin is always performed[65], so the final count of the number of pin operations also counts the number of times that the program executes such boolean tests. This is not the case for optimized code, however. Optimized code executes the *same* number of boolean checks as unoptimized code, since the pin operations appear at the same points in the program in both cases, while performing fewer actual pins. Thus, optimized code is

[64]Making this comparison inline would be quite easy.

[65]The compiler ought therefore to eliminate the boolean checks in this case.

actually doing somewhat more work than is indicated by the number of pins recorded; furthermore, the amount of extra work (and therefore, the size of the error in estimating the pin cost) depends on the relative number of pin operations performed by the optimized and nonoptimized code.

Another factor that leads us to treat these numbers as approximations is that a given pin operation may or may not include pointer arithmetic, depending on the source program. Thus, the cost of pinning will appear to vary somewhat from program to program. Finally, as we mentioned earlier, the cost of pinning a nonpersistent object is of the same magnitude as other operations on db pointers; therefore the variation observed for nonpersistent applications could be significant.

### 7.2.2. Test II: Tree Traversal

The second test program builds and traverses a balanced, index-like tree structure. Such a structure is characterized by page-size nodes, each having many pointers. Like a graph traversal, a tree walk is also recursive; however, since more work is done at each node, we can expect coalescing to have a greater effect in reducing the overall execution time. For this study, we built a 2-level tree with a fanout of 100, giving a total of 101 nodes.[66] Again, we emphasize that the tree — like all of the other structures in this study — was intentionally made small enough to fit into the buffer pool. A fanout of 100 was small enough to allow us to build a complete tree of two levels within the buffer pool and yet large enough to allow "a lot" of work to be done at each node. The traversal algorithm walks the tree ten times; at each node, every pointer is examined, and if a given pointer is not null, the traversal descends through that pointer. (The pointers in the leaves, of course, are all null.)

Table 7.2 shows the results of the tree traversal experiment. In the unoptimized E case, there were 101 pins executed at each node. One hundred of these pins can be explained by the fact that the traversal compares every pointer in a node to NULL, causing each one to be pinned individually. The remaining pin arises from processing the root node. Since no pointer in the root is NULL, then each one will be pinned twice: once to make the comparison with NULL and once again as the traversal descends through that pointer. Thus, processing the root contributes 100 extra pins, which averages out to about one per node in the tree.

Again, optimization is able to reduce the number of pin operations to one per node. As expected, since the tree traversal does more work at each node, the performance improvement is greater in this experiment than in the graph example. In the persistent cases (rows four and five), optimization reduces the number of pin operations by 99% and the execution time by 93%. Again, however, the savings in execution time is proportionately less for the nonpersistent cases (rows two and three) due to the lesser cost of a pin relative to other (pointer-related) costs.

[66]Nodes are actually declared with a fanout of 200 to ensure that each node goes on a separate page.

| Tree Traversal Depth = 2, Fanout = 100 | | | | |
|---|---|---|---|---|
| expr | | time per node (μsec) | pins per node in-mem | sm |
| C++ | | | 521 | 0 |
| ¬pst | ¬opt | 3168 | 101 | 0 |
| ¬pst | opt | 1980 | 1 | 0 |
| pst | ¬opt | 33960 | 0 | 101 |
| pst | opt | 2316 | 0 | 1 |

Results for Tree Traversal

Table 7.2

There, the execution time drops by only 38%.

We can once again estimate the cost of a pin operation from this table, as we did for Table 7.1. For the nonpersistent case, this computation yields 11.9 μs per pin, while the persistent case works out to 316.4 μs per pin. As we explained earlier, these numbers are only approximate. Comparing them with the pinning times derived from Table 7.1, we see that two numbers for the nonpersistent case differ by almost 30%, a very significant margin, while the numbers for the persistent case differ by only 3%. This wide variation can be explained in light of the earlier discussion of error accumulation. Since the boolean test at the start of a pin operation is a significant fraction of the total cost to pin a nonpersistent object, the error in our pin cost estimate will be far more noticable for nonpersistent cases. Furthermore, as we explained, the greater the difference between optimized and nonoptimized code in the actual amount of pinning done, the greater the error accumulation. In the graph example, there was a difference of 7.2 pins per node, while in the tree example, there was a difference of 100 pins per node.

### 7.2.3. Test III: Relation Scan

The final data structure that we examine is a flat file (i.e. a relation), where we measure the costs of three processing steps: loading, scanning, and deleting. The load program creates 4500 tuples of 100 bytes each within an initially empty file. The scan program simply reads the first four bytes of each tuple in the file, and its execution is repeated ten times. Finally, the delete program destroys all of the objects one at a time. Since we have already demonstrated in the graph and tree examples that the optimizer becomes more effective the more work we do on each object, we will not repeat that result here. Instead, this experiment will do almost *no* processing of the objects: the load program creates uninitialized objects, the scan program only reads four bytes from each object (without interpreting those bytes in any way), and the delete program simply deletes each object.

The intention of this final experiment is quite different from the previous two. The graph and tree examples serve to measure the cost of using db types and to test the effectiveness of our optimizations. In contrast, this test

will measure the performance of file (`collection`) operations as currently implemented. Our approach here is to compare the performance of a program written in E against a version that we hand-coded in C++ with direct calls to the Storage Manager.[67]

Recall from Chapter 3 that a collection provides a typed abstraction of the files implemented by the Storage Manager. This abstraction is realized as a (generic) dbclass and supports the creation of new objects (with the **in...new** construct), the deletion of objects (with the **delete** operator), and the scanning of objects in a collection (with the `scan` iterator). Although knowledge of the `collection` class is wired into the compiler, the class methods themselves have been implemented in E. The `scan` method, for example, is an E iterator that yields a db pointer to each object in the file; it calls the Storage Manager routines `sm_GetFirstOid` and `sm_GetNextOid` to obtain the sequence of object addresses. Writing the methods in E had the advantage of being very easy to implement and to change as necessary. However, this design also places an extra layer of procedure calls around each Storage Manager operation. As we shall see, the overhead of this implementation has a significant effect on performance.

Table 7.3 presents the results of this last experiment. As in the graph and tree examples, the times given are in μs per object. We described the load, scan, and delete steps at the beginning of this section; the time in parentheses for the C++ scan case will be explained shortly. The obvious result from Table 7.3 is that file operations are significantly more expensive in E programs when hand-coded in C++. This overhead can be explained by several aspects of the `collection` implementation. The first problem, as we have indicated, is the extra procedure call overhead that results from implementing the `collection` methods as E routines. This overhead is somewhat worse for the scan method, as iterators are slightly more expensive to invoke than normal procedures.

| File Manipulations<br>No. Objects = 4500, Object Size = 100 bytes<br>(times indicate μs/object) | | | |
|---|---|---|---|
| expt | load | scan | delete |
| C++ | 384 | 480 (318) | 556 |
| E | 524 | 571 | 731 |

Results for Relation Experiment

Table 7.3

[67] Ideally, we would have hand-coded the graph and tree examples as well. Time constraints prevented this extension, however.

The fact that the collection class is written in E also leads to another, more subtle, source of overhead. The representation of a collection object itself is a pointer to the actual Storage Manager file, and the id of the file (FID) is embedded within the OID part of this pointer. Now, in order to invoke a Storage Manager operation on our file, we must extract the file's FID from the pointer and pass that FID in the call. However, since E does not allow a db pointer to be interpreted as anything but a pointer, there is no way (within the language) for an E program to treat a db pointer as a DBREF — even with casting. As a result, the collection methods each pass the pointer to a routine (written in C) that simply returns the pointer's value as a DBREF. This aspect of the implementation implies that there is not one, but rather several, extra procedure calls for each call to a Storage Manager routine.

For the operations of creating or destroying objects in a collection, it would be a fairly simple matter to have the compiler introduce inline code to call the appropriate Storage Manager routines. This would eliminate not only the call to the class method, but also the calls currently needed to convert between db pointers and DBREFs. Improving the scan performance, however, is not as easily accomplished.

The Storage Manager supports two means of scanning the objects in a file. One approach is for a client to use a series of `sm_GetNextOid` operations to obtain the sequence of object addresses; each object is then pinned with a separate call to `sm_ReadObject`. E programs scan collections in this way, although the part that obtains the object addresses (the `scan` iterator) is independent of the part that pins the object (the body of the iterate loop). The other means of scanning a file is to use an option in the Storage Manager interface that combines the calls to `sm_ReadObject` and `sm_GetNextOid` into a single call. That is, there is an optional argument to `sm_ReadObject` that allows the client to receive the OID of the next object following the one currently being pinned. This interface allows the client to scan a file with one Storage Manager call per object, as opposed to the two that are required in the first style of scan.

The column labeled "scan" in Table 7.3 shows the cost per object to scan the file using the `sm_GetNextOid` approach. The number in parentheses for the C++ case indicates the result of using the special interface; since the E program can only use the `scan` iterator, it has no corresponding entry. Clearly, the combined call provides a significant savings if one is able to utilize it. However, given the design of collections in E, this appears to be a difficult task. While the separation of the scan iterator from the loop that uses it provides a clean abstraction that was easy to implement, it also prevents us from using the special `sm_ReadObject` interface. As far as the (current) compiler knows, `scan` is just a normal iterator of unknown purpose. Furthermore, it is not clear that the compiler, even if it knew the semantics of the scan iterator, could combine the `sm_GetNextOid` calls with the `sm_ReadObject` calls that are in the loop body. One problem would be deciding which of the (probably many) pin operations should include the next oid request, as we would not want simply to add the request to every call. (A call to `sm_ReadObject` that requests the next oid is more expensive than one that does not; by adding the request to every pin operation, we could easily ask for the same "next" oid many times.) Another

problem is that there may not even *be* any pins in the loop body; the loop may simply pass the object pointers to a procedure. It thus appears that utilizing the faster scan interface would require quite a bit of special-case coding.

In a larger sense, however, it may be that the Storage Manager is not providing the appropriate mechanism for optimizing scan performance. The separation of control flow from processing — the basic motivation for iterators — is a useful programming methodology that has many practical benefits. The Storage Manager interface could conceivably provide efficient, explicit scan support while keeping the `sm_ReadObject` calls separate. For example, instead of `sm_GetNextOid`, a better call might be `sm_GetNextPageOfOids`, which would return an array of all the OIDs on the next file page. Better yet, the Storage Manager could provide a more explicit scan abstraction where, once the scan was started, getting the "next" OID would be very inexpensive (relative to the current `sm_GetNextOid` call).

## 7.3. SUMMARY

This chapter has presented the results of a first investigation into the performance of E programs. The exercise has been useful in demonstrating that the ideas developed in Chapters 5 and 6 can indeed improve the performance of E programs, sometimes dramatically. More importantly, however, it has pointed out areas where optimization is less effective than we would like, and it has revealed several aspects of the implementation of E that could be improved.

# CHAPTER 8

# CONCLUSIONS

## 8.1. THESIS SUMMARY

This thesis has presented the design and implementation of E, a persistent systems programming language. In Chapter 1, we introduced E and described its relationship to the other components of the EXODUS toolkit. We laid out E's major design goals and described the problems in the target programming domain that motivated them: the need for a query structuring mechanism, the lack of type information during system implementation, and the prevalence of disk-based data.

Chapter 2 presented a survey of some related languages, both past and present. The common feature that E shares with all of them is the provision of typed, persistent data. E differs from previous work in being a systems implementation language (as opposed to a high-level application language) and by having its model of persistence based on storage class (rather than on reachability). The implementation of E differs from other persistent language implementations in that its framework for managing I/O is based on static analysis.

In Chapter 3, we presented the main features of the E language design through a series of refinements to an example program. Since E is a superset[68] of C++, we began by showing a C++ implementation of the example, which was a binary tree index. Iterators were then introduced and shown to be a convenient mechanism for scanning the index in the presence of duplicate keys. Next we described generator classes, and we showed how they allowed the index code to be made independent of the key and entity types. Finally, we discussed database types, persistent variables, and collections, and we refined the example one last time so that the index became an object in the persistent store.

Chapters 4, 5, and 6 then described the implementation of persistence and code generation in version 2.2 of the E compiler. Chapter 4 presented an overview of the compiler's internal organization and explained our approach to several implementation issues, such as the representation of pointers, the processing of type declarations, and the implementation of persistent objects. Chapters 5 and 6 detailed the translation of E expressions into code to manipulate persistent objects. Chapter 5 outlined the basic organization of our code generation scheme, introduced the concept of an item, and showed how the program's syntax tree is altered to include calls to the storage layer for accessing persistent objects. Chapter 6 then described compiled item faulting, a

---

[68] With the exception of class nesting, as discussed in Section 3.2.2.

mechanism that reduces the number of run-time calls to the storage layer through a combination of static analysis and dynamic checking.

Finally, Chapter 7 presented the results of a small performance study. We examined a few selected E program examples in order to gain an initial idea of the quality of generated E code. The code generation scheme described in Chapter 6 was shown to provide significant performance improvement in the manipulation of persistent objects. This scheme was also shown to improve the performance of nonpersistent applications that use db types, although the speedup was not as great.

## 8.2. RETROSPECTIVE AND FUTURE WORK

Let us now consider the work accomplished so far, critique that work, and suggest directions for future research. Given that the preceding chapters have emphasized the positive features of the language, here we deliberately concentrate on E's shortcomings. We consider each of three major areas: the language design (both syntax and semantics), the compiler implementation, and the programming environment.

### 8.2.1. Language Design

#### 8.2.1.1. Db Types Versus Non-db Types

As we stated in Section 3.5.1, the reason for having db types in the language is to allow nonpersistent C++ data structures to be manipulated with no loss of performance. As Chapter 7 showed, the difference in performance between a C++ program and the same program implemented in E with db types can be significant. While the separation of db types from non-db types may thus be a good idea, having two sets of keywords for describing them is somewhat awkward. More importantly, however, db types present certain problems related to the semantics of pointers. The current language design allows the assignment of a non-db pointer to a db pointer, but not vice versa. The reason for accepting assignment in the one direction is to allow, for example, passing a string literal to a routine that expects a `dbchar*`. The reason for disallowing assignment in the other direction is that it is not clear what such an assignment should mean in the case where the db pointer references a persistent object.

The difference between db and non-db pointers gives rise to another problem when combined with un-type-checked function calls, a feature derived from C and supported by C++ and E. If a db pointer is passed to a routine that expects a virtual pointer (or vice versa), disaster ensues. The compiler can do nothing to prevent such errors, since the function's declaration provides no basis for rejecting the call. Library routines such as `printf` and `scanf` are particularly painful examples. For instance, if `p` is a `dbchar*`, then `printf("%s", p)` will cause an error, since the call will pass a DBREF instead of a `char*`. The fact that `printf("%c", *p)` works only confuses the matter more. There are several options for improving this situation, and at the very least, the compiler could warn the programmer that a db pointer is being passed in an unchecked function call.

#### 8.2.1.2. Strings and Other Variable Size Types

One feature that was part of the original language design was support for variable-length arrays (our so-called varrays) [Rich87]. Such a feature might be useful for implementing sets and strings, for example. These arrays were intended to support dynamic resizing, reflecting the capability of the underlying Storage Manager objects to grow and shrink. While the idea is simple enough at first glance, complications arise if we wish to allow varrays to be composed with other type constructors, for we must then manage an arbitrary nesting of variable length objects. After much consideration (and even an implementation outline [Rich88]), the idea was tabled in favor of addressing the other, more fundamental implementation issues presented in this thesis.

It has recently been suggested, however, that variable-length objects be reintroduced into the language, but in a more limited form [Solo89]. Specifically, there should be a mechanism for declaring an array whose dimension is not bound until creation time; once created, however, the array size remains fixed. Such an abstraction has appeared in other languages such as Algol-60 and Ada, is easier to implement than fully dynamic varrays, and would probably meet many of the needs for which one might have used a varray. The graph program in Chapter 7 is a good example. Instead of a compile-time binding of the out-degree of every node to some maximum, we could instead determine this number for each node at the time of its creation. Adding a flexible array language feature to E could be important, not only for its convenience, but also for performance reasons. Currently in E, if one wishes to include, for example, a variable length string as a data member of a dbclass, that member must be implemented as a pointer to a different object that contains the string. If the string were implemented as a flexible array of characters, then the compiler could allocate the string at the end of the same object that contained the class instance, embedding the string's offset within that instance.

#### 8.2.1.3. Generators and Inheritance

The current language design overloads the syntax for deriving subclasses to serve also as the syntax for instantiating generators. Furthermore, one may ***only*** instantiate a generator via this syntax. To declare a stack of integers, for example, one must first instantiate a class, e.g. `intStack`, and then declare an `intStack` instance. A cleaner, more natural design would be to allow instantiation within the context of a variable declaration. For example, the following declaration would declare `s` to be an integer stack instance:

```
stack[ int ]      s;
```

This style of instantiation syntax is found in CLU [Lisk77] and Trellis/Owl [Scha86]. Of course, one may still introduce a simple name for the type with a typedef, for example:

```
typedef stack[ int ]      intStack;
```

This declaration introduces `intStack` as a synonym for `stack[ int ]`.

Also important is the more general problem of the relationship of generators to inheritance. The existing E semantics in this area are not clean. For example, despite the current syntax, a class instantiated from a generator is *not* in any sense a subtype of that generator. Furthermore, while one may derive a subclass from an instantiated class, one may not derive a generator from a generator, nor a generator from a non-generator class. These issues have been examined previously [Meye86, Scha86] and need to be addressed in the context of E.

### 8.2.2. Compiler Implementation

#### 8.2.2.1. Alias Analysis and Other Optimizations

As we stated in Chapter 6, version 2.2 of the E compiler does not implement alias analysis, nor does it perform interprocedural analysis. Such techniques could help to improve the performance of E programs in several ways. First, by providing a more accurate estimate of the side effects of a given assignment or function call, they may make it possible to eliminate some of the unpin operations currently scheduled in phase II. Second, if it can be determined that two different pointers reference the same object, we may be able to coalesce their pinning ranges. Such an optimization would extend the current notion of coalescing, which considers only items based on the *same* pointer.

E can also benefit from other conventional optimizations. One important technique is the detection of loop induction variables. By knowing that the increment to a pointer or an array index is the same in every iteration of a loop, the code generator can be more intelligent in pinning objects based on such items. Currently, an array is either pinned in its entirety, or it is pinned one element at a time, depending on whether the array size falls below or above the compiler's coalescing threshold. The number of pin operations performed in processing an n-element array therefore jumps from O(1) to O(n) as the array size crosses this threshold. Detecting induction variables would allow the compiler to restructure a given loop so that it processes the array in large chunks. For example, the following E loop looks for the end of a character string:

```
while( *p != '\0' )
       p++;
```

Noting that p is incremented exactly once in each iteration, the compiler could transform this statement into a nested loop. Each iteration of the outer loop would pin a large chunk of bytes which the inner loop would then process.

#### 8.2.2.2. Other Performance Enhancements

In addition to — and orthogonal to — potential compiler optimizations, there are other routes to improving the performance of E programs that would be interesting to explore as well. For example, any improvement to the performance of the storage layer (particularly on a buffer hit) would be reflected directly in better performance for E

programs. We might consider, for example, adding another (faster) layer of caching above the existing interface. An alternative route would be to bypass the top layer of the Storage Manager altogether and to have E code communicate with the next layer down. In addition (hopefully) to reducing the cost of a pin operation, such an interface would also open up other new possibilities for the compiler. For example, while the current interface only supports object-at-a-time access, the next layer down presents opportunities for page-at-a-time accesses. For certain kinds of operations (e.g. scanning a relation), page-at-a-time access might provide significantly better performance.

#### 8.2.2.3. A Hybrid Approach

In describing E's approach to code generation in Chapter 6, we contrasted compiled item faulting with dynamic object faulting. The E compiler reduces the number of calls to the storage layer via static analysis together with simple run-time checking. In contrast, dynamic object faulting reduces the number of calls to the storage layer via pointer swizzling and extensive run-time support. It is not clear that these two approaches are necessarily incompatible. Rather, it may be possible to develop a hybrid approach that combines some form of pointer swizzling together with static analysis to achieve the "best" of both worlds.

#### 8.2.2.4. Generators

An important aspect of implementing E that was not covered in this thesis is the way in which generators are instantiated with actual parameters. Following CLU [AtkR78], the methods of an E generator class are compiled separately and shared by all classes instantiated from it. This decision was motivated by the expectation that E generators would be used to build frequently instantiated packages, such as access method code. Since such packages tend to be quite large, a code sharing approach seems appropriate.

There is, however, an unfortunate interaction between the code produced for a generic method and the code produced for db types. A method for a generic class often does not know the offset of a given data member of the (actual) class until run-time. As a result, the compiler translates references to such members into expressions involving pointer arithmetic and dereferencing. Since the code generation phase for persistence occurs after the code generation phase for generators, it is not able to distinguish these pointer manipulations from others. As a result, the compiled code will execute many more pin operations than necessary.

There are several possible approaches to solving this problem. For small generator classes (e.g. `stack`), perhaps a better solution is to provide macro expansion as an implementation option for instantiating a generator. (That is, an instantiated type would essentially reproduce the generator code with the actual type names in place of the formal parameter names, and the result is then compiled.) The interaction problem between compiler phases would not arise here, as the methods received by the code generator would then be type-specific. Such an implementation is probably inadequate for the large generator classes used to implement database access methods,

however, and other solutions must be sought. One approach would be to pass more information between the compiler phases, e.g. by tagging generator-induced offset calculations as such, so that the code generator for persistence could produce better code.

### 8.2.3. Programming Environment Support

Currently, E programs are built according to the same paradigm used for building C or C++ programs. That is, the program exists as a collection of independent files in the operating system, and the fact that one module has any relation to another is (at best) maintained with a facility such as *make* [Feld79]. The shortcomings of this approach are well-known, and integrated programming environments have recently received much attention [Comp87, Soft87]. For a number of reasons, some of which we mentioned in Section 4.4, E has a particular need for a more structured environment in which to compile and run programs.

#### 8.2.3.1. Classes as Objects

Perhaps the most serious problem with the current E environment stems from the C model of defining and using types. That is, type definitions are stored in header (.h) files, while the code to implement the types is stored in separate source (.c) files. An application program that wishes to use a type includes the text of the type's .h file during compilation, and then later the resulting object module is linked with the object modules derived from compiling the type's methods. With luck, then the .h file that was included when compiling the application is the same .h that was included when compiling the type's methods. When the modules are linked together, we must further hope that the symbols are bound to objects of the correct type; the linker does not care about matching types, only matching names.

The basic problem with the semantics of file inclusion is that a type has no existence outside of a compilation session; as far as the compiler is concerned, the type is created for the first (and only) time every time its header file is included. While this looseness can cause problems in any system-building environment, it can prove disastrous for persistent data. For instance, if a program compiled with one version of a type creates a persistent object, and another program compiled with a different version of the type then accesses the object, the result is likely to be unpleasant. Section 4.4 also mentioned another problem that file inclusion presents for E. In order to support virtual functions for persistent objects, the compiler must generate a unique tag for a given type, and furthermore, it must always generate the same tag for that type. Thus, the compiler really needs to be able to distinguish the first time it sees a type from subsequent occurrences. In other words, there needs to be a semantic distinction between *creating* a type and simply *using* it. (In the former case, the compiler generates a new tag, and in the latter, it reuses the existing one.)

Both of these problems suggest that file inclusion as a means of using types does not provide the appropriate semantics for a persistent language. Similar to the way programs use definitions in Modula-2 [Wirt82] for example, an E program might declare its intention to use a particular type via some kind of import statement. An associated E environment would allow programmers to add new types to various libraries and would track dependencies between the types. In addition, such an environment would maintain the association between a persistent object and its type to ensure that an application can access an object only if it really linked to the correct type.

#### 8.2.3.2. Schema Evolution

One very important requirement of an environment to support a persistent language is the need to manage change, especially in the definition of types. The requirements are more stringent here than in a Modula-style environment, as the presence of persistent objects raises a number of additional issues: if the definition of a type is changed, what should be done with the extant objects of that type? Should they be left alone, should the be discarded, or should the system attempt to evolve them automatically? Should we allow types to change at all, or should we only allow the addition of new versions of types? If we do allow versions, what is the relationship between one version and another? Can a single program access objects of different versions of the same type? The problem of schema evolution is an interesting research topic that could be addressed in the context of building an environment for E. Work in the area of object-oriented database systems has produced several papers on the subject [Skar86, Penn87, Bane87], but the problem is far from solved. E presents some unique additional challenges because the programmer has much more control over the physical layout of objects than is provided by an OODBS.

#### 8.2.3.3. Debugging Support

Currently, debugging E programs is, to be kind, cumbersome. Two factors contribute to this problem. First, E code is translated into C, and the symbols known to the debugger are those produced by the C compiler. Moreover, in order to output legal C code, the E compiler must alter function and variable names in certain cases, e.g. to produce a set of C functions with unique names from a set of E functions with overloaded names. As a result, the symbols available during a debugging session are often quite different from the names declared in the source code. The second problem concerns db type objects (both persistent and nonpersistent). Since the debugger knows nothing of persistent objects, we cannot print a persistent variable by simply giving its name. Furthermore, the programmer cannot give the usual dereferencing syntax to print the data at the end of a db pointer, as the compiler has converted all such pointers into DBREFs. Clearly, better debugging support is needed if E is to be used for building large systems. At a minimum, such support would include an E expression interpreter.

## 8.3. CONCLUSION

We have made much progress in the design and implementation of E. We have shown the language to be a powerful programming tool, and we have demonstrated that the compiler can produce good code. We believe that the language design and compilation techniques developed in this thesis are important contributions to the area of persistent programming. As this chapter has shown, however, there are several areas in which the language design can be improved, and there are many possible avenues for pursuing further improvements both in the quality of generated E code and in the environment support provided to the E programmer. Finally, and perhaps most importantly, ideas for future improvements will also come from E programmers as they gain more experience with the language.